Constraints
https://doi.org/10.1007/s10601-023-09366-z

A feature commonality-based search strategy to find high *t*-wise covering solutions in feature models

Mathieu Vavrille[1]

Accepted: 21 November 2023

Abstract
t-wise coverage is one of the most important techniques used to test configurations of software for finding bugs. It ensures that interactions between features of a Software Product Line (SPL) are tested. The size of SPLs (of thousands of features) makes the problem of finding a good test suite computationally expensive, as the number of *t*-wise combinations grows exponentially. In this article, we leverage Constraint Programming's search strategies to generate test suites with a high coverage of configurations. We analyse the behaviour of the default random search strategy, and then we propose an improvement based on the commonalities (frequency) of the features. We experimentally compare to uniform sampling and state of the art sampling approaches. We show that our new search strategy outperforms all the other approaches and has the fastest running time.

Keywords Feature model · Commonality · Constraint programming · Search strategy · Test coverage

1 Introduction

Efficient testing of Product Lines is of high importance to assess quality, or the absence of bugs (in the case of Software Product Lines) [1]. In highly configurable systems, this testing task is complicated by the large number of interacting features. For example, the Linux kernel contains thousands of interacting features (such as compilation options or installed libraries) [2]. Configurations (i.e. sets of features) can be tested by instantiating them on the given product line (for example, compiling the Linux kernel with specific options and libraries). These tests can be expensive (in terms of running time [2], memory [3], or manpower [4]), so efficient test suites (a set of configurations) need to be generated.

A way to measure the quality of a test suite is the *t*-wise coverage [5]. It aims to ensure that all interactions (combinations) of up to t features are tested. But there can be $2^t\binom{n}{t}$ t-wise combinations on n features. Thus, with thousands of features, computing the t-wise combinations allowed by the product line can be prohibitive, let alone generating a minimal test suite

✉ Mathieu Vavrille
mathieu.vavrille@univ-nantes.fr

[1] LS2N, UMR 6004, Nantes Université, École Centrale Nantes, CNRS, f-44000 Nantes, France

that covers all these combinations. To overcome this issue, approaches were designed using approximations based on random processes such as uniform [6] or weighted [7] sampling. These approaches lose the guarantees, but the diversity induced by the randomness allows for good experimental coverage and running times.

In this article, we use Constraint Programming's random search strategies to find high-coverage test suites. Search strategies are a way of making the search find solutions in different solution spaces. In particular, random search strategies do not require to compute expensive metrics (such as the number of allowed combinations) and can generate diverse (i.e. high coverage) solutions. The contributions of this article are as follows.

- We analyse the theoretical properties of the default random search strategy. We show that the (non-uniform) distribution of solutions returned by this default random search strategy is well suited to the task of computing solutions with good t-wise coverage.
- We design an improvement to this search strategy by using information on the product line, namely the commonality. The commonality of a feature is the number of times it appears in all the possible configurations. We use this information to make better choices during the decisions of the search strategy, to find solutions that cover more unseen combinations.

We experiment these two search strategies and compare them to state-of-the-art sampling approaches. We show that the search strategies outperform all other approaches in the t-wise coverage and running time. Our new approach improves the default random search strategy without any running time overhead.

This article is structured as follows. Section 2 defines the notions used in the rest of the article and section 3 presents the related works. Section 4 presents an analysis of the RandomSearch strategy and our new strategy to find good coverage solutions. Finally, section 5 presents the methodology and the results of the experiments.

2 Preliminaries

This section introduces the notions used in the article.

2.1 Constraint programming

Constraint Programming (CP) is a programming paradigm that allows a user to state a problem by describing the relations (constraints) that hold on some variables (unknowns). A Constraint Satisfaction Problem (CSP) is defined as follows.

Definition 1 (Constraint Satisfaction Problem (CSP) [8]) A *Constraint Satisfaction Problem (CSP)* $\mathcal{P}$ is a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where:

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables;
- $\mathcal{D}$ is a function that associates a domain to with each variable;
- $\mathcal{C}$ is a set of constraints, where each constraint is a relation over a subset of variables of $\mathcal{X}$.

A CP solver is an algorithm that finds a solution to the problem, i.e. an instantiation of the variables that satisfies all the constraints. To find solutions, CP solvers usually alternate between two steps. First, there is a propagation phase that checks the satisfiability of the constraints of the CSP. This phase can also filter (i.e. remove) values in the domains of

variables that cannot appear in solutions. Then, if some domains are not reduced to singletons, the search space is reduced by adding unary constraints (called decisions) in a depth-first search style. A decision is, for example, the assignment of a value to a variable.

The search strategy, which chooses which decisions are made during the search, has a major impact on the efficiency of the solver. Many different algorithms have been proposed to find a good variable-value pair. Some search strategies are designed to be efficient for many types of problems (black box search strategies), such as `FRBA` [9] and $\texttt{wdeg}^{\texttt{ca.cd}}$ [10], but for some specific problems a tailored strategy can improve the running time, such as `SetTimes` [11] for scheduling problems.

In this article we are interested in the RANDOMSEARCH search strategy. It is a search strategy that picks (uniformly) at random an uninstantiated variable, and instantiates it to a random value in its domain. Once a solution is found the search must be restarted, otherwise close solutions will be returned. Due to its random behaviour, it searches in diverse solution spaces, and therefore returns diverse solutions. Section 4.1 quantifies this notion of diversity granted by RANDOMSEARCH.

2.2 Feature models

Feature models are graphical and condensed representations of the products of a product line [12]. Given a fixed set of features $\mathcal{F}$, a feature model is a pair of, first, a feature diagram, which gives a hierarchical structure of the features organization, and second, a conjunction of propositional formulas over $\mathcal{F}$. We use the recursive definition of feature diagrams presented in [13].

Definition 2 (Feature Diagram) A *feature diagram* is an n-ary labelled tree, where the nodes can be of different types. A feature diagram D stores a feature $D.\texttt{feature} \in \mathcal{F}$ at its root. The children can be from:

- a `mandatory/optional` group, with the sets $D.\texttt{mand}$ containing the mandatory children and $D.\texttt{opt}$ containing the optional children,
- an exclusive (`xor`) group, with the set $D.\texttt{xor}$ containing the children,
- an `or` group, with the set $D.\texttt{or}$ containing the children.

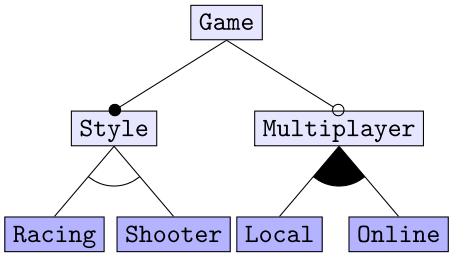
In addition, each feature may only appear once in a feature diagram.

Example 1 Figure 1a shows a feature model representing video games. It is mandatory that a game has a `Style` (black dot above the `Style` node), and optionally a `Multiplayer` mode (empty dot above the `Multiplayer` node). The `Style` can be either a `Racing` game or a `Shooter` game, but not both (represented by the arc between the two nodes). If there is a `Multiplayer` mode, it can be `Local` or `Online`, or both (represented by the black arc between the two nodes). There is also a constraint stating that a `Shooter` must have an `Online` mode.

Table 1b shows the configurations allowed by this feature model. The last two configurations (7 and 8) are allowed by the feature diagram, but not by the cross-tree constraint.

As shown in Figure 1a, feature models are defined using a tree structure, the feature diagram, and cross-tree constraints. Feature diagrams define the hierarchical structure of features in a feature model.

We call D' a sub-feature diagram of D if D is an ancestor of D' or D itself. Feature diagrams constrain the allowed configurations (set of features) of the feature model.



Cross-tree constraints:
`Shooter` $\Longrightarrow$ `Online`

(a) Example of Feature Model

	1	2	3	4	5	6	7	8
Game	∗	∗	∗	∗	∗	∗	∗	∗
Style	∗	∗	∗	∗	∗	∗	∗	∗
Racing	∗	∗	∗	∗				
Shooter					∗	∗	∗	∗
Multiplayer	∗	∗	∗		∗	∗	∗	
Local	∗		∗		∗		∗	
Online	∗	∗			∗	∗		

(b) Allowed configurations. There is one column per configuration, and a ∗ means that the feature is present. Configurations 7 and 8 are allowed by the feature diagram but not by the cross-tree constraint.

Fig. 1 A feature model and its set of allowed configurations

Definition 3 (Allowed Configuration) Given a feature diagram D, a configuration $C \subseteq \mathcal{F}$ is allowed iff:

- $D.\texttt{feature} \in C$
- for all D' sub-feature diagrams of D, $\forall D'' \in D'.\texttt{children}$, $D''.\texttt{feature} \in C \Rightarrow D'.\texttt{feature} \in C$
- for all D' sub-feature diagrams of F, if $D'.feature \in C$ then:
  - $\forall D'' \in D'.\texttt{mand}$, $D''.\texttt{feature} \in C$
  - $\exists D'' \in D'.\texttt{or}$, $D''.\texttt{feature} \in C$
  - $\exists! D'' \in D'.\texttt{xor}$, $D''.\texttt{feature} \in C$

We denote the set of allowed configurations by $Sols(D)$.

Informally, all features in $D.\texttt{mand}$ children must be taken, at least one feature in the $D.\texttt{or}$ group must be taken, and exactly one feature in the $D.\texttt{xor}$ group must be taken. When a feature is taken, its parent feature is also taken.

To allow more expressiveness when modelling feature interactions, feature diagrams are extended with propositional formulas that allow modelling interactions between features that are not ancestors of each other.

Definition 4 (Feature Model) A *Feature Model* M is a pair $\langle D, \psi \rangle$ where D is a feature diagram and ψ is a boolean formula where variables are features contained in $\mathcal{F}$. A configuration is allowed by M if it is allowed by D and satisfies the boolean formula ψ. We note $Sols(M)$ the set of allowed configurations.

The propositional formulas allow for more diverse constraints, but also make the problem much harder, as simply finding one configuration is NP-complete. Some definitions restrict the cross-tree constraints to implication or exclusion of features [12], but this has been shown to reduce the expressiveness [14].

Feature models can be translated into CNF formulas, where an instantiation corresponds to a unique configuration [15]. The model defines one variable X_f for each feature f. The conversion of propositional formulas can lead to an exponential number of clauses [16]. To avoid this exponential explosion, CP was used in [17, 18] (for extensions of feature models to integer variables and global constraints).

2.3 *t*-wise coverage

To test Software Product Lines, a test suite should be generated.

Definition 5 (Test Suite) Given a feature model M, a test suite $\mathcal{S}$ is a list of configurations of M.

Remark In this definition, $\mathcal{S}$ is a list, because the order of the configurations is important (the first configurations in $\mathcal{S}$ are tested first). Also, the same configuration can appear twice in $\mathcal{S}$. In the case of testing a deterministic product line this is detrimental to the test suite (because the same configuration is tested twice). In the case of a non-deterministic product line (for example a software with randomness), it may be useful to test multiple times the same configuration to increase the confidence in the test. However, for the metric we use in this article, there is no benefit to testing the same configuration twice.

In this article, the metric used to assess the quality of the test suite is the t-wise coverage. This metric focuses on testing interactions of t features through combinations.

Definition 6 (t-wise combination) A t-wise combination is a mapping $\sigma : \mathcal{F}' \to \{0, 1\}$ with $\mathcal{F}' \subseteq \mathcal{F}$ and $\left|\mathcal{F}'\right| = t$.

A configuration C covers a t-wise combination σ iff $\forall f \in \mathcal{F}', \sigma(f) = 1 \Leftrightarrow f \in C$. In this case we say that σ is included in the configuration, and write $\sigma \subset C$. We note $Comb_t(C)$ all the t-wise combinations covered by a configuration C (when there is no ambiguity on the value of t we omit it in $Comb_t$). By extension, given a test suite $\mathcal{S}$, $Comb(\mathcal{S})$ is the set of combinations covered by at least one configuration of $\mathcal{S}$ (i.e. $Comb(\mathcal{S}) = \bigcup_{C \in \mathcal{S}} Comb(C)$).

Given a feature model M, a combination is said to be possible if there is a configuration in $Sols(M)$ that covers it. For simplicity we note $Comb(M) = Comb(Sols(M))$ all the combinations covered by at least one configuration allowed by M. The coverage of a test suite is the fraction of possible combinations that are covered, i.e.

$$Cov(\mathcal{S}) = \frac{|Comb(\mathcal{S})|}{|Comb(M)|}$$

As t grows, the number of t-wise combinations grows exponentially. Indeed, the number of possible combinations can be as large as $\binom{|\mathcal{F}|}{t} 2^t$. The number of combinations covered by a single configuration also grows exponentially as t grows, and is equal to $\binom{|\mathcal{F}|}{t}$.

Ideally, all interactions of features are tested, so that all $|\mathcal{F}|$-wise combinations are covered, but in practice this is impossible due to the exponential growth of the number of combinations. A study by the NIST [1] states that most of the faults/bugs in software come from up to 6-wise combinations. This greatly reduces the number of combinations to test, but for large feature models it would still not be reasonable to try to enumerate all the possible 6-wise combinations.

Remark In the following, we say that a test suite $\mathcal{S}$ has a "good" t-wise coverage when $|Comb(\mathcal{S}|$ is high, or equivalently when $Cov(\mathcal{S})$ is close to 1. This is one of the metrics we use in the experiments in Section 5 to assess the quality of a test suite and compare approaches.

2.4 Links between commonalities and uniform sampling

This section recalls the properties of t-wise coverage of uniform samplers stated in [19]. A sampler is a random selection process where the result is not deterministic.

Notation Given a random variable Z taking values in a set $\mathcal{S}$, for $s \in \mathcal{S}$, "$Z = s$" is the random event of Z taking the value s. We note by $\mathbb{P}(Z = s)$ the probability that this event happens. If the random variable Z takes countable values (in $\mathbb{N}$ or $\mathbb{R}$), its expected value is noted $\mathbb{E}(Z)$.

On a feature model M, a sampler $\mathcal{U}$ generates a random allowed configuration, i.e. $\mathcal{U}(M)$ is a random variable taking values in the set $Sols(M)$. A uniform sampler guarantees that each solution is equally likely to be selected, and can be formalised as follows.

Definition 7 (Uniform Sampler) Let M be a feature model. A function $\mathcal{U}$ is a uniform sampler on M iff

$$\forall C \in Sols(M), \mathbb{P}\left(\mathcal{U}(M) = C\right) = 1/\left|Sols(M)\right|$$

Furthermore, multiple calls to $\mathcal{U}(M)$ should be independent, i.e. the i-th configuration returned by $\mathcal{U}$ should be independent from all the $i - 1$ previous configurations.

Applied to configurations of feature models, uniform sampling can generate a test suite. It has already been used on feature models in SMARCH [6] and extended to weighted sampling in BAITAL [7]. There is no guarantee of t-wise coverage, but there is no need to compute the exponential set of all t-wise combinations: diversity is provided by randomness.

When using random algorithms, since there are fewer guarantees, it is important to have information on the average behaviour. The average behaviour of uniform samplers on the t-wise coverage was studied in [19]. The authors found that the t-wise coverage depends on the commonalities of the combinations, defined as follows.

Definition 8 (Commonality) The *commonality* of a combination σ in a feature model M, noted φ_σ, is its frequency of appearance in the set of allowed configurations, i.e.

$$\varphi_\sigma = \frac{\left|\{C \in Sols(M) \mid \sigma \subset C\}\right|}{\left|Sols(M)\right|}$$

Commonalities give important information about the feature model. It allows to know which combinations are more present in the set of allowed configurations. A user may want to design a test suite that covers the frequent combinations, as these may be the most used, or conversely a user may want to focus on low commonalities to test combinations that may have been missed by other tests.

The problem of computing the commonality of a configuration is hard because it requires calls to a #-SAT solver. For example the strategy 3 of baital [7] makes $|\mathcal{F}| + 1$ calls to a #-SAT solver to compute all the commonalities of features. On large feature models this can be prohibitively expensive. If the cross-tree constraints are dropped (only the feature diagram is considered), it is possible to compute the commonalities for all features (1-wise combinations) in linear time [13, 20]. This quickly gives an approximation of the commonality of the features.

Uniform samplers guarantee that all solutions have the same probability of being returned. For t-wise coverage, however, we are interested in the probability that a t-wise combination will be returned by the sampler. The following proposition states that this probability is equal to the commonality of the combination.

Proposition 1 ([19]) *Let M be a feature model, $\mathcal{U}$ be a uniform sampler (i.e. $\forall C \in Sols(M), \mathbb{P}\left(\mathcal{U}(M) = C\right) = 1/\left|Sols(M)\right|$), and σ be a combination, then*

$$\mathbb{P}\left(\sigma \subset \mathcal{U}(M)\right) = \varphi_\sigma\,.$$

The probability of a given combination being in the solution returned by a uniform sampler is equal to the combination's commonality. This means that if there are features with very low commonality, a sampler may never return a solution that contains them in a reasonable number of samples. For example, the authors in [19] remarked than in their experiments, 38.8% of the features (1-wise combinations) have a commonality $\varphi_\sigma < 0.0001$, so it is very unlikely that a uniform sampler will produce a solution containing these features.

Notation Given a sampler $\mathcal{A}$ (uniform or not), a feature model M, and a t-wise combination σ, we note $p_\sigma^{\mathcal{A}}(M) = \mathbb{P}(\sigma \subset \mathcal{A}(M))$ the probability that the sampler $\mathcal{A}$ returns a solution that covers the combination σ. In the following, when there is no ambiguity in the feature model, we simply write $p_\sigma^{\mathcal{A}}$. In this article we consider two types of samplers. For a uniform sampler, noted $\mathcal{U}$, Proposition 1 states that $p_\sigma^{\mathcal{U}} = \varphi_\sigma$. For a sampler based on the RandomSearch strategy, the probability $p_\sigma^{\mathcal{R}}$ is unknown, we analyse it in Section 4.1. In the following, an exponent $\mathcal{U}$ in p_σ denotes the use of a uniform sampler, an exponent $\mathcal{R}$ denotes the use of RandomSearch, and an exponent $\mathcal{A}$ denotes the use of any sampling algorithm (i.e. the property applies to all sampling algorithms).

For the task of t-wise coverage, if many combinations are unlikely to be found, the test suite would not have a good t-wise coverage. Let $\mathcal{S}_n$ be a test suite generated by calling a sampler n times independently on a feature model M. $\mathcal{S}_n$ is a random variable taking values in $Sols(M)^n$. The t-wise coverage of $\mathcal{S}_n$, $Cov(\mathcal{S}_n)$, is also a random variable taking values in $[0, 1]$. To evaluate the behaviour of a sampler in terms of t-wise coverage, we are interested in the expected value of $Cov(\mathcal{S}_n)$, i.e. $\mathbb{E}(Cov(\mathcal{S}_n))$. A formula for this expected t-wise coverage is given in [19].

Proposition 2 ([19]) *Let M be a feature model. Let $p_\sigma^{\mathcal{A}}$ be the probability that a sampler $\mathcal{A}$ (uniform or not) returns a solution containing the combination σ. Let $\mathcal{S}_n$ be a list of n configurations of a feature model M generated by a such sampler (by calling it n times independently). $\mathcal{S}_n$ is a random variable, and so is the set of t-wise combinations covered $Cov(\mathcal{S}_n)$. The expected value of $Cov(\mathcal{S}_n)$ is*

$$\mathbb{E}(Cov(\mathcal{S}_n)) = \frac{1}{|Comb(M)|} \cdot \sum_{\sigma \in Comb(M)} (1 - (1 - p_\sigma^{\mathcal{A}})^n) .$$

This proposition confirms the intuition that the expected t-wise coverage depends on the probability of sampling each combination. If there are combinations with a low sampling probability, the expected coverage will increase (as the number of solution increases) slower than if all the combinations had a high sampling probability. In Section 4.1 we prove a lower bound on the sampling probability $p_\sigma^{\mathcal{R}}$ when using RandomSearch.

3 Related works

3.1 Sampling based approaches

A well-known approach to searching for diverse configurations is to use randomness, and for example a uniform sampler. Recent advances in uniform SAT samplers such as Smarch [6] can efficiently generate configurations of large feature models. However, this approach does not give any guarantee of coverage and it has been shown experimentally that up to 10^{14} configurations (almost the enumeration of all solutions) need to be generated achieve a 100% coverage on some feature models in [19].

BAITAL [7] corrects this issue by modifying the weighted sampler WAPS [21]. This sampler compiles the cnf formula representing a feature model into a d-DNNF representation. This representation is then annotated with weights, and weighted sampling can be performed very efficiently. BAITAL will perform r rounds, each round drawing s samples from a given distribution. At the start of each round, the distribution is modified by changing the annotation of the d-DNNF representation depending on the solutions found. If a feature has only been sampled a few times, its weight is increased to increase the probability of sampling a configuration containing it. Each annotation phase is costly, but it helps to find new combinations. We compare our approach to BAITAL in the experimental section 5.

CMSGEN [22] is a recent SAT sampler that has been shown to generate test suites with higher coverage than BAITAL. CMSGEN modifies a SAT sampler to use the RANDOMSEARCH strategy (picking a random uninstantiated variable, and a random value between 0 and 1). This sampler is therefore non-uniform (this can even be shown on a problem with two variables x and y, and the clause $x \vee y$). In Section 4.1 we analyse the behaviour of RANDOMSEARCH on the task of test suite generation for t-wise coverage. We show why RANDOMSEARCH is well suited for this task, and at the same time explain the reasons for the great results of CMSGEN. RANDOMSEARCH (modified in the style of how CMSGEN modified the SAT solver) is also compared experimentally to the other approaches.

3.2 Dedicated approaches

t-wise coverage is a well-studied problem on feature models. Most approaches either require access to the set of possible combinations (by making $\binom{|\mathcal{F}|}{t}2^t$ calls to a SAT solver), or will iteratively generate this exponential set. These approaches often have the guarantee that all combinations are covered, at the cost of an expensive generation of combinations. AETG [23] is one of the first algorithms for t-wise coverage. The authors propose a way to select variables to set (or forbid) in the searched configuration based on the combinations not yet covered. ICPL [24] is based on the fact that a t-wise covering test suite is a good starting point for generating a $t+1$-wise covering test suite. This remark also allows to speed up the generation of possible $t+1$-wise combinations, as some of them were detected as impossible by the t-wise test suite. In IncLing [25], the authors propose several improvements (such as the detection of dead or core features and a feature ranking heuristic) in an incremental sampling. In [26], the authors propose to use advances in SAT solvers to detect impossible combinations more efficiently (instead of making a SAT call to verify each combination). By using unsatisfiability cores returned by SAT solvers on unsatisfiability, they can reduce the set of possible combinations. This greatly reduces the number of SAT calls required to find a covering test suite.

These approaches are very different from the one we use in this article (i.e. sampling). With a sampling approach there is no strong guarantee of coverage, but solutions can be found quickly. The sampling approaches can use statistical information (such as the commonalities) to increase the probability of sampling in an interesting space. On the other hand, dedicated approaches have the guarantee of generating a covering test suite, but at the cost of having to generate the set of possible combinations, which may be prohibitive for large feature models or large values of t. The focus is often on finding the *smallest* possible set of configurations that ensures the full coverage. These approaches benefit from information discovered during the search, such as unsatisfiability cores in [26]. In addition, these approaches are often deterministic. They use heuristics to choose which features to include or remove from the

configuration. These heuristics depend on the set of t-wise combinations already covered to find solutions that cover as many new combinations as possible.

3.3 In constraint programming

Constraint Programming provides a variety of modelling and solving tools (such as search strategies and global constraints). In PACOGEN [27, 28], constraint programming is used to find the smallest test suite that ensures full pairwise coverage. The authors propose a data structure (in a matrix) to store the complete test suite being generated, and a global constraint ensuring that the pairwise combinations are covered. As other approaches enumerating the combinations, this approach would not scale big feature models (with thousands of features).

Samplers have also been designed in the CP framework. In [29], the authors compile the constraint network into a MDD (Multi-valued Decision Diagram) tree representation. This approach is similar to the compilation of the d-DNNF representation in WAPS. Compiling the MDD is costly (in memory and in running time), but once the MDD is compiled, many operation can be performed very efficiently. In particular, the uniform sampling of a solution can be done in linear time in the size of the MDD.

Recently, the advances made in hashing-based SAT samplers (see the UNIGEN [30] line of work) hava been extended to CP. By adding random hashing constraints to the model, the set of solutions is randomly cut into small cells. In CP, a 2-independent family of hashing functions, the linear modular equalities, have been used to design a practically uniform sampler [31]. In the same idea, table constraints have also been used as hashing constraints in TABLESAMPLING [32]. A table constraint is a constraint given in extension, i.e. for a given subset of variables, all the allowed instantiations are given in the constraint. In a t-wise framework, this can be seen as allowing or disallowing some t-wise combinations on a given set of variables. We evaluate TABLESAMPLING in section 5 to see if such hashing constraints based on table constraints lead to good t-wise coverage.

4 Frequency difference search strategy

This section first presents an analysis of the behaviour of RANDOMSEARCH. This analysis gives a better understanding of the difference with uniform sampling. We then improve RANDOMSEARCH using knowledge from previously found solutions, to apply it specifically to the problem of t-wise coverage.

4.1 RANDOMSEARCH's Behaviour

As shown in Proposition 2, the t-wise coverage of a sampler $\mathcal{A}$ is related to its probability $p_\sigma^{\mathcal{A}}$ of sampling a given combination σ. In the case of a uniform sampler $\mathcal{U}$, Proposition 1 states that $p_\sigma^{\mathcal{U}} = \varphi_\sigma$. For a sampler $\mathcal{R}$ using the RANDOMSEARCH strategy, this probability $p_\sigma^{\mathcal{R}}$ is unknown: we study it here.

4.1.1 Example for a single feature

The main difference between uniform sampling and RANDOMSEARCH is that uniform sampling focuses on configurations, while RANDOMSEARCH focuses on features. At the decision

level, all features have the same probability of being selected (or removed). The toy feature model in Figure 2 shows this behaviour.

At the root node `Main`, this feature model has two exclusive children `Noise` and `Studied`. We are interested here in the `Studied` feature, which is only contained in the configuration {`Main`, `Studied`}. However, there are many other configurations when the feature `Noise` is selected. This feature has n optional children, so there are 2^n possible configurations.

On the one hand, it is very unlikely that a uniform sampler will generate the solution containing the feature `Studied` because it is flooded by other configurations. The exact probability of sampling feature `Studied` is $p^{\mathcal{U}}_{\texttt{Studied}} = \varphi_{\texttt{Studied}} = 1/(2^n + 1)$.

On the other hand, RANDOMSEARCH is much more likely to sample the feature `Studied`. The CSP representation of a feature model contains one variable per feature. We make the CSP explicit for the example 2.

Example 2 To represent the feature model given in Figure 2 as a CSP, one variable $X_{\texttt{F}}$ is created for each feature ($\mathcal{F} = \{\texttt{Main}, \texttt{Studied}, \texttt{Noise}, \texttt{N}_1, \ldots, \texttt{N}_n\}$). All the variables have a boolean domain. The constraints are:

- The root node must be selected:
$$X_{\texttt{Main}} = 1\,.$$
- A `xor` node (the `Main` node) states that if the parent is taken, only one child is taken:
$$X_{\texttt{Studied}} + X_{\texttt{Noise}} = X_{\texttt{Main}}\,.$$
- If an optional child is taken, then the parent must be taken:
$$\forall i \in \{1, \ldots, n\},\, X_{\texttt{N}_i} \Rightarrow X_{\texttt{Noise}}\,.$$

When all variables are instantiated, the combination can be retrieved by keeping all features whose variables take the value 1, i.e. if S is a solution to the CSP, the associated configuration is $C = \{\texttt{F} \in \mathcal{F} \mid S(X_{\texttt{F}}) = 1\}$.

At the start of the search $X_{\texttt{Main}}$ is propagated to 1, and no more filtering can be done. A decision is then computed. There are $n+2$ uninstantiated variables: $X_{\texttt{Studied}}$, $X_{\texttt{Noise}}$, $X_{\texttt{N}_1}, \ldots, X_{\texttt{N}_n}$. RANDOMSEARCH chooses one variable uniformly, so it has a probability $1/(n+2)$ to choose $X_{\texttt{Studied}}$. A value is chosen randomly, so it has a probability $1/2$ of being 1. In this case the decision $X_{\texttt{Studied}} = 1$ is pushed to the search, and the solution will contain the feature `Studied`. Overall, there is at least a probability $\frac{1}{2(n+2)}$ that RANDOMSEARCH will produce the solution containing `Studied`.

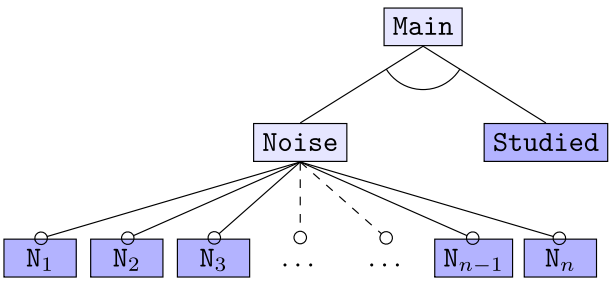


Fig. 2 Example of noisy Feature Model

This toy example shows the advantage of RandomSearch for the task of t-wise coverage. Choosing each variable with the same probability ensures that each variable has a non-negligible chance of being taken.

4.1.2 Generalisation for multiple features

Having understood the behaviour for a single feature, we can now generalise the reasoning to multiple features, hence t-wise combinations. In this section, we first give and prove a looser bound. This proof gives the intuition for the behaviour of RandomSearch on t-wise combinations. We then give a tighter bound and prove it in Appendix 6. This proof is a refinement of the one presented here, but is more technical and does not give more insight into the behaviour.

Proposition 3 *On a feature model with n features, the probability $p_\sigma^{\mathcal{R}}$ of sampling an allowed t-wise combination σ can be lower bounded by*

$$p_\sigma^{\mathcal{R}} \geq \frac{1}{2^t \binom{n}{t}}$$

Proof We restrict our analysis to the first t decisions made during the search. In Theorem 1 we do not make this assumption and prove a better lower bound. To be sure that the sampled solution contains σ, these first decisions must be made on the features of σ. In the first decision there is a $\frac{t}{n}$ chance of choosing one of the variables of σ. On the second decision, there may be only $m \leq n-1$ uninstantiated variables left. There is a chance $\frac{t-1}{m} \geq \frac{t-1}{n-1}$ to choose a second variable of σ at the second decision. Continuing the reasoning, the probability of choosing all the variables of σ during the t first decisions is greater than

$$\frac{t}{n} \cdot \frac{t-1}{n-1} \cdots \frac{1}{n-t+1} = \frac{1}{\binom{n}{t}}.$$

To get exactly the combination σ, the chosen values for each variable F must be the one in the combination, i.e. $\sigma(F)$. For each decision there are two choices with equal probability, hence the factor $\frac{1}{2^t}$. □

We are interested in the case where t is small (less than 7 as noted in [1]), and n is large. In this setting, the proposition can be refined by the following theorem, which gives a lower bound as a convergence result.

Theorem 1 *Given a feature model with n features, and σ an allowed t-wise combination, there exists a sequence u_n^t such that*

$$p_\sigma^{\mathcal{R}} \geq u_n^t$$

and

$$u_n^t \underset{n\to\infty}{\sim} \frac{1}{\binom{n}{t}}.$$

Proof Proved in Appendix A. □

Informally, $p_\sigma^{\mathcal{R}}$ can be approximately lower bounded by $1/\binom{n}{t}$. Compared to the previous proposition, the factor $1/2^t$ has been dropped.

When t is fixed to a small value, this lower bound is polynomial in n, the number of features in the feature model. On the other hand, for uniform sampling, the sampling probability can

Algorithm 1 FREQUENCYDIFF($\mathcal{P}$, $\mathcal{F}$, φ, φ^{obs}).

Require: A CSP $\mathcal{P} = \langle (X_1, \ldots, X_n), \mathcal{D}, \mathcal{C} \rangle$, a list of features (associated to variables) $\mathcal{F} = \{F_1, \ldots, F_n\}$, a mapping φ giving the commonality of every feature, and a mapping φ^{obs} giving the observed frequency of every feature in the previous solutions.

$W \leftarrow$ array of size n (indexed from 1) initialized at 0
for $i = 1$ to n **do**
  if $\mathcal{D}(X_i) = \{0, 1\}$ **then**
    $W[i] \leftarrow \left| \varphi^{obs}_{F_i} - \varphi_{F_i} \right|$
  end if
end for
$varId \leftarrow$ PICKWEIGHTEDRANDOM(W)
if $\varphi^{obs}_{F_{varId}} > \varphi_{F_{varId}}$ **then**
  $chosenValue \leftarrow 0$
else
  $chosenValue \leftarrow 1$
end if
if RANDOM() $> \frac{1+W[varId]}{2}$ **then**
  $chosenValue \leftarrow 1 - chosenValue$
end if
return DECISION($X_{varId} = chosenValue$)

only be lower bounded by $1/2^n$, as seen in the previous example in Figure 2. We recall that Proposition 2 states that the lower the sampling probabilities, the worse the expected t-wise coverage. The polynomial lower bound on the sampling probability using RANDOMSEARCH is an argument for the fact that it is a sampling method that generates test suites with high t-wise. We experimentally prove this fact in Section 5.

4.2 Search strategy

The previous section showed that RANDOMSEARCH is a good starting point for a search strategy to generate a good coverage test suite. However it lacks insight into the previously found solutions. It only can only avoid returning a solution that has already been found.

4.2.1 Presentation of the algorithm

We now present the search strategy we have designed to generate a high t-wise coverage test suite. It is an improvement on RANDOMSEARCH which uses knowledge of the solutions already returned. We also use the commonalities to guide the search. We call this new search strategy FREQUENCYDIFF because it uses the difference between the observed frequency of features, and the commonalities. In this section we assume that we have access to the commonalities of all features F in φ_F. In practice, we use an approximation of the commonalities. The experiments in Section 5 show that an approximation of the commonalities is sufficient to outperform other approaches.

Our approach is a search strategy, i.e. the choice of an uninstantiated variable and a value in its domain to branch on during the solving process. This search strategy guides the search towards an interesting solution. When a solution is found, the search is restarted. It is presented in Algorithm 1. The search strategy has access to the model $\mathcal{P}$ with the variables and domains, the features $\mathcal{F}$ (such that X_i is the variable associated with the feature F_i), the commonalities for each feature φ, and the observed frequency of each feature φ^{obs}. Given a set of previously

found solutions $\mathcal{S}$, the observed frequency of a feature F is $\varphi_F^{obs} = \frac{|\{C \in \mathcal{S} | F \in C\}|}{|\mathcal{S}|}$. These observed frequencies can be updated in time $\mathcal{O}(|\mathcal{F}|)$ when a solution is found.

The strategy first computes the absolute difference between the observed and theoretical frequencies and stores it in an array of weights W. These absolute differences are quantification of how some features are underrepresented (or over-represented) by the current set of solutions. The goal of FREQUENCYDIFF is to correct this under-representation (or over-representation) by increasing the random weights of such features.

The next step of the algorithm, line 7, is to choose the decision variable. An index is chosen according to the weights in W (i.e. $\mathbb{P}(varId = i) = W[i] / \sum_j W[j]$) [33]. This weighted choice will favour those features that have an observed frequency far from their commonality.Note that this also applies to the absence of features.

Example 3 Using the example feature model in Figure 1a, suppose that the first configuration returned is configuration 5, which contains the `Shooter` feature. The `Shooter` feature has a commonality of $\frac{1}{3}$ (because it only appears in configurations 5 and 6), but it has an observed frequency of 1, so its weight is be $\frac{2}{3}$. This weight is high, which increases the chance of returning configurations *not* containing `Shooter`.

The next and final step is to choose the value associated with the variable in the decision. This choice is made in two steps. In a first step, the chosen value is fixed depending on the comparison between the observed frequency and the commonality. If the observed frequency is higher than the commonality, we choose the value 0 to exclude the feature from the constructed configuration. Otherwise, value of 1 is chosen to include the feature in the configuration. Then in a second step (line 13), a random swap is performed. The probability that the value is swapped is proportional to the difference between the observed frequency and the commonality (stored in the array W). If this difference is close to 1, i.e. there is a large underrepresentation (or overrepresentation), then it is unlikely that the chosen value will be swapped.

To summarise the search strategy: weights are computed with the difference between the observed frequency and the commonality, a variable is drawn according to these weights, the value is chosen to bring the observed frequency closer to the commonality, and this value is swapped with a small probability.

Randomisation of the value

We have chosen to randomise the choice of the value (line 13) to prevent the search from getting stuck in an unsatisfiable subspace. It is possible that some features are always present or absent (called *core* or *dead* features) due to some constraints. It is not trivial to check whether a feature is a core or dead feature (unless it is trivially a *core* feature from the feature diagram, for example a mandatory child of the root node). We have chosen not to perform this check to avoid a pre-processing step.

If it had not been randomised, our search strategy might select such a feature as the decision variable. Then the chosen value would be 0 in the case of a core feature in line 9 or 1 in the case of a dead feature in line 11. When the solver makes this decision, it enters an unsatisfiable sub-space. Many computations and backtracks may be necessary to leave this sub-space.

To avoid failing in that case, we used use two techniques. First, we added randomisation of the value. This way there is a small chance of not entering the unsatisfiable sub-space. If the search did enter in such a sub-space, we used restarts monitoring the number of fails during the search. If there are too many fails (i.e. backtracks), the search is restarted from

scratch. Randomisation allows the solver to avoid making the same mistakes over and over again when restarting.

5 Experimental results

This section describes our experiments.First the methodology is presented in Section 5.1 (implementation details, benchmark and state-of-the-art approaches tested). Section 5.2 contains the comparison to the other approaches in terms of t-wise coverage and running time. Finally, Section 5.3 confirms the fact that the approaches behave the same way for higher values of t.

5.1 Methodology

5.1.1 Implementation

Our implementation is available online [1]. It is implemented in `Java`, using the CP solver `choco-solver` version 4.10.10 [34]. For the commonalities φ we used the linear time approximation presented in [13]. FrequencyDiff and RandomSearch are implemented using the search strategies of `choco-solver`. After each solution the search is restarted and the solution is excluded. A restart strategy is used when there are too many fails. The number of fails to restart follows a Luby sequence [35] of factor 50.

We compared these strategies to three state-of-the-art approaches:

- Baital[2] with 5 and 10 rounds. We use strategy 4 presented in [7] (numbered strategy 5 in the implementation) as it is among the best strategies in terms of coverage, and also among the fastest as it does not need to compute the set of combinations.
- Uniform sampling. We use Baital with 1 round for convenience, which is equivalent to using the uniform sampler WAPS [21].
- TableSampling [32] using the implementation in `choco-solver`. TableSampling takes 3 parameters as input: κ the pivot values for the number of solutions enumerated at each step, v the number of variables in the table, and p the probability of keeping a tuple in the table. The authors recommend using $\kappa = 1/p$, and values of v depending on the allowed running time and the desired randomness randomness. We use the parameter sets $(\kappa, v, p) \in \{(4, 4, 1/4), (8, 6, 1/8), (16, 8, 1/16)\}$.

The experiments were run on single threads on a Xeon E7-8870 v4 / 20c / 1.4GH processor. For each instance, 100 solutions are generated. These 100 solutions are generated twice with different random seeds, and the results of running time or size of the coverage are averaged (using the arithmetic mean) over these two runs.

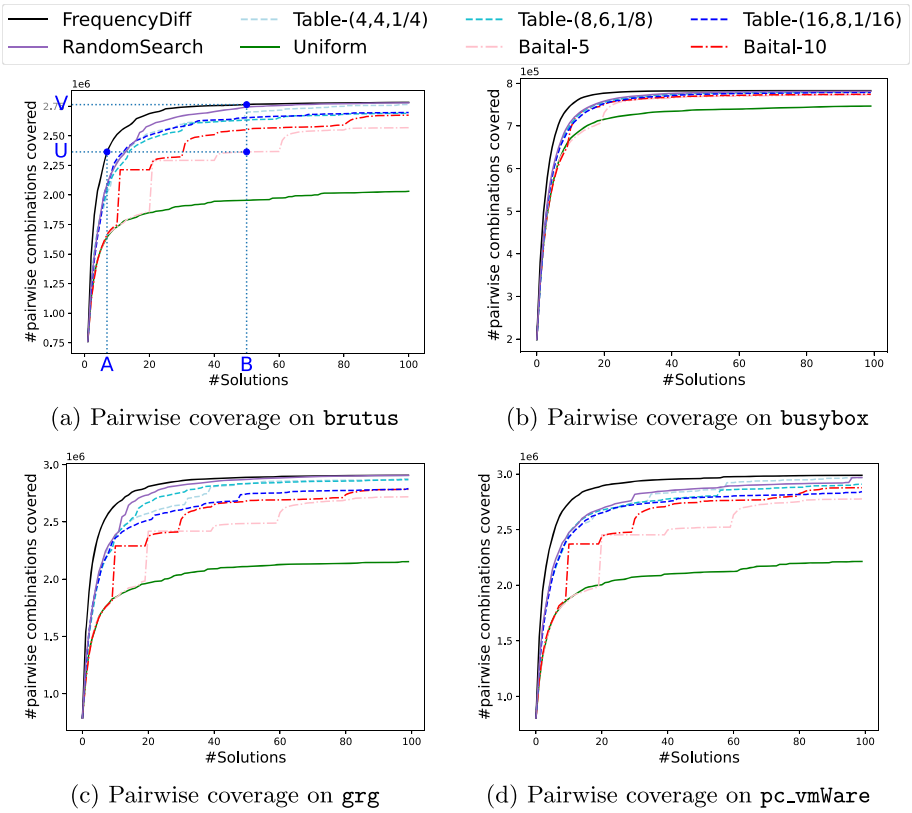
5.1.2 Benchmark

The instances used to test our approach come from the `uvl-models`[3] repository, in the UVL input format [36]. It contains feature models from various domains (e.g. automotive, operating systems).

[1] https://github.com/MathieuVavrille/frequency-diff

[2] https://github.com/meelgroup/baital

[3] https://github.com/Universal-Variability-Language/uvl-models



(a) Pairwise coverage on `brutus`

(b) Pairwise coverage on `busybox`

(c) Pairwise coverage on `grg`

(d) Pairwise coverage on `pc_vmWare`

Fig. 3 Evolution of the pairwise coverage on different instances

BAITAL takes a CNF formula as input, we used FeatureIDE [4] to convert the UVL format to dimacs format. Instances where the conversion did not terminate (mostly due to the size of the instance) or raised an error (mostly due to bad feature naming that could not be easily fixed) were excluded from the benchmark. On most instances (the large ones), the conversion from UVL to dimacs format took about 10s, which we do not include in the total running time of BAITAL and uniform sampling. On two instances, BAITAL did not generate configuration containing all the features due to an issue in the compilation of the d-DNNF representation. These two instances were also removed from the benchmark.

In the end, we applied the approaches to 123 instances. A large part of the instances (116 instances) come from the same initial benchmark [14]. These instances have between 1178 and 1408 features and between 816 and 956 cross-tree constraints.

5.2 Comparison with other approaches

In this section we present the experimental results of the experiments in terms of coverage and running time. We focus on pairwise coverage as we can compute it exactly. The improvement ratios for $t > 2$ are shown in Section 5.3. All the aggregated ratios (of coverage or running time) are summarized at the end of this section in Table 1.

[4] https://github.com/FeatureIDE/FeatureIDE

Table 1 Summary of the pairwise coverage/running time average ratios between other approaches and FREQUENCYDIFF

	RANDOMSEARCH	Uniform Sampling	BAITAL		TABLESAMPLING $-(\kappa, v, p)$		
			5	10	$(4, 4, \frac{1}{4})$	$(8, 6, \frac{1}{8})$	$(16, 8, \frac{1}{16})$
Coverage-100	1.00	1.34	1.07	1.04	1.01	1.02	1.04
Coverage-50	1.01	1.37	1.16	1.07	1.03	1.04	1.06
Size-100	1.39	23.89	7.44	5.10	2.37	3.65	5.21
Size-50	1.50	14.21	6.98	4.12	2.25	2.88	3.58
Time Speedup	1.18	12.27	60.35	118.29	41.14	82.83	961.99

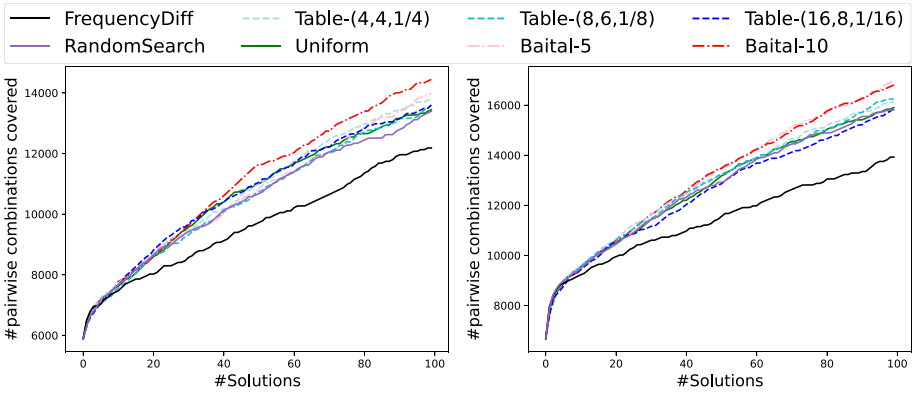
5.2.1 Coverage

Figure 3 shows plots of the evolution of the number of pairwise combinations found by the different approaches. The behaviour is roughly the same for all instances, except for two specific instances discussed in Section 5.2.2. Apart from these two instances, our approach gives the best coverage on all but five instances (and RANDOMSEARCH often gives the second best coverage). On these five instances, FREQUENCYDIFF is the second best approach, just behind RANDOMSEARCH. This means that the search strategies (RANDOMSEARCH and FREQUENCYDIFF) outperform the other approaches on 121 instances (all of the benchmark except two particular instances). FREQUENCYDIFF alone itself outperforms *all* the other approaches on 116 out of the 123 instances of the benchmark.

A first way to aggregate the results of all instances into a single value is to look at the coverage after a fixed number of solutions. Figure 3a shows the values behind the following description. In this figure, after 50 solutions (label B), BAITAL (with 5 rounds) covered 2,363,836 combinations (label U), and FREQUENCYDIFF covered 2,763,767 combinations (label V). The ratio of improvement is $V/U = 1.17$, which means that after 50 solutions FREQUENCYDIFF covered 17% more combinations. We call this factor the *coverage improvement*. To aggregate the result we use the geometric mean of these factors.

We can also look at the other approaches in detail.

- Uniform sampling is the worst approach in terms of coverage. It fails to find new combinations after the first few solutions, and after 100 solutions the coverage is much lower than all other approaches. On average, FREQUENCYDIFF finds 34% more combinations (i.e. the improvement in coverage is 1.34).
- Jumps in coverage can be seen in the plots for BAITAL. These are due to the weights updates between rounds. Before the first update, the curve follows the uniform sampling curve because all the weights are equal. At the end of the rounds, the weights are recomputed according to the solutions found, so the sampling is weighted towards features (and combinations) that have not yet been found, resulting in the jump in coverage. On average, FREQUENCYDIFF finds 4% more combinations than BAITAL-10.
- TABLESAMPLING performs better than overall than BAITAL. It seems that the lower the number of variables in the tables, the better the coverage. This number of variables can bring the sampling closer or further from uniformity. When v is high, the sampling is more uniform, so the coverage is closer to that of uniform sampling, which is the worst of all the approaches. If v is lower, TABLESAMPLING is closer to RANDOMSEARCH, so the coverage is better.



(a) Instance `dm_ASEJ1` (b) Instance `dm_DissModel`

Fig. 4 Evolution of the pairwise coverage on two particular instances where our approach did not perform well

5.2.2 Analysis of two particular instances

Of all the test suites generated, there were only two instances where FREQUENCYDIFF did not outperform the other approaches. These two instances are described in detail in this section

Figure 4 shows the evolution of the pairwise coverage on these two instances. These plots are very different from the previous pairwise coverage plots in Figures 3 and 7. Instead of having a logarithmic growth (it becomes harder to find new combinations after some solutions), the two instances have a linear growth of the coverage. This means that each solution does not cover many new combinations.

To understand this behaviour, we show the feature model associated with instance `dm_ASEJ1` in Figure 5. The other instance (`dm_DissModel`) is similar. This instance consists of 4 main features in an `or` group. The feature `Modes` has an `or` group with 4 children. The special feature in this feature model is `Strands`. It has 100 children in an `alternative` group (only one of the children can be taken). It also contains a cross-tree constraint (the instance `dm_DissModel` does not have such a constraint).

The optimal strategy for optimising the t-wise coverage on this instance is to choose a different $\texttt{S}_i$ on each solution. The reason why FREQUENCYDIFF does not perform well is because of the very low commonality (i.e. frequency) of the features $\texttt{S}_i$ (around $1/100$)

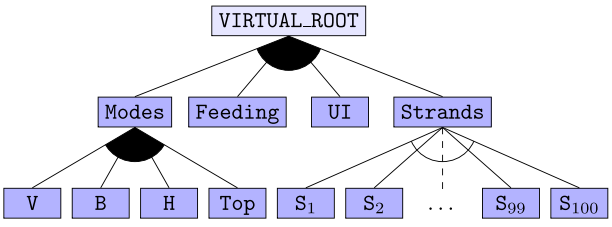


Fig. 5 Feature Model of the instance `dm_ASEJ1`. The feature's names have been simplified for clarity

combined with the alternative group they are in. This increases the chances of selecting the same feature more than once.

Example 4 We show how the search can behave. We assume that we have already sampled 20 solutions with FREQUENCYDIFF, all containing a different $\texttt{S}_i$. We want to estimate the probability of picking an already chosen $\texttt{S}_i$. The observed frequency of the already chosen $\texttt{S}_i$ is $1/20$. For the already chosen $\texttt{S}_i$, the weights are $1/20 - 1/100 \approx 1/20$, and for the not yet chosen $\texttt{S}_i$ they are only $1/100$. Then the probability of choosing an already taken $\texttt{S}_i$ is more than $1/2$ (it is $\frac{20 \cdot \frac{1}{20}}{20 \cdot \frac{1}{20} + 80 \cdot \frac{1}{100}} = \frac{1}{1.8}$). By default the value chosen would be 0, as the observed frequency is higher than the commonality, but there is a $\frac{1+\frac{1}{20}}{2}$ chance of swapping this value, thus adding the feature to the solution.

We argue that modelling this instance as a feature model was not the correct way to represent it. Therefore, the use of feature model based approaches is not appropriate. The `Strands` feature with 100 children S_i should have been modelled as an integer variable (for example in a CP framework) as $\texttt{Strands} \in \{1, \ldots, 100\}$. Such modelling of integer variables (called attributes) in feature models have been studied in [17] and allows the use of CP's global constraints. RANDOMSEARCH would work as is in a CP framework (where the values are not necessarily binary). If necessary, it would also be possible to completely prevent already chosen $\texttt{S}_i$ from being taken. The fact that FREQUENCYDIFF does not perform well on these two instances is not representative of the performance benchmark as a whole.

5.2.3 Size of the solution set

The coverage improvements may seem small, but as the number of combinations found increases, it becomes harder and harder to find new combinations. Therefore, even a small percentage of improvement is hard to achieve when the coverage is already high. Due to the inverse exponential behaviour of the coverage, the number of solutions needed to achieve the same coverage varies greatly for different approaches.

To aggregate the results, we look at how many solutions are required by the different approaches to give the same coverage. In Figure 3a, after 50 solutions (label B), BAITAL covers 2,363,836 combinations (label U). FREQUENCYDIFF covers the same number of combinations after only 7 solutions (label A). This means that FREQUENCYDIFF can give the same coverage with $B/A \approx 7$ times fewer solutions than BAITAL. We call this ratio the *size improvement*. Again, we aggregate the results using the geometric mean.

On average, FREQUENCYDIFF requires 5 times fewer solutions than BAITAL-10 to achieve the same coverage. This means that on average, the coverage of FREQUENCYDIFF will be the same after 20 solutions as that the one of BAITAL-10 after 100 solutions.

The coverage of TABLESAMPLING is better than the one of BAITAL, and this reflects on the number of solutions required to obtain a given coverage. To get the same coverage as TABLESAMPLING after 100 solutions, FREQUENCYDIFF requires 2.4 times fewer solutions.

To achieve the same coverage as RANDOMSEARCH (the second best approach), FREQUENCYDIFF requires 1.4 times fewer solutions.

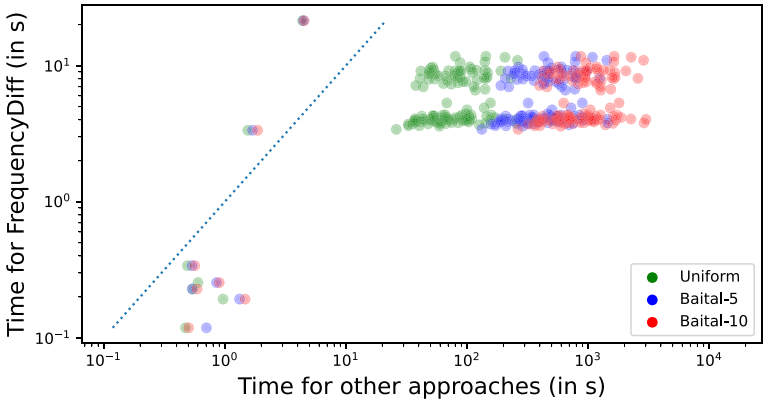
Recall that the goal of generating solutions is to use them as tests of a product line. The actual implementation of the solution in the product line could be expensive (in the case of an automotive product line) or time-consuming (in the case of software compilation). Any reduction in the number of solutions generated, without affecting the coverage, is directly reflected

faster or lower cost testing of the product line. Our strategy FREQUENCYDIFF significantly improved this size of generated solutions.

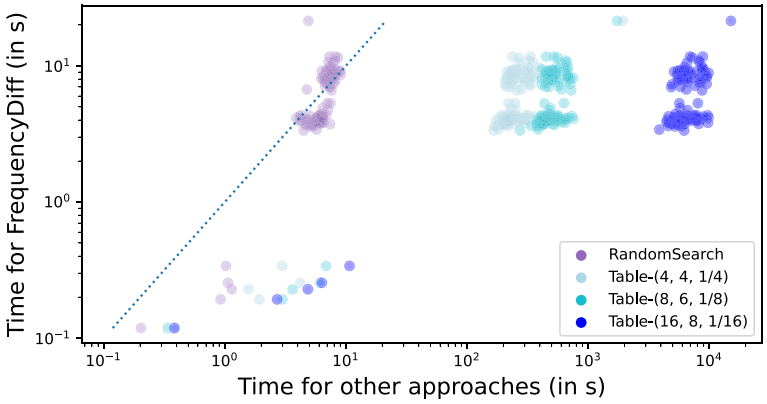
5.2.4 Running time

A scatter plot of the running time of the other approaches compared to `frequency_diff` is shown in Figure 6. For clarity, the plot is divided into a comparison with SAT sampling approaches (uniform sampling and BAITAL) in Figure 6a and CP sampling approaches (RANDOMSEARCH and TABLESAMPLING) in Figure 6b. In the two scatter plots, the y axis is the time taken by FREQUENCYDIFF. If an dot (an instance) is below the dotted line, it means that FREQUENCYDIFF was faster.

FREQUENCYDIFF is the fastest of all approaches overall. RANDOMSEARCH can be considered as fast as FREQUENCYDIFF. All other approaches are orders of magnitude slower than the search strategy approaches. In only two cases was FREQUENCYDIFF slower than uniform sampling and BAITAL.



(a) Compared to BAITAL and uniform sampling



(b) Compared to TABLESAMPLING and RANDOMSEARCH

Fig. 6 Scatter plot of running times. Each dot is an instance

In the following, we compute the speedup of our strategy over other approaches, i.e. how much faster our strategy could generate the requested 100 solutions. Let τ^I_{freq} be the time taken by our approach to sample 100 solutions on the instance I, and $\tau^I_{\mathcal{U}}$ be the time taken by uniform sampling. The speedup between our approach over uniform sampling is then $\frac{\tau^I_{freq}}{\tau^I_{\mathcal{U}}}$ on instance I. We use the geometric mean to average the speedups over all instances in $\mathcal{I}$:

$$speedup = \left(\prod_{I \in \mathcal{I}} \frac{\tau^I_{freq}}{\tau^I_{\mathcal{U}}} \right)^{1/|\mathcal{I}|}$$

FREQUENCYDIFF has a speedup of 12.27 over uniform sampling. Most of the time of BAITAL is spent when computing the weights of the d-DNNF representation, which is done at the beginning of each round (and only once in the case of uniform sampling). This is reflected in the running time, as BAITAL-5 is about 5 times slower than uniform sampling, and BAITAL-10 is 10 times slower.

FREQUENCYDIFF has a speedup of 41 compared to TABLESAMPLING with parameters $(4, 4, \frac{1}{4})$. With parameters $(16, 8, \frac{1}{16})$, TABLESAMPLING is the slowest. The running time increases significantly when the number of variables increases, because more tuples have to be generated during table creation. As remarked in the previous section, the coverage is better when v is smaller. We can conclude that there is no reason (on the t-wise coverage problem) to use high values of v. We would also like to mention that our results are opposite to the results of the benchmark used to evaluate TABLESAMPLING. In fact, the problem we consider is completely different: in their experiments they show that TABLESAMPLING is more uniform than RANDOMSEARCH. They also show that TABLESAMPLING is able to sample solutions faster than RANDOMSEARCH. This is because the benchmark used by the authors contain hard CP instances, and the use of a dedicated search strategy (which is not overridden by TABLESAMPLING) greatly improves the running time. On the other hand, on feature models it is much easier to find a solution, and there is disadvantage to using RANDOMSEARCH as the search strategy.

5.2.5 Summary of the experiments

We summarise the pairwise coverage and running time results given in the previous sections.

Table 1 gives all the ratios (compared to FREQUENCYDIFF) that were partly mentioned in the previous sections. When a coverage ratio was mentioned in the previous sections, it referred to the coverage after 100 solutions. It is also possible to give this ratio after only 50 solutions, and these ratios are present in the table with the label 50. These ratios do not differ much between the ones computed after 100 solutions. The running time is only recorded at the end of the 100 solution generation.

It should be noted that all the ratios compared to FREQUENCYDIFF in this table are greater than 1. This means that overall on the whole benchmark, in terms of coverage or running time, FREQUENCYDIFF is the best strategy compared to the other approaches tested.

5.3 Higher value of *t*

In the previous section we plotted the evolution of the pairwise coverage. Ideally we would like to evaluate the t-wise coverage for higher values of t, but it is too expensive to compute exactly. Most instances have more than a thousand features, so there are more than a billion

3-wise combinations covered by each solution. However, we can compute the exact number of 3-wise (and even 4-wise) combinations on a smaller instance, or approximate this value using ApproxCov [37] for larger instances.

5.3.1 Exact computation

To validate that our approach works well for higher values of t, we compute the exact number of covered combinations on the small instance `berkeleydb` (with 76 features). In Figure 7 we plot the evolution of the t-wise coverage for $t \in \{1, 2, 3, 4\}$ on the instance `berkeleydb`. We can see that the behaviour of the approaches remains the same for all values of t.

We can see that Baital performs better for $t = 1$. This is a consequence of the litteral-weight function used as the distribution weight. This function assigns weights to the features, and samples according to these weights. So we see that after one update of the weights (at solution 10 for Baital-10 and solution 20 for Baital-5) all feature-wise combinations are found.

5.3.2 Approximate computation

In the instance `berkeleydb`, the exact number of combinations can be computed for higher values of t because there are few features. However, this is impossible for larger

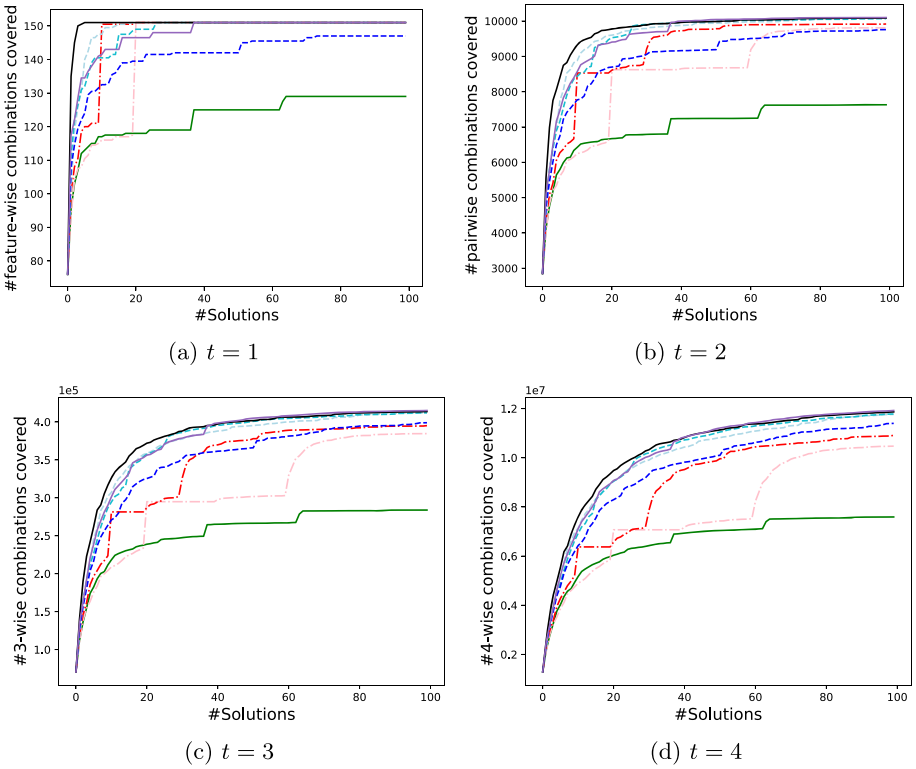


Fig. 7 Evolution of the t-wise coverage on the `berkeleydb` instance, with different values of t

instances. ApproxCov [37] is an algorithm for computing an approximation of the number of combinations covered by a test suite. Here we use this tool to get an approximation of the behaviour of our approach compared to the other approaches. We use ApproxCov with default parameters, i.e. with a probability of 0.95, the result is within a factor of 1.05 of the exact value.

In Figure 8 we plot the average improvement of the coverage size over all instances, for different numbers of solutions (by restricting the generated test suite to the first n configurations, for $n \in \{10, 20, \ldots, 90, 100\}$). We can see that for all values of $t \in \{2, 3, 4, 5\}$, FrequencyDiff generates test suites that cover more combinations than the other approaches (all the coverage improvements are greater than 1). This confirms the fact that our approach can generate good coverage test suites for higher values of t, even for large instances.

In the previous section we only showed the pairwise coverage. As noted in [24], a test suite with good t-wise coverage will most likely also have good $t+1$-wise coverage. We have proved this experimentally. Thus, the pairwise coverage results we showed in the previous section extend to higher values of t.

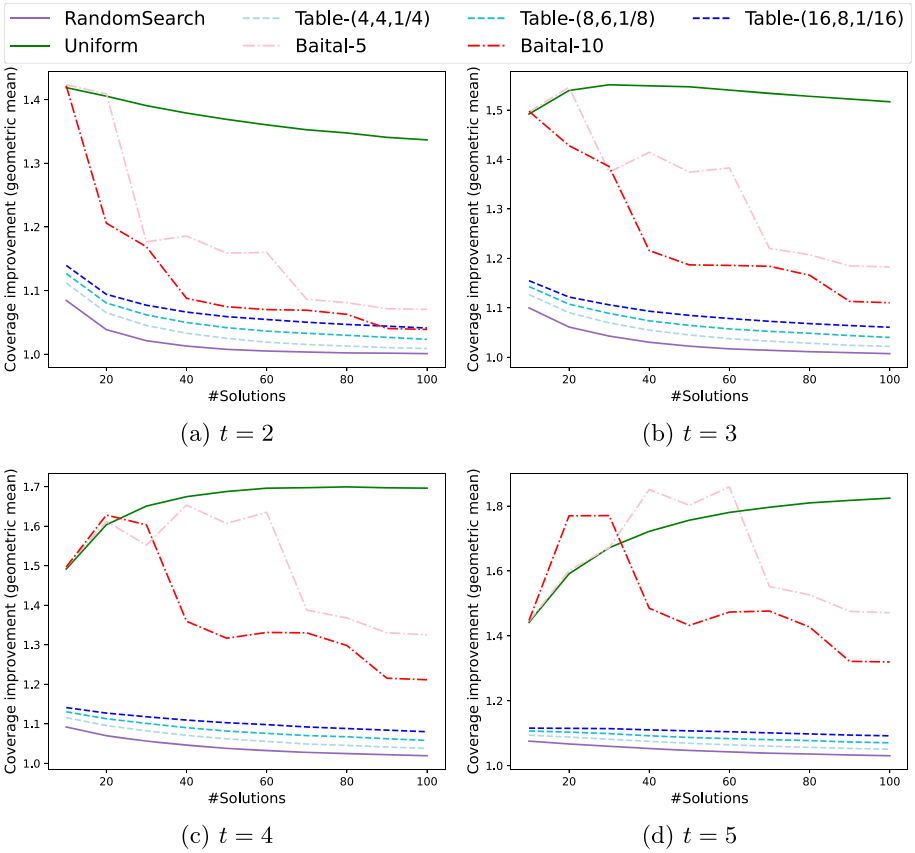


Fig. 8 Evolution of the improvement of FrequencyDiff in t-wise (approximate) coverage size, with different values of t. For a given number of solutions, we plot the (geometric) average of the improvement in coverage size compared to FrequencyDiff. For example, for $t = 3$, after 60 solutions, FrequencyDiff covers on average 1.2 times more combinations than Baital-10

6 Conclusion

In this article we showed that CP's search strategies are an excellent way of generating test suites with high t-wise coverage. We explained this result by analysing RandomSearch's probability of returning a solution containing a given combination. This analysis showed that RandomSearch is more suited at generating t-wise covering test suites than uniform sampling.

We proposed an improvement to RandomSearch called FrequencyDiff, which uses information about the previously generated solutions. Using this information, and comparing it with the commonality of the features, it tweaks the distribution used in RandomSearch to favour interesting features.

We experimentally tested these search strategies on feature models with more than a thousand features. The results showed that the search strategies outperformed other sampling approaches by significantly improving the coverage, and reducing the number of solutions required to achieve a given coverage. Moreover, the search strategies are faster than the other sampling approaches tested. FrequencyDiff also improves the coverage over RandomSearch, making it the best of the approaches we tested, with no running time overhead.

More importantly, this work is also a new step towards the use of CP solvers in feature modelling. In general, SAT solvers are faster on problems that are naturally expressed with clauses, but CP solvers can deal with more expressive languages. In particular, feature models with cardinality constraints (given l and u, the number of selected children of the node should be between l and u) are naturally expressed in CP without the need for translation (to CNF for SAT). This is a very interesting and challenging research direction, as it would also allow to analyse feature models with integer variables, redefine commonalities and use CP samplers.

Appendix

A Proof of the lower bound of RandomSearch's probability

This appendix proves Theorem 1. The proof is in two steps. First, we need to define some mathematical objects and show some properties on them. We then relate these objects to the probability of RandomSearch to sample a given combination.

First, we introduce a family of polynomials. This family of polynomials appears in the lower bound that we prove.

Definition 9 We note P^t the only polynomial of degree $t-1$ such that $P^t(i) = t! \cdot 2^i$ for i in $\{0, \ldots, t-1\}$. By convention we extend the definition to $P^0(n) = 0$.

There is a unique polynomial of degree d which passes through $d+1$ points, so P^t is well defined. We now prove a recurrence formula for these polynomials.

Lemma 1 *The polynomials P^t follow the recurrence relation*

$$P^t(n) = P^t(n-1) + t \cdot P^{t-1}(n-1)\,.$$

Proof Let $P = \frac{P^{t+1}(n+1) - P^{t+1}(n)}{t+1}$. We want to prove that $P = P^t$, and to do so we have to show that it has degree t and that for $i \in \{0, \ldots, t-1\}$, $P(i) = P^t = t! \cdot 2^i$.

- The monomial of degree t cancels in the subtraction $P^{t+1}(n+1) - P^{t+1}(n)$, so P only has degree $t-1$

- Let $i \in \{0, \dots t-1\}$,

$$\begin{aligned} P(i) &= \frac{P^{t+1}(i+1) - P^{t+1}(i)}{t+1} \\ &= \frac{(t+1)! \cdot 2^{i+1} - (t+1)! \cdot 2^i}{t+1} \\ &= t! \cdot 2^i \end{aligned}$$

So $P = P^t$. The property follows from a rearrangement of the equality. □

With these polynomials we define a (two-dimensional) sequence. This sequence is the lower bound of $p_\sigma^{\mathcal{R}}$ used in the theorem we want to prove.

Definition 10 We define the two-dimensional sequence u on $0 \leq t \leq n$ as

$$u_n^t = \frac{(n-t)! \cdot \left(t! \cdot 2^n - P^t(n)\right)}{n! \cdot 2^n}.$$

This sequence is used in the theorem we want to prove as a lower bound. We first show a convergence property of u.

Lemma 2 *Let t be an integer, then u_n^t is equivalent to $1/\binom{n}{t}$ as n tends to infinity, i.e.*

$$u_n^t \underset{n\to\infty}{\sim} 1/\binom{n}{t}.$$

Proof

$$\begin{aligned} u_n^t \cdot \binom{n}{t} &= \frac{(n-t)! \cdot \left(t! \cdot 2^n - P^t(n)\right)}{n! \cdot 2^n} \cdot \binom{n}{t} \\ &= \frac{\left(t! \cdot 2^n - P^t(n)\right)}{t! \cdot 2^n} \\ &= 1 - \frac{P^t(n)}{t! \cdot 2^n} \\ &\xrightarrow[n\to\infty]{} 1 \end{aligned}$$

□

To link this sequence to the probability under consideration, we use the following lemma which gives a recurrence relation for u_n^t.

Lemma 3 *For n and t integers such that $1 \leq t \leq n$, the sequence u follows the recurrence relation*

$$u_n^t = \frac{1}{2n}\left(t \cdot u_{n-1}^{t-1} + (n-t) \cdot u_{n-1}^t\right)$$

Proof We simply replace the definition of u on the right hand side of the equality.

$$\begin{aligned}
\frac{1}{2n}\left(t \cdot u_{n-1}^{t-1} + (n-t) \cdot u_{n-1}^{t}\right) &= \frac{1}{2n}\left(t \cdot \frac{(n-t)! \cdot \left((t-1)! \cdot 2^{n-1} - P^{t-1}(n-1)\right)}{(n-1)! \cdot 2^{n-1}}\right. \\
&\qquad \left. +(n-t) \cdot \frac{(n-t-1)! \cdot \left(t! \cdot 2^{n-1} - P^{t}(n-1)\right)}{(n-1)! \cdot 2^{n-1}}\right) \\
&= \frac{(n-t)!}{n! \cdot 2^{n}}\left(t! \cdot 2^{n} - t \cdot P^{t-1}(n-1) - P^{t}(n-1)\right) \\
&= \frac{(n-t)! \cdot \left(t! \cdot 2^{n} - P^{t}(n)\right)}{n! \cdot 2^{n}} \\
&= u_{n}^{t}
\end{aligned}$$

□

To prove the main theorem, we prove the following stronger version.

Lemma 4 *Let n be the number of features and σ be a t-wise combination appearing in on the set of solutions, then $p_\sigma \geq u_n^t$.*

Proof We note l_n^t the lower bound, and show that it verifies the same recurrence relation as u_n^t. We now consider a fixed t-wise combination σ, on a problem with n variables. We know that there is at least one configuration C containing σ. We recall that there is one boolean variable X_f associated with each feature in $\mathcal{F}$. When all variables are instantiated, it defines a configuration such that $f \in C \Leftrightarrow X_f = 1$.

To obtain the recurrence relation, we apply the RANDOMSEARCH search strategy. Suppose that there are m uninstantiated variables. Then

- with probability $\frac{t}{m}$ one variable of σ is picked by RANDOMSEARCH (let's note it X_f). Then there is a $\frac{1}{2}$ probability that the correct value is chosen for X_f (i.e. $\sigma(X_f)$). Then the other $t-1$ variables must also be fixed, among the remaining $m-1$ uninstantiated variables. In the best case, some values are propagated, reducing the number of uninstantiated variables. In the worst case, there is no propagation. In this case, the probability of choosing the correct values for the $t-1$ remaining variables from the $m-1$ variables is l_{m-1}^{t-1}. Overall, this has a probability of

$$\frac{t}{2m} l_{m-1}^{t-1}$$

  ;
- otherwise (so with probability $\frac{m-t}{m} = 1 - \frac{t}{m}$), a variable X_f is chosen that is not in σ. Then there is a chance that the value chosen for X_f does no longer allow the combination σ. However, we know that there is (at least) one solution that contains σ, so if X_f takes the same value as in that solution, there is still a chance of returning a solution with σ. Since there are two possible values for X_f, there is a $\frac{1}{2}$ chance that it takes the value in C (i.e. $X_f = 1 \Leftrightarrow f \in C$). Then the t variables in σ still have to be chosen from the $m-1$ remaining variables, hence a probability of l_{m-1}^t.

Combining these probabilities, we have

$$p_\sigma \geq l_n^t = \frac{1}{2n}\left(t \cdot l_{n-1}^{t-1} + (n-t) \cdot l_{n-1}^{t}\right).$$

l_n^t follows the same recurrence relation as u_n^t and has the same initial values ($l_n^0 = 1$ for $n \geq 0$, and $l_t^t = 1/2^t$ for $t \geq 0$), so it is equal to u_n^t. □

We can now prove the main theorem simply by using the previous lemmas.

Theorem 1 Given a feature model with n features, and σ an allowed t-wise combination, there exists a sequence u_n^t such that

$$p_\sigma^{\mathcal{R}} \geq u_n^t$$

and

$$u_n^t \underset{n\to\infty}{\sim} \frac{1}{\binom{n}{t}}.$$

Proof We simply use Lemmas 2 and 4. □

Acknowledgements I want to thank Charlotte Truchet and Charles Prud'homme for their help proofreading the article, and their valuable comments during our discussions.

Funding The author has no relevant financial or non-financial interests to disclose.

Declarations

Conflicts of interest The author did not receive support from any organization for the submitted work.

References

1. Kuhn, D., Kacker, R., & Lei, Y. (2010) Practical Combinatorial Testing. *Special publication (NIST SP), national institute of standards and technology*, Gaithersburg, MD. https://doi.org/10.6028/NIST.SP.800-142
2. Melo, J., Flesborg, E., Brabrand, C., & Wasowski, A. (2016). A quantitative analysis of variability warnings in linux. In I. Schaefer, V. Alves, & E. S. de Almeida (Eds.), *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems* (pp. 3–8). Salvador, Brazil: ACM,???? https://doi.org/10.1145/2866614.2866615
3. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., & Baudry, B. (2019). Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empir. Softw. Eng., 24*(2), 674–717. https://doi.org/10.1007/s10664-018-9635-4
4. Cmyrev, A., & Reissing, R. (2014). Efficient and effective testing of automotive software product lines. *Applied Science and Engineering Progress 7*(2), 53–57. https://doi.org/10.14416/j.ijast.2014.05.001
5. Kuhn, D.R., Kacker, R.N., & Lei, Y. (2013). *Introduction to Combinatorial Testing*. CRC press, ???
6. Oh, J., Gazzillo, P., Batory, D., Heule, M., & Myers, M. (2019). *Uniform sampling from kconfig feature models*. The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19 2
7. Baranov, E., Legay, A., & Meel, K. S. (2020). Baital: an adaptive weighted sampling approach for improved t-wise coverage. In P. Devanbu, M. B. Cohen, & T. Zimmermann (Eds.), *ESEC/FSE '20: 28th ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 1114–1126). Virtual Event, USA: ACM,???? https://doi.org/10.1145/3368089.3409744
8. Rossi, F., van Beek, P., & Walsh, T. (eds.) (2006). *Handbook of Constraint Programming. Foundations of Artificial Intelligence*, (vol. 2). Elsevier, ???. https://www.sciencedirect.com/science/bookseries/15746526/2
9. Li, H., Yin, M., & Li, Z. (2021). Failure based variable ordering heuristics for solving csps (short paper). In: L.D. Michel (ed.) *27th International conference on principles and practice of constraint programming, CP 2021*, Montpellier, France (Virtual Conference), October 25-29, 2021. LIPIcs, vol. 210, pp. 9–1910. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, ???. https://doi.org/10.4230/LIPIcs.CP.2021.9
10. Wattez, H., Lecoutre, C., Paparrizou, A., & Tabary, S. (2019). Refining constraint weighting. In: *31st IEEE international conference on tools with artificial intelligence, ICTAI 2019*, (pp. 71–77). Portland, OR, USA:IEEE, ???. November 4-6, 2019.https://doi.org/10.1109/ICTAI.2019.00019
11. Godard, D., Laborie, P., & Nuijten, W. (2005). Randomized large neighborhood search for cumulative scheduling. In S. Biundo, K. L. Myers, & K. Rajan (Eds.), *Proceedings of the fifteenth international*

conference on automated planning and scheduling (ICAPS 2005) (pp. 81–89). Monterey, California, USA: AAAI, ???. http://www.aaai.org/Library/ICAPS/2005/icaps05-009.php
12. Benavides, D., Segura, S., & Ruiz-Cortés, A. (2009). *Automated analysis of feature models: A detailed literature review*. Techn. Ber. ISA-09-TR-04. Seville, Spain: Applied Software Engineering Research Group, University of Seville
13. Vavrille, M., Meunier, E., Truchet, C., & Prud'Homme, C. (2023). *Linear Time Computation of Variation Degree and Commonalities on Feature Diagrams*. Technical Report RR-2023-01-DAPI, Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, LS2N, UMR 6004, F-44000 Nantes, France. https://hal.science/hal-03970237
14. Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., & Schaefer, I. (2017). Is there a mismatch between real-world feature models and product-line research? In E. Bodden, W. Schäfer, A. van Deursen, & A. Zisman (Eds.), *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017* (pp. 291–302). Paderborn, Germany: ACM,????. https://doi.org/10.1145/3106237.3106252
15. Batory, D.S. (2005). Feature models, grammars, and propositional formulas. In: J.H. Obbink, K. Pohl (eds.) *Software product lines, 9th international conference, SPLC 2005, proceedings. lecture notes in computer science*, (vol. 3714, pp. 7–20). Rennes, France, September 26-29, 2005,: Springer, ???. https://doi.org/10.1007/11554844_3
16. Jackson, P.B., & Sheridan, D. (2004) Clause form conversions for boolean circuits. In: H.H. Hoos, D.G. Mitchell (eds.) *Theory and applications of satisfiability testing, 7th international conference, SAT 2004,* Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers. Lecture Notes in Computer Science, (vol. 3542, pp. 183–198). Springer, ???. https://doi.org/10.1007/11527695_15
17. Karatas, A.S., Oguztüzün, H., & Dogru, A.H. (2010). Global constraints on feature models. In: D. Cohen (ed.) *Principles and practice of constraint programming - CP 2010 - 16th international conference, CP 2010*, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings. Lecture Notes in Computer Science, (vol. 6308, pp. 537–551). Springer, ???. https://doi.org/10.1007/978-3-642-15396-9_43
18. Karatas, A. S., Oguztüzün, H., & Dogru, A. H. (2013). From extended feature models to constraint logic programming. *Sci. Comput. Program., 78*(12), 2295–2312. https://doi.org/10.1016/j.scico.2012.06.004
19. Oh, J., Gazzillo, P., & Batory, D.S (2019) *t*-wise coverage by uniform sampling. In: T. Berger, P. Collet, L. Duchien, T. Fogdal, P. Heymans, T. Kehrer, J. Martinez, R. Mazo, L. Montalvillo, C. Salinesi, X. Tërnava, T. Thüm, T. Ziadi (eds.) *Proceedings of the 23rd international systems and software product line conference, SPLC 2019*, Volume A, Paris, France, September 9-13, 2019, (pp. 15–1154). ACM, ???. https://doi.org/10.1145/3336294.3342359
20. Fernández-Amorós, D., Heradio, R., Cerrada, J. A., & Cerrada, C. (2014). A scalable approach to exact model and commonality counting for extended feature models. *IEEE Trans. Software Eng., 40*(9), 895–910. https://doi.org/10.1109/TSE.2014.2331073
21. Gupta, R., Sharma, S., Roy, S., Meel, K.S. (2019) WAPS: weighted and projected sampling. In: T. Vojnar, L. Zhang (eds.) *Tools and algorithms for the construction and analysis of systems - 25th international conference, TACAS 2019, held as part of the european joint conferences on theory and practice of software, ETAPS 2019*, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, (vol. 11427, pp. 59–76). Springer, ???. https://doi.org/10.1007/978-3-030-17462-0_4
22. Golia, P., Soos, M., Chakraborty, S., & Meel, K.S. (2021) Designing samplers is easy: The boon of testers. In: *Formal methods in computer aided design, FMCAD 2021*, New Haven, CT, USA, October 19-22, 2021, (pp. 222–230). IEEE, ???. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_31
23. Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering, 23*(7), 437–444.
24. Johansen, M.F., Haugen, Ø., & Fleurey, F. (2012). An algorithm for generating t-wise covering arrays from large feature models. In: E.S. de Almeida, C. Schwanninger, D. Benavides (eds.) *16th International software product line Conference, SPLC '12*, Salvador, Brazil - September 2-7, 2012, Volume 1, pp. 46–55. ACM, ???. https://doi.org/10.1145/2362536.2362547
25. Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., & Saake, G. (2016). Incling: efficient product-line testing using incremental pairwise sampling. In: B. Fischer, I. Schaefer (eds.) *Proceedings of the 2016 ACM SIGPLAN international conference on generative programming: Concepts and experiences, GPCE 2016*, Amsterdam, The Netherlands, October 31 - November 1, 2016, (pp. 144–155). ACM, ???. https://doi.org/10.1145/2993236.2993253
26. Yamada, A., Biere, A., Artho, C., Kitamura, T., & Choi, E. (2016) Greedy combinatorial test case generation using unsatisfiable cores. In: D. Lo, S. Apel, S. Khurshid (eds.) *Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016*, Singapore, September 3-7, 2016, (pp. 614–624). ACM, ???. https://doi.org/10.1145/2970276.2970335
27. Hervieu, A., Baudry, B., & Gotlieb, A (2011). PACOGEN: automatic generation of pairwise test configurations from feature models. In: T. Dohi, B. Cukic (eds.) *IEEE 22nd international symposium on software*

reliability engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011, (pp. 120–129). IEEE Computer Society, ???. https://doi.org/10.1109/ISSRE.2011.31
28. Hervieu, A., Marijan, D., Gotlieb, A., & Baudry, B. (2016). Practical minimization of pairwise-covering test configurations using constraint programming. *Inf. Softw. Technol., 71*, 129–146. https://doi.org/10.1016/j.infsof.2015.11.007
29. Perez, G., & Régin, J. (2017). Mdds: Sampling and probability constraints. In: J.C. Beck (ed.) *Principles and practice of constraint programming - 23rd international conference, CP 2017*, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, (vol. 10416, pp. 226–242). Springer, ???. https://doi.org/10.1007/978-3-319-66158-2_15
30. Chakraborty, S., Meel, K.S., & Vardi, M.Y. (2014). Balancing scalability and uniformity in SAT witness generator. In: *The 51st annual design automation conference 2014, DAC '14*, San Francisco, CA, USA, June 1-5, 2014, (pp. 60–1606). ACM, ???. https://doi.org/10.1145/2593069.2593097
31. Pesant, G., Quimper, C., & Verhaeghe, H. (2022) Practically uniform solution sampling in constraint programming. In: P. Schaus (ed.) *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022*, Los Angeles, CA, USA, June 20-23, 2022, Proceedings. Lecture Notes in Computer Science, (vol. 13292, pp. 335–344). Springer, ???. https://doi.org/10.1007/978-3-031-08011-1_22
32. Vavrille, M., Truchet, C., & Prud'homme, C. (2022). Solution sampling with random table constraints. *Constraints An Int. J., 27*(4), 381–413. https://doi.org/10.1007/s10601-022-09329-w
33. Walker, A. J. (1977). An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw., 3*(3), 253–256. https://doi.org/10.1145/355744.355749
34. Prud'homme, C., Fages, J.-G., & Lorca, X. (2016). *Choco Solver Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S*. http://www.choco-solver.org
35. Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Inf. Process. Lett., 47*(4), 173–180. https://doi.org/10.1016/0020-0190(93)90029-9
36. Engelhardt, D. (2020) *Towards a universal variability language*. Master's thesis. TU Braunschweig, Germany
37. Baranov, E., Chakraborty, S., Legay, A., Meel, K.S., & Vinodchandran, N.V. (2022). A scalable t-wise coverage estimator. In: *44th IEEE/ACM 44th international conference on software engineering, ICSE 2022*, Pittsburgh, PA, USA, May 25-27, 2022, (pp. 36–47). ACM, ???. https://doi.org/10.1145/3510003.3510218