# Baital: An Adaptive Weighted Sampling Approach for Improved t-wise Coverage

Eduard Baranov
UCLouvain
Belgium
eduard.baranov@uclouvain.be

Axel Legay
UCLouvain
Belgium
axel.legay@uclouvain.be

Kuldeep Meel
NUS
Singapore
meel@comp.nus.edu.sg

## ABSTRACT

The rise of highly configurable complex software and its widespread usage requires design of efficient testing methodology. T-wise coverage is a leading metric to measure the quality of the testing suite and the underlying test generation engine. While uniform sampling based test generation is widely believed to be the state of the art approach to achieve t-wise coverage in presence of constraints on the set of configurations, uniform sampling fails to achieve high t-wise coverage in presence of complex constraints.

In this work, we propose a novel approach Baital, based on adaptive weighted sampling using literal weighted functions, to generate test sets with high t-wise coverage. We demonstrate that our approach leads to significantly high t-wise coverage. The novel usage of literal weighted sampling leaves open several interesting directions, empirical as well as theoretical, for future research.

## KEYWORDS

feature models, t-wise coverage, weighted sampler

**ACM Reference Format:**
Eduard Baranov, Axel Legay, and Kuldeep Meel. 2020. Baital: An Adaptive Weighted Sampling Approach for Improved t-wise Coverage. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

The software has been one of the primary driving forces in the transformation of humanity in the past half-century; in the modern world, software touches every aspect of modern lives ranging from medical, legal, judicial to policymaking. The widespread and diverse usage has led to the design of highly configurable software systems operating in diverse environments. Since software failures can lead to catastrophic effects, adequate testing of configurable systems is paramount. Testing of configurable systems adds complexity on top of an already notoriously difficult problem of testing standard software.

In the context of configurable systems, every configuration refers to an assignment of values to different parameters. For the exposition, we will restrict our discussion to parameters that only take binary values; the techniques proposed in this work are general and applicable to parameters whose possible set of values form a finite set, and the benchmarks employed in our empirical study arise from such domains. The primary challenge in the testing of configurable systems arising from the observation that bugs often arise due to interactions induced by the combination of parameter values. In the combinatorial testing literature, the term feature is often used to indicate a given parameter value. One such example is an extensive study by Abal, Brabranc, and Wasowski[1] that identified 42 bugs caused by the feature combinations in the Linux kernel. Furthermore, modeling of system and environment leads to constraints over the possible set of configurations of interest.

Combinatorial testing, also known as combinatorial interaction testing (CIT), has emerged as one of the dominant paradigms for testing of configurable software wherein the focus is to employ techniques from diverse areas to generate test suites to attain high coverage. One of the widely used metrics is t-wise coverage, wherein the focus is to achieve coverage of all combinations of features of size $t$.

A fundamental problem in CIT is the generation of test-configuration that seeks to maximize t-wise coverage, which is measured as the fraction of feature combinations appearing in the test set out of the possible valid feature combinations. The complexity of the problem arises from the presence of constraints to capture the set of invalid configurations. The holy grail of test generation in CIT is the design of test generation methods that can handle complex constraints, scale to systems involving thousands of features, and achieve higher t-wise coverage. Since achieving *high* t-wise coverage can be feasible for large values of $t$, the practitioners often focus on small values of $t \in \{1, 2, 3\}$, wherein $t = 1$ corresponds to achieving feature-wise coverage.

For long, uniform sampling has been viewed as a dominant domain-agnostic paradigm to achieve higher t-wise coverage, as demonstrated by theoretical and empirical analysis [41, 51]. As an example, the accepted solution for SPLC 2019 challenge, *Product Sampling for Product Lines: The Scalability Challenge*, was uniform sampler, Smarch, contributed by Oh, Gazzillo, and Batory [48]. Uniform sampling seeks to sample each valid configuration with equal probability. Recent works [48, 53] present tools that are capable of performing uniform sampling and of generating a partial covering array for large feature models. Uniform sampling provides an excellent heuristic to choose samples that would cover various feature combinations; however, this approach has a limitation since feature combinations are not distributed equally among the configurations: some of them can appear in millions of configurations while others only in tens. This fact prevents uniform sampling from achieving high t-wise coverage. In [48] a feature model was shown to have only about 48% pairwise coverage achievable with uniform sampling for any reasonable number of samples, while 70% coverage required more than $10^{10}$ samples. It is perhaps worth highlighting strengths of the uniform sampling approach: (1) domain agnosticism, and (2) principled use of the progress in SAT solving since the problem of uniform sampling is well formulated and fundamental problem with close relationship to other problems in other problems in complexity theory [3, 7, 23]. In this context, one wonders: *whether there exists a domain-agnostic alternative to uniform sampling which can benefit from advances in formal reasoning tools?*

The key contribution of this work is an affirmative answer to the above question: We present an adaptive literal-weighted sampling approach, called Baital, that achieves significantly higher t-wise coverage. Our formulation builds on the recent advances in formal methods community in the development of distribution-aware sampling for the distributions specified using literal weighted functions, a rich class of distributions with a diverse set of applications. In contrast to prior sampling-based approaches that advocate sampling from a fixed distribution, Baital adapts the distribution based on the generated samples. Therefore, Baital follows a multi-round architecture wherein the initial weight function is uniform, and then the weight function is updated after each round. The update of weight function typically requires the underlying sampling algorithm to redo the entire work from scratch. To allow the reuse of the computation, we adapt the recently introduced knowledge compilation-based approach for weighted sampling [20].

We introduce several strategies for the modification of literal-weight function. Through an extensive empirical evaluation, we demonstrate Baital achieves significantly higher t-wise coverage than uniform sampling. As an example, the accepted solution to the SPLC 2019 challenge could achieve less than 50% 2-wise coverage while Baital can achieve over 90% coverage with just 1000 samples. Our extensive comparison among different proposed strategies reveals surprising high coverage achieved by strategies based on limited statistics from the generated samples.

In summary, this work introduces a new paradigm based on adaptive weighted sampling to achieve high t-wise coverage that can benefit from advances in knowledge compilation. The high coverage obtained by Baital leads to several exciting directions of future work: First, we would like to develop a deeper understanding of the theoretical power of literal weight functions in the context of t-wise coverage seems a promising research area. Secondly, our work highlights challenges of measuring t-wise coverage for larger $t$, and the development of efficient algorithmic techniques to estimate such coverage would be beneficial in evaluating different techniques and aid in the development of more strategies for Baital.

## 2 BACKGROUND

### Boolean Formulas and Weight Function

A literal is a boolean variable or its negation. A clause is a disjunction of a set of literals. A propositional formula $F$ in conjunctive normal form (*CNF*) is a conjunction of clauses. Let $Vars(F)$ be the set of variables appearing in $F$. The set $Vars(F)$ is called *support* of $F$. A *satisfying assignment* or *witness* of $F$, denoted by $\sigma$, is an assignment of truth values to variables in its support such that $F$ evaluates to true. We denote the set of all witnesses of $F$ as $R_F$. Let $var(l)$ denote the variable of literal $l$, i.e., $var(l) = var(\neg l)$ and $F_{|l}$ denotes the formula obtained when literal $l$ is set to true in $F$.

Given a propositional formula $F$ and a weight function $W(\cdot)$ that assigns a non-negative weight to every literal, the weight of assignment $\sigma$ denoted as $W(\sigma)$ is the product of weights of all the literals appearing in $\sigma$, i.e., $W(\sigma) = \prod_{l \in \sigma} W(l)$. Without loss of generality, we assume that the weight function is normalized, i.e., $W(l) + W(\neg l) = 1$. The weight of a set of assignments $\mathcal{U}$ is given by $W(\mathcal{U}) = \sum_{\sigma \in \mathcal{U}} W(\sigma)$. Note that, we have overloaded the definition of weight function $W(\cdot)$ to support different arguments – a literal, an assignment and a set of assignments.

### t-wise Coverage

The formulation of combinatorial interaction testing (CIT) assigns a variable corresponding to every feature. Let $X$ be the set of all the variables (corresponding to features). Furthermore, every configuration $\sigma \in 2^X$ can be represented as conjunction of the set of literals of size $|X|$, where $|\cdot|$ denotes size of a set. For example, let $X = \{x_1, x_2, x_3\}$, then $\sigma = \{x_1, x_3\}$ can be equivalently represented as $\{x_1, \neg x_2, x_3\}$

Given a configuration $\sigma \in 2^X$ represented as a set of literals, let $\mathrm{Comb}(\sigma, t)$ represent the set of t-sized feature combinations due to $\sigma$, which is essentially the set of all subsets of literals of the size $t$ in $\sigma$. For a set $\mathcal{U} \subseteq 2^X$, we can extend the notion of Comb and denote $\mathrm{Comb}(\mathcal{U}, t) = \cup_{\sigma \in \mathcal{U}} \mathrm{Comb}(\sigma, t)$. Note that while for a given $\sigma$, $|\mathrm{Comb}(\sigma, t)| = \binom{|X|}{t}$ but this does not imply $|\mathrm{Comb}(\mathcal{U}, t)| = |U| \times \binom{|X|}{t}$ since $|\mathrm{Comb}(\sigma_1, t) \cup \mathrm{Comb}(\sigma_2, t)|$ is not necessarily equal to $|\mathrm{Comb}(\sigma_1, t)| + |\mathrm{Comb}(\sigma_2, t)|$.

t-wise coverage of a set $\mathcal{U}$ is defined as a ratio between number of t-sized combinations due to $\mathcal{U}$ over the total number of t-sized combinations. Feature models have constraints over a set of variables defined with a formula $F$, therefore for t-wise coverage only configurations that are witness of $F$ are considered. Formally, $\mathrm{Cov}(\mathcal{U}, t) = |\mathrm{Comb}(\mathcal{U}, t)| \;/\; |\mathrm{Comb}(R_F, t)|$.

Allowing only a fixed number of configurations, the coverage optimisation problem searches for a fixed-sized set of configurations that maximises the t-wise coverage. Given a formula $F$ over $X$, size of feature combination $t$ and allowed number of samples $s$, the problem of $\mathrm{OptTCover}(F, t, s)$ seeks for $\mathcal{U}$ such that:

$$\mathcal{U} = \underset{\mathcal{U} \subseteq R_F, |\mathcal{U}|=s}{\mathrm{argmax}} |\mathrm{Cov}(\mathcal{U}, t)|.$$

For a given $F$, $t$ and $s$, an ideal test generator seeks to solve $\mathrm{OptTCover}(F, t, s)$

### Knowledge Compilation

We focus on subsets of Negation Normal Form (NNF) where the internal nodes are labeled with disjunction ($\vee$) or conjunction ($\wedge$) while the leaf nodes are labeled with $\bot$ ($false$), $\top$ ($true$), or a literal. For a node $v$, let $\vartheta(v)$ and $Vars(v)$ denote the formula represented by the DAG rooted at $v$, and the variables that label the descendants of $v$, respectively.

We define the well-known decomposed conjunction [12] as follows:

**Definition 2.1.** A conjunction node $v$ is called a *decomposed conjunction* if its children (also known as conjuncts of $v$) do not share variables. Formally, let $w_1, \ldots, w_k$ be the children of $\wedge$-node $v$, then $Vars(w_i) \cap Vars(w_j) = \emptyset$ for $i \neq j$.

If each conjunction node is decomposed, we say the formula is in *Decomposable* NNF (DNNF).

**Definition 2.2.** A disjunction node $v$ is called *deterministic* if each two disjuncts of $v$ are logically contradictory. That is, if $w_1, \ldots, w_n$ are the children of $\vee$-node $v$, then $\vartheta(w_i) \wedge \vartheta(w_j) \models false$ for $i \neq j$.

If each disjunction node of a DNNF formula is deterministic, we say the formula is in deterministic DNNF (d-DNNF), and we can perform tractable model counting on it.

## 3 BAITAL: ACHIEVING HIGHER T-WISE COVERAGE

In this section, we first discuss the relationship of OptTCover$(F, t, s)$ with the classical problem of set cover. We then discuss the limitations of lifting techniques in the context of set cover to configuration testing due to presence of constraints. Inspired by the classical greedy search in the context of set cover, we design BAITAL, a novel framework that employs adaptive weighted sampling combined with recently proposed knowledge compilation-based sampling approach to achieve efficient sampling routines that achieves high coverage. The development of BAITAL leads to the need for strategies for update of weight functions; to this end, we present several strategies that seek to use different statistics from the generated sampling.

### 3.1 Relationship with Set Cover

As a starting point, we observe that the problem of OptTCover$(F, t, s)$ can be reduced to that of the optimization variant of the classical problem of set cover [26]; the variant is also referred to as Maximum Coverage. The reduction to set cover allows us to simply modify the classical greedy search strategy to obtain a $\mathcal{U}$ such that Cov$(U, t)$ is at least $(1 - 1/e)$ of the optimal solution $\mathcal{U}^*$ of OptTCover$(F, t, s)$. We present the algorithm below for completeness and to discuss its simplicity and yet its impracticality in practice.

**Algorithm 1:** GreedyCovertion$(F, s)$

```
1 $\mathcal{S} \leftarrow \emptyset$
2 for $i \leftarrow 0$ to $s$ do
3     $\sigma \leftarrow$ MaxDistSolution$(F, \mathcal{S})$
4     $\mathcal{S} \leftarrow \mathcal{S} \cup \{\sigma\}$
5 return $\mathcal{S}$
```

The algorithm GreedyCovertion is presented in Algorithm 1. GreedyCovertion assumes access to the subroutine MaxDistSolution that takes in input $F$ and $\mathcal{S}$, and returns $\sigma^*$ such that

$$\sigma^* = \underset{\sigma \in R_F}{\text{argmin}} \, |\text{Cov}(\sigma, t) \cap \text{Cov}(\mathcal{S}, t)| \quad (1)$$

The following proposition follows from classical result [43].

PROPOSITION 3.1. *Cov$(\mathcal{S}, t) \geq (1 - \frac{1}{e} + o(1))$OptTCover$(F, t, s)$, where $(1 - \frac{1}{e} + o(1)) \approx 0.632$.*

While GreedyCovertion is a simple algorithm, the roadblock lies in ensuring an efficient implementation of MaxDistSolution. Recall that the set of $|\text{Cov}(\sigma, t)| = \binom{|X|}{t} \in \Omega((n/t)^t)$, therefore, even obtaining a $P^{NP}$ subroutine for MaxDistSolution seems a daunting challenge. We are not aware of any polynomial reduction of MaxDistSolution to polynomially many MaxSAT queries. It is worth emphasizing the input size is $|F| + |\mathcal{S}|$. We leave an efficient implementation of GreedyCovertion as an open question.

### 3.2 Adaptive Weighted Sampling

It is worth recalling that MaxDistSolution seeks to find $\sigma$ whose Cov$(\sigma, t)$ has the smallest intersection with the existing Cov$(\mathcal{S}, t)$. The lack of efficient implementation of MaxDistSolution makes us wonder if one can employ a randomized sampling-based approaches to find $\sigma$ that seeks to achieve the goal of MaxDistSolution.

The usage of sampling has long been explored in the context of ideal test generation. In fact, random testing is the recommended strategy [30]. The key idea is to sample solutions of $F$ uniformly at random. The past few years is witness to several efforts to understand the scalability of uniform samplers in the context of CIT. Furthermore, in response to SPLC Challenge 2019 [50], the accepted tool Smarch generated a set of uniformly distributed samples and revealed that uniform sampling could reach only 48% pairwise coverage with 1000 samples [47]. Moreover, the coverage growth with the number of samples was prolonged, requiring a million samples to reach 50% coverage. The weak performance of uniform sampling can be attributed to an uneven distribution of feature combinations among the configurations: having $10^{14}$ configurations one-third of literals (and consequently all feature combinations involving them) are part of less than $10^6$ configurations. The probability of covering these combinations with uniform sampling is minuscule. In this regard, we seek to find different sampling strategies that can achieve higher coverage.

The key contribution of our work is an adaptive weighted sampling-based generation of tests BAITAL. The core architecture of our framework is a multi-round process wherein each round seeks to generate samples using a weighted sampler. In contrast, the weight distribution is adjusted based on the samples generated so far. To accomplish an efficient procedure, we need to tackle three challenges:

**Challenge 1: Representation of weights over assignment space**
We now present a rationale for our design choices to handle the above three challenges. To represent weights over assignment space, we turn to a literal-weighted function that assigns a non-negative weight to every literal such that the weight of an assignment is the product of the weight of its literal. The choice of literal-weighted function is primarily motivated due to the observation that a wide variety of distributions arising from diverse disciplines can be represented as literal-weight function [13, 20].

**Challenge 2: Dynamic update of weight function**
Since the choice of literal weight function allows us to concentrate on performing the update of only the literal weights, our strategy would be to collect statistics corresponding to every literal and then assign the weights accordingly.

**Challenge 3: Efficient weighted sampling techniques that can handle incremental queries**
Prior techniques that seek to employ sampling techniques either fall into uniform sampling methods that solely focus on drawing uniform sampling and hope to achieve higher coverage. On the other hand, techniques that seek to induce bias in the distributions update the constraints and require the underlying sampling engine to perform computations from scratch. In this work, our critical insight is to build on recent

advances in knowledge compilation, a field in Artificial Intelligence that focuses on the representation of constraints in a representation language into a target language where different queries are often tractable. In particular, we seek to represent the initial set of constraints of $F$ into a representation language $\mathcal{T}$ such that we can perform an update of the weights and subsequent sampling in time linear in the size of $\mathcal{T}$ without needing to perform the compilation process again. As is discussed in Section 2, the process of compilation is often the most expensive, and therefore, our process amortizes the cost of compilation with the generation of more samples.

We now first present an overview of Baital and then discuss each of the components in detail in the following sections.

In order to overcome the limitation of the uniform sampling, configurations with rare feature combinations shall have a higher probability of being chosen. Indeed, the precomputation of weights for all configurations is infeasible. Therefore, we are proposing to adjust the weights dynamically by examining already sampled configurations. This way, we can force the sampler to prefer configurations with uncovered feature combinations over the other ones. The algorithm runs as follows: a first few samples are generated according to the uniform distribution of configurations. After this step, the sampler pauses while the examination of the chosen configurations is done. The algorithm estimates which feature combinations have been covered already and chooses the weights for the next round of sample generation accordingly. The next several samples are chosen following the new weight distribution. The process repeats until the desired number of samples is generated.

**Algorithm 2:** AdaptiveWeightedSampling($F$, *rounds*, *spr*)

```
1 $\mathcal{S} \leftarrow \emptyset$
2 $vars \leftarrow$ Set of variables in $F$
3 $\mathcal{T} \leftarrow Compile(F)$
4 for $roundNb \leftarrow 0$ to $rounds$ do
5     $weights \leftarrow generateWeights(vars, \mathcal{S})$
6     $newSamples \leftarrow getSamples(F, weights, spr)$
7     $\mathcal{S} \leftarrow \mathcal{S} \cup newSamples$
8 return $\mathcal{S}$
```

The overall algorithm is shown in Algorithm 2. It has three input parameters: the CNF formula $F$ defining constraints on a feature model, the number of rounds *rounds*, and the number of samples generated at each round *spr*, i.e. the total number of samples to be generated is $rounds \times spr$. We first compile the formula $F$ into a d-DNNF $\mathcal{T}$. At the start of each round (line 4) the algorithm generates an assignment of weights to variables that depends on the set of samples generated during the previous rounds. We discuss *generateWeights* function in details in the next subsections. The literal-weighted sampling algorithm is called to find *spr* samples chosen according to the distribution imposed by generated weights (line 6). For the first round, the weights are always equal to 0.5, corresponding to the uniform distribution. The new samples are added to the result (line 7), and the procedure repeats *rounds* times.

## 3.3 Weighted Sampling via Compilation

In [53] it was observed that advances in knowledge compilation can be used to design efficient uniform samplers and designed an efficient uniform sampler KUS. In [20] KUS was extended to design a weighted sampler WAPS. Given an input formula $F$ and the desired number of samples $s$, WAPS uses a three staged process: **Compilation**, **Annotation**, and **Sampling**.

**Compilation** Given a formula $F$, a state of the art compiler is employed to obtain the equivalent d-DNNF $\mathcal{T}$.

**Annotation** A bottom-up subroutine is called to annotate every node $v$ of $\mathcal{T}$ with its corresponding weight, i.e., the label for the node $v$ is $W(\vartheta(v))$. It is worth highlighting that for a formula in d-DNNF, the labeling of every node with its weight can be accomplished in time linear in the size of $\mathcal{T}$.

**Sampling** To perform sampling, a top-down subroutine is followed wherein for every $\wedge$ node, once the samples from all the children are generated, we randomly shuffle each of the lists and then conjoin the list of samples. In the case of $\vee$ node, we first determine, via the Binomial coefficient, the number of samples each of the nodes should generate, and then we take the union of all the lists.

It is known that there exist polynomially sized formulas whose d-DNNF representations are exponential [12]. Furthermore, the compilation process may require a significantly larger time than the size of resulting $\mathcal{T}$. For example, for every formula $F$ that is valid, there exists a polynomially sized d-DNNF $\mathcal{T}$, but the existence of PTIME compilation process would imply P=Co-NP. Fortunately, our adaptive sampling strategy only modifies the weight function and does not alter $F$. We make the following key observation about **Annotation** and **Sampling** in WAPS that strongly supports our choice of knowledge compilation-based approach for Baital.

Proposition 3.2. ***Annotation*** *and* ***Sampling*** *can be accomplished in time linear in the size of* $\mathcal{T}$

Proof. The proof follows trivially from the algorithmic description of **Annotation** and **Sampling** in [20, 53]. Essentially, the key observation is that **Annotation** is performed in a bottom-up process while visiting each node exactly once, while **Sampling** is performed in a top-down manner while visiting each node exactly once. □

The weighed sampling procedure getSamples from Algorithm 2 is shown in Algorithm 3.

**Algorithm 3:** getSamples($\mathcal{T}$, weights, spr)

```
1 Annotate($\mathcal{T}$, $weights$)
2 $\mathcal{S} \leftarrow$ Sample($\mathcal{T}$, $spr$)
3 return $\mathcal{S}$
```

## 3.4 Round Weights Generation

In literal-weighted sampling, configurations are chosen according to the distribution imposed by a weight function $W$. As shown in Section 2, $W$ is defined by assigning a (normalized) weight to each literal.

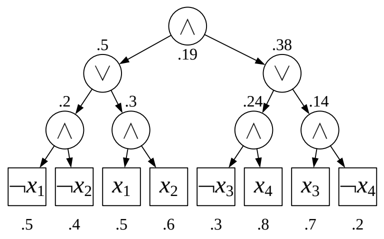

**Figure 1: Annotated d-DNNF corresponding to $F := (x_1 \leftrightarrow x_2) \wedge (x_3 \leftrightarrow \neg x_4)$. The weight to literals are annotated next to the corresponding leaves.**

For the first round the weight function is constant, $W(l) = 0.5$ for any literal $l$ which corresponds to the uniform distribution. For the following rounds the definition of $W$ is based on the knowledge about a feature model, in particular its constraints $F$, and on a set of already generated samples $\mathcal{S}$. Let $g(l, t, \mathcal{S})$ be a function that represents the knowledge about the t-sized feature combinations involving a literal $l$ in the set of samples $\mathcal{S}$. Let $h(l, t, F)$ be a function that represents the knowledge about the t-sized feature combinations involving a literal $l$ by configurations in $R_F$. We define the weight function as follows:

$$W(l, t, \mathcal{S}, F) = f(g(l, t, \mathcal{S}), h(l, t, F), g(\neg l, t, \mathcal{S}), h(\neg l, t, F)), \quad (2)$$

where $F$ is a propositional formula, $l$ is a literal, $t$ is a size of feature combinations, $\mathcal{S}$ is a set of already generated samples, and $f$ is a function outputting a value in the interval $[0, 1]$. Since the weights of a literal and its negation must be related, the weight function is dependent on the feature combinations of the literal negation as well. Each round of Baital starts with the computation of $W$ (line 5 of Algorithm 2) which is further used for sample generation.

There exist multiple ways to define functions $f$, $g$, and $h$ resulting in different weight functions and, consequently, in different t-wise coverage. We have considered several options to define these functions, which we call strategies. In the remaining, we assume that $F$ and $t$ are fixed and omit them. Notice that $h(l)$ depends only on fixed $F$ and $t$ and, therefore, is not changing between the rounds and can be computed only once.

**Algorithm 4:** generateWeights($vars$, $\mathcal{S}$) // Strategy 1

```
1 for $l$ in $literals$ do
2     $g[l] \leftarrow$ number of distinct t-sized feature combinations
          with $l$ in $\mathcal{S}$
3     $h[l] \leftarrow$ number of distinct t-sized feature combinations
          with $l$ in $R_F$
    /* $h$ is computed in the first round only */
4 for $l$ in $literals$ do
5     $diff[l] \leftarrow g[l] \,/\, h[l] \;-\; g[\neg l] \,/\, h[\neg l]$
6     $weights[l] \leftarrow 0.5 * (1 - \text{sign}(diff) * \text{sqrt}(\text{abs}(diff)))$
7 return weights
```

**Strategy 1.** The first option is to define $g$ and $h$ as the number of distinct feature combinations involving each literal in a set of samples and in $R_F$, respectively. For the computation of $h$ it is not necessary to enumerate all configurations in $R_F$, it can be done by checking, for each feature combination $\{l_1, \ldots, l_t\}$, satisfiability of $F \wedge l_1 \wedge \cdots \wedge l_t$. Indeed, since the SAT solver can provide an assignment in case the formula is satisfiable, during the computation of $h$, the full covering array is implicitly built. However, for 1000 variables, such array would contain millions of samples, thus being impractical.

The heuristic behind this strategy is the following: if the majority of feature combinations involving a literal are already covered but it is not the case for the negation of the literal (or vice versa), then the sampler should prefer configurations involving a negation of a literal in the next round. We use the difference between ratios of covered feature combinations for literal and its negation as a defining value for the weight. Therefore, function $f$ is computing this difference and transforms it into the interval $[0, 1]$. Algorithm 4 illustrates the weight generation for strategy 1 with a pseudo-code. First, $g$ and $h$ are computed over lines 2 and 3. In the following step, $f$ is computed over lines 5 and 6. The choice of sqrt-based transformation function line 6 is done after the empirical evaluation of several options.

Strategy 1 involves a lot of computations for functions $g$ and $h$. In particular, $h$ checks satisfiability of $2^t * \binom{Vars(F)}{t}$ formulas. Secondly, counting the number of distinct combinations in the set of samples has a high time and memory costs for large real-world models involving thousands of variables. Therefore, we considered several modifications of the weight function that would require fewer resources for computation and evaluated their effect on t-wise coverage.

**Algorithm 5:** generateWeights($vars$, $\mathcal{S}$) // Strategy 2

```
1 for $l$ in $literals$ do
2     $g[l] \leftarrow$ number of distinct t-sized feature combinations
          with $l$ in $\mathcal{S}$
3     $h[l] \leftarrow 2^{t-1} * \binom{Vars(F)-1}{t-1}$
4 for $l$ in $literals$ do
5     $diff[l] \leftarrow g[l] \,/\, h[l] \;-\; g[\neg l] \,/\, h[\neg l]$
6     $weights[l] \leftarrow 0.5 * (1 - \text{sign}(diff) * \text{sqrt}(\text{abs}(diff)))$
7 return weights
```

**Strategy 2.** In this strategy we attempt to reduce the computation cost by simplifying $h$. Without checking the existence of configurations in $R_F$ involving each feature combination, $h$ can be overapproximated by the number of t-sized feature combinations in $2^{Vars(F)}$ involving a particular literal. Algorithm 5 shows the $generateWeights$ function for the **Strategy 2**. The only difference from Algorithm 4 is on line 3, since $f$ and $g$ are defined as in **Strategy 1**.

**Strategy 3.** Both **Strategies 1** and **2** are trying to optimize the coverage for a particular size $t$ of feature combinations. Indeed, the computation of $g$ and $h$ depends on $t$, and for the same set of samples, the weights generated for $t = 2$ and $t = 3$ would be different. Nevertheless, one could expect that a set of samples with high pairwise coverage would also have high 3-wise coverage. This hypothesis allows us to gradually simplify the computation of $g$.

**Algorithm 6:** generateWeights($vars$, $\mathcal{S}$) // Strategy 3

```
for $l$ in $literals$ do
    $g[l] \leftarrow |\sigma \in \mathcal{S} \mid l \in \sigma|$
    $h[l] \leftarrow |\sigma \in R_F \mid l \in \sigma|$
for $l$ in $literals$ do
    $\text{diff}[l] \leftarrow g[l]/(g[l] + g[\neg l]) - ln(h[l])/(ln(h[l]) + ln(h[\neg l]))$
    $\text{weights}[l] \leftarrow 0.5 * (1 - \text{sign(diff)} * \text{sqrt(abs(diff))})$
return weights
```

A very simple and easily computable measure of literal participation in a set of samples is the number of samples the literal is present. It allows finding the number of feature combinations the literal is involved in, though not the number of distinct combinations. Similarly, $h$ in this strategy is defined as the total number of configurations involving the literal. $h$ can be computed by calling #SAT $Vars(F) + 1$ times (any configuration involves either a literal or its negation but not both). Since the values of $h$ and $g$ can differ by several orders of magnitude, we changed $f$: it is comparing the ratio of samples involving the literal in the sample set and the ratio of configurations involving a literal on a logarithmic scale. The weight generation function is shown in Algorithm 6.

**Algorithm 7:** generateWeights($vars$, $\mathcal{S}$) // Strategy 4

```
for $l$ in $literals$ do
    $g[l] \leftarrow |\sigma \in \mathcal{S} \mid l \in \sigma|$
for $l$ in $literals$ do
    $\text{weights}[l] \leftarrow 0.5 * (1 - (g[l] - g[\neg l]) \ / \ (g[l] + g[\neg l]))$
return weights
```

**Strategy 4.** In this strategy we modify **Strategy 3** by fixing $h$ constant. The hypothesis of this strategy considers that since values of $g$ and $h$ differ by several orders of magnitude, their direct or indirect comparison might not be useful. Therefore, in this strategy, we simply compute $g$ and compare the appearances of each literal with its negation. The function is shown in Algorithm 7.

## 4 EXPERIMENTS

For the analysis of our approach we have implemented a prototype of Baital in Python[1]. It uses the literal-weighted sampler WAPS[20], which in its turn needs D4 [31] for d-DNNF compilation. To evaluate our approach we designed a set of experiments helping to answer the following research questions.

Our main goal is to push forward the t-wise coverage of the state-of-the-art approach for large SPLs, therefore the first two questions naturally arise.

**RQ1** Can Baital be used to generate partial covering arrays for large SPLs?

**RQ2** Can Baital achieve higher t-wise coverage than uniform sampling?

In addition, we are also interested in the evaluation of effects of parameters in our approach on the efficiency and performance.

[1]Implementation and benchmarks can be found in supplemental materials

**RQ3** How often shall weight function be regenerated in order to obtain best coverage/performance?

**RQ4** What is the impact of different strategies on coverage and performance?

### 4.1 Benchmarks

We took a large number of publicly available feature models from real-world configurable systems that were used before in evaluation of uniform sampling tools. In particular, we took the majotity of the benchmarks appearing in [29, 35, 51]. Few largest feature models were excluded from our benchmark set as neither uniform sampling tools nor our implementation were able to produce a partial covering array in our settings. Our benchmark set consists of 123 feature models. The smallest feature model JHipster has 45 variables and 104 clauses. There are two a bit larger models with approximately 500 variables and 1000 clauses. The remaining benchmarks have at least 998 variables and the majority of them have between 2968 and 4138 clauses, while 9 of them have around 50000 clauses with the largest one involving 62183 clauses. The configurable system with the highest number of variables is a uClinux feature model with 11254 features and 31637 clauses. The number of configurations in the benchmarks ranges between $10^{91}$ and $10^{417}$ for all benchmarks except the 3 small ones.

For a more detailed exploration of strategies proposed in Section 3.4 (the last two research questions), we have chosen another 3 benchmarks of different size and complexity. Two feature models were taken from [29] and transformed into CNF formulas with FeatureIDE 3.6.0[2]: "axTLS" with 94 variables and 190 clauses, "embtoolkit" with 2128 variables and 15483 clauses. The last benchmark is taken from [50], "FinancialServices01" version "2018-05-09" ("financial" in the rest of the paper) has 771 variables and 7241 clauses. It is particularly interesting since uniform sampling cannot achieve high coverage even with $10^7$ samples (below 70% for 1-wise coverage and below 50% for 2-wise coverage) [48].

### 4.2 Experiments, Settings and Competitors

The first experiment is designed to compare Baital with the state-of-the-art uniform sampling. We generated a 1000 samples with both uniform sampling and Baital for each feature model from our benchmark set. In our approach samples were generated in 10 rounds, 100 samples each round. In the preparation of experiment we tried several tools for uniform sampling, namely QuickSampler [16], Smarch [47], KUS[53] and WAPS with weights corresponding to the uniform sampling. Quicksampler despite being fast checks the validity of samples at the very last step, therefore the number of valid samples is usually lower than the requested one. In many cases it generated 1000 samples none of which were valid. Also it was shown in [51] that the results of Quicksampler are often far from uniform. The remaining 3 tools were able to generate uniformly distributed sets of samples and, unsurprisingly, the resulted t-wise coverage was almost identical. Comparing their execution time, KUS was the fastest, closely followed by WAPS, while Smarch was considerably slower taking 1000 times more time on some of the benchmarks. Considering a minor difference between KUS and

[2]https://github.com/FeatureIDE/FeatureIDE/releases/tag/v3.6.0

| Benchmark | NVars | Uniform | Strategy 2 | Strategy 4 |
|---|---|---|---|---|
| mpc50 | 1213 | 2169357 | 2508473 | 2670210 |
| pc_i82544 | 1259 | 2291769 | 2741419 | 2887288 |
| refidt334 | 1263 | 2330423 | 2765357 | 2941096 |
| dreamcast | 1252 | 2308471 | 2650081 | 2842723 |
| XSEngine | 1260 | 2342488 | 2718346 | 2901167 |
| busybox_1_28_0 | 998 | 1936827 | 1952827 | 1955269 |
| integrator_arm9 | 1267 | 2337336 | 2737317 | 2926949 |
| ecos-icse11 | 1244 | 2299262 | 2674216 | 2842662 |
| freebsd-icse11 | 1396 | 3564507 | 3712877 | 3720593 |
| uClinux-config | 1850 | 118043805 | 132583812 | 133987974 |

**Table 1: Number of distinct feature combinations obtained in 1000 samples**

WAPS, we decided to use the latter one, allowing to better evaluate the additional complexity of our approach.

For the first experiment, we used only **Strategies 2** and **4** in Baital since they do not require additional precomputations of $h$ on each benchmark. Indeed, the computation of all feasible pairwise combinations for uClinux-config feature model would require 250 millions of SAT solver calls. For the same reason, we do not compare the t-wise coverage on these benchmarks but the number of distinct feature combinations in the sample sets, i.e. $|\text{Comb}(\mathcal{S}, t)|$. In addition, we limited the exploration to the pairwise coverage as the computation of Comb for higher-wise coverage were exceeding available RAM on the majority of experiments.

The second experiment is designed to evaluate the parameters of Baital. The experiment was conducted on "axTLS", "embtoolkit" and "financial" benchmarks. On each of the benchmarks we generated sample sets varying strategies from Section 3.4 and round lengths. The comparison was done for pairwise coverage and execution time. The results of uniform sampling were used as a baseline.

Due to the probabilistic nature of both uniform sampling and Baital, all experiments have been run 5 times and the reported results show the mean values. Nevertheless, for the majority of the benchmarks, the difference between the highest and the lowest coverage or the number of distinct feature combinations for the same input parameters was below 0.5% with the maximum value on financial benchmark 1.8%. All experiments were conducted on a laptop with Intel(R) Core(TM) i7-8650U CPU, RAM limited to 12GB, running Ubuntu 18.04.3 LTS.

## 4.3 Comparison with Uniform Sampling (RQ1, RQ2)

The first experiment compared the number of feature combinations in sets of samples generated by uniform sampling and Baital. Table 1 presents the excerpt of the results for several feature models showing the number of distinct feature pairs in 1000 generated samples[3]. For 2 benchmarks, 3 set of samples (uniform and Baital with **Strategies 2** and **4**) had equal number of feature pairs. Among the other 121 benchmarks, on 116 of them the **Strategy 2** showed the increase between 14% and 22% of the number of covered feature pairs and **Strategy 4** showed increase between 21% and 29%. The remaining 5 benchmarks showed smaller increase. Exploration of those feature models showed that both approaches obtained high coverage, therefore the difference is their results was not big. For example, busybox_1_28_0 has 98.5% coverage with uniform sampling, and Baital can still improve the result to 99.5%, though the difference in the number of feature combinations is only 4%.

The execution time of Baital depends on 3 major factors: sampling time of each round, computation of weight function for the following round and the number of rounds. The first factor is the execution time of WAPS. As explained in Section 3.3, WAPS execution consists of 3 steps: Compilation, Annotation, and Sampling. Since the formula is not modified during the sampling process, Compilation is done only in the first round. On the benchmarks this step takes several seconds with a highest value of 26 seconds and median 6.5 seconds.

The computation of the weight function for the next round depends on the strategy and the number of variables in the formula. For **Strategies 3** and **4**, as they just compute the number of appearances of a literal in samples, it takes less than one second even on the largest benchmark with 11254 variables. **Strategies 1** and **2**, on the other hand, need to find distinct feature combinations and their execution time also depends on the size of feature combinations. For pairwise sampling, it takes 1 second for 771 variables, 16 seconds for 1396 variables and up to 250 seconds for the largest benchmark with 11254 variables. Note that the set of distinct feature combinations do not need to be recomputed from scratch at each round, the results of first rounds can be reused in the latter ones. Therefore, the time does not increase with growth of the round number.

For the benchmarks, 102 have finished within 1 hour for both strategies, another 16 finished within 2 rounds. The remaining 5 benchmarks with the longest Annotation process took several hours. The difference between **Strategy 2** and **Strategy 4** was mostly within 10 minutes with the maximum of 35 minutes for the benchmark with 11254 variables.

As a result of this experiment we can conclude that Baital can generate partial covering arrays at reasonable time for large feature models. The resulted coverage of feature combinations is considerably higher than the uniform sampling for the vast majority of benchmarks.

## 4.4 Comparison of Weight Generation Strategies (RQ3, RQ4)

Figures 2, 3, 4 show the coverage on three benchmarks achieved with each strategy. The results of uniform sampling are shown for comparison. For all strategies, the weight function has been changed after every 100 samples and the coverage was computed after each generated sample. Therefore, the first 100 samples are generated uniformly in all strategies, resulting in almost identical coverage. For the second round, each strategy generated new weights, and the next set of samples are chosen with respect to the new distribution. The considerable boost over the uniform sampling can be noticed: strategies are helping to choose samples with feature combinations that didn't appear in samples before. In all three benchmarks, 200 samples generated with Baital have better coverage than 3000 samples chosen uniformly.

[3]Full version of Table 1 can be found in supplemental materials.

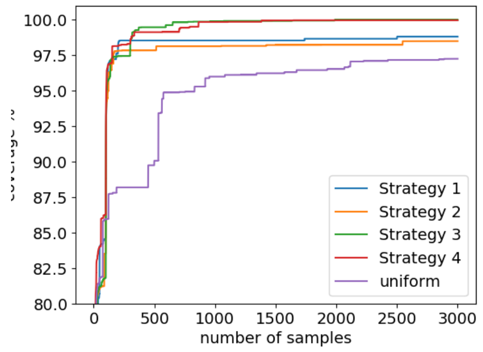

**Figure 2: Number of feature combinations for axTLS benchmark**

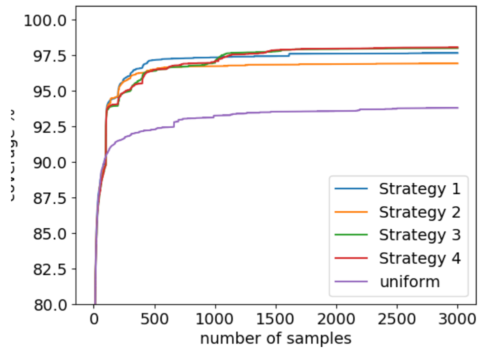

**Figure 3: Number of feature combinations for embtoolkit benchmark**

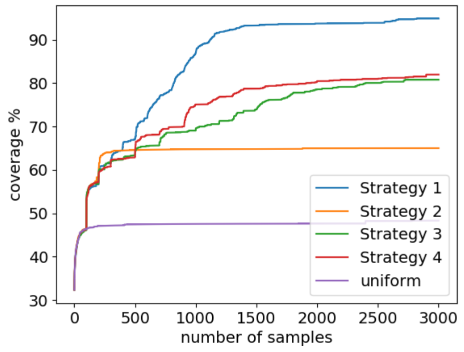

**Figure 4: Number of feature combinations for financial benchmark**

The difference between strategies becomes noticeable after second change of weights. In the first two benchmarks, the coverage is quite high even for the uniform sampling and results of different

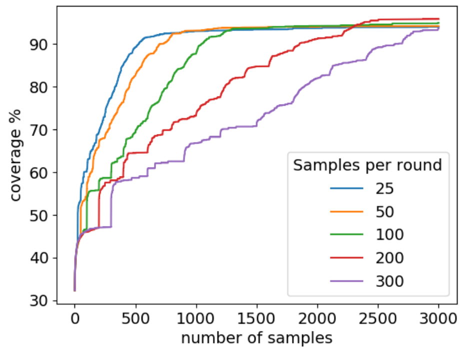

**Figure 5: Number of feature combinations for financial benchmark with Strategy 1**

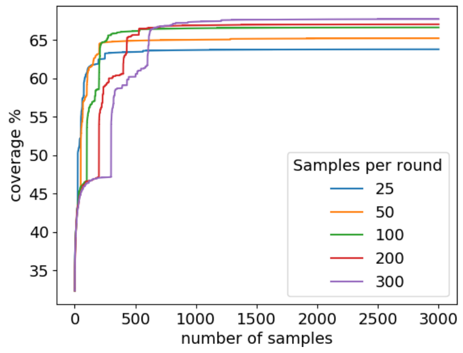

**Figure 6: Number of feature combinations for financial benchmark with strategy 2**

strategies are close. However, in the financial benchmark **Strategy 1** clearly outperforms other strategies, while **Strategy 2** is the worst though still have a much better result than uniform sampling. In this benchmark almost 25% of feature combinations are infeasible and they are not distributed equally between literals. This results in bad approximation of function $h(l, t)$ and, consequently, low performance of **Strategy 2**. **Strategies 3** and **4** showed very close results in all 3 benchmarks, therefore we can conclude that extra precomputations in **Strategy 3** do not improve to the resulted coverage.

Figures 5, 6, 7 show the dependency of different frequencies of weight function change on pairwise coverage in the financial benchmark. We omit the plot for **Strategy 3** as it has similar results to **Strategy 4**. These plots show that higher frequency change allows to obtain high coverage with fewer samples. However, by raising the number of generated samples, lower frequencies of weight updates can reach the same coverage or even slightly outperform it. This behaviour is a consequence of our choice of weight function. The difference between the weight functions of two consecutive rounds is small if few new feature combinations were covered during the first of the two rounds. Therefore, for the first several rounds,

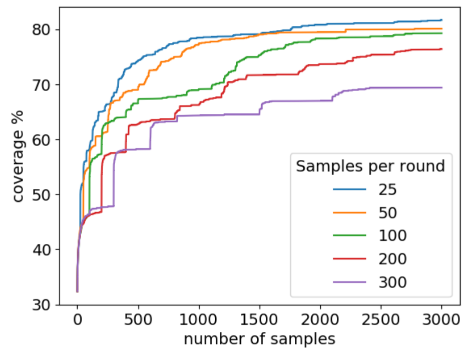

**Figure 7: Number of feature combinations for financial benchmark with strategy 4**

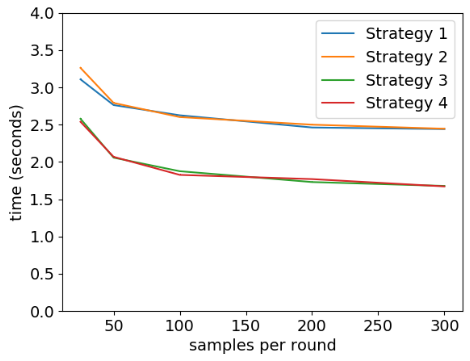

**Figure 8: Execution time for generation of 3000 samples for axTLS benchmark. Uniform sampling took 0.7 second.**

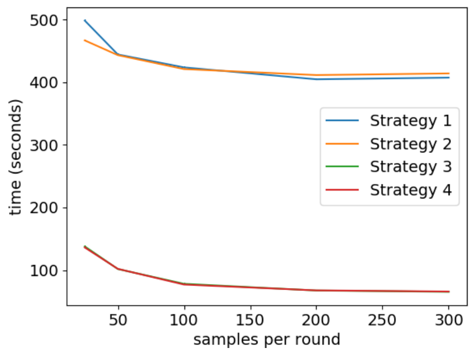

**Figure 9: Execution time for generation of 3000 samples for embtoolkit benchmark. Uniform sampling took 61 seconds.**

when a lot of new feature combinations are found, high frequency of weight function chance is beneficial to reach uncovered areas faster, while for the latter rounds, there would be not enough new combinations to noticeably modify weights distribution.

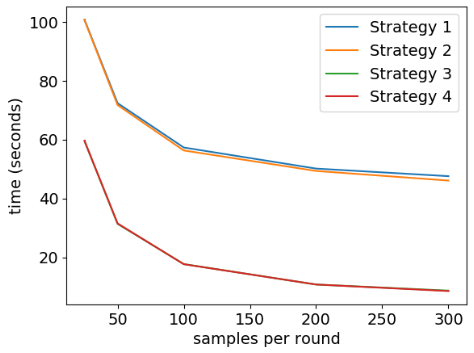

**Figure 10: Execution time for generation of 3000 samples for financial benchmark. Uniform sampling took 4 seconds.**

The evaluation of execution time on the benchmarks is shown on Figures 8, 9 and 10. The precomputations of function $h$ are not included in the total time. Strategies using the same function $g$ showed close results. Indeed, **Strategies 1** and **2** are slower than the other ones and the time difference depends on the number of variables. For a small axTLS benchmark, **Strategies 1** and **2** are just 1 second slower than **Strategies 3** and **4**, while they are 40 and 340 seconds slower for larger financial and embtoolkit benchmarks, respectively. Regarding the frequency of weight function generation, clearly, extra d-DNNF dag reannotations has negative impact on execution time, though feature models with relatively simple d-DNNF dag, such as embtoolkit, do not have strong effect of this parameter.

As a result of this experiment we can conclude that **Strategy 1** can generate the best partial covering array while being the slowest one. **Strategy 4** is the fastest one and provide a good coverage outperforming **Strategies 2** and **3**. For the frequency of weights change, to generate a small number of samples with high coverage, high frequency of weight function generation is preferable, however for larger sample sets it might be better to reduce it, as similar coverage could be obtained with shorter execution time.

## 4.5 Threats to Validity

**Construct Validity.** Many prior works use combinatorial interaction testing for feature models and t-wise coverage as qualitative metric for partial covering arrays for the cases where the model is too large and complex to compute a full covering array. Our approach is following these works by building partial covering arrays with better t-wise coverage.

**Internal Validity.** Due to the probabilistic nature of our approach, several runs may not yield identical results. Indeed, the choice of samples during the first round affects the weights for the following round. To mitigate the effect of this behaviour on the experiment results, we run each benchmark multiple times and report the mean value. In addition, we examined the result of different runs and noticed, that the difference between the best and the worst result was below 0.5% for the majority of benchmarks with a maximum 1.8%. We also noted that even the worst result was always better than the uniform sampling except two benchmarks where equality was obtained. For the baseline we used uniform sampling which is a state-of-the-art approach and we have tried several tools for uniform sampling that yield similar results.

**External Validity.** To mitigate the threat of non-generalisability of our study we have used a large number of feature models. These models cover a wide range in the number of features and constraints. These benchmarks were used before in several prior studies [29, 35, 48, 51].

## 5 RELATED WORK

Combinatorial interaction testing was initially proposed in [11] and has since been extensively studied. Many works on the topic were surveyed in [44] and a more recently in [57]. Multiple approaches have been proposed to build a set of samples for CIT. The approaches can be classified based on their access to the entire set of possible configurations.

The first set of approaches assume explicit access to the entire set of configuration. The greedy algorithm is one of the most popular ideas starting from the empty set of samples and adding new ones until the full coverage is achieved. There is a number of tools available including [10, 24, 39, 56, 59]. Another type of greedy algorithm initially build a set of samples with full coverage for the first $n$ parameters and then add an extra parameter at each interaction. IPO algorithm [34] with several extensions [32, 33] is following this idea as well other some works [8, 58]. Another group of approaches, sometimes called heuristical, start from some set of samples that do not provide full coverage and try to modify it in order to obtain the full coverage. Examples of such algorithms are [49], tabu search [19, 45] and genetic algorithm [40]. Lin et al. [36] attempted to combine heuristic and greedy approaches switching between them with a predefined probability, which allows a better exploration of configuration space and potentially smaller test suite. The requirement of explicit access to the entire set of configuration severely limits the above set of works in their ability to handle a large set of features.

The second set of approaches assume implicit access to the entire set of valid configuration: often represented as a set of solutions for a given set of constraints. Among constraint-based approaches, one set of techniques often seek to generate a random configuration and then check if the generated configuration satisfies the constraints [9, 15]. Such approaches can handle scenarios where the fraction of valid configurations over the set of all configurations is high but fail to handle cases where the density of valid configuration is low as typically observed in complex systems. It is worth noting that a low density of valid configurations does not imply a low number of valid configurations. For example, for 50 binary features, even if the set of valid configurations is $2^{40}$, the density of such a set is still $2^{-10}$.

Another set of constraint-based approach seeks to rely on employing uniform samplers constructed on top of recent advances in combinatorial solving such as SAT solving and knowledge compilation. Oh et al. [46] employed Binary Decision Diagrams for encoding the configurations of feature models. The tool SMARCH [47] uses #SAT solver for uniform sample generation. Lopez-Herrejon et al. [37] evaluated the dependency between t-wise coverage and the number of samples, building a Pareto front. Two uniform sampling tools QuickSampler [16] and Unigen2 [5] were evaluated in [51].

Combinatorial interaction testing (CIT) is not the only technique used to build sample sets for software product lines. Many approaches and their flavors have been surveyed in [2]. Several other approaches to choose samples for analysis of feature models, including t-wise sampling, one-disabled [1] and statement-coverage [56] were compared in [41].

Another application of the sampling of configurable systems is performance prediction [52, 54, 55] as different features and their interactions can affect the performance. In recent work, Kaltenecker et al. [25] proposed a new metric and the corresponding sampling method where the number of selected features in configurations are chosen according to some probability distribution. This method allows us to better distribute configurations for performance prediction. However, since this approach pursues a different goal from CIT and optimizes a different metric, it is not directly comparable with our approach.

The problem of weighted sampling has witnessed sustained interest from theoreticians and practitioners alike for the past three decades owing to its widespread usage. Of several proposed techniques, Monte Carlo Markov Chain(MCMC)-based methods[22, 38] enjoy widespread adoption owing to their simplicity but the heuristics employed to act as a proxy of mixing of the underlying Markov Chains, however, invalidate distributional guarantees [27]. Approaches based on interval propagation and belief networks[14, 18, 21], also often lead to scalable techniques, but their distributions can often be far from the desired distribution.[28]. While hashing-based techniques have achieved significant progress in the context of uniform sampling [6, 7, 42], their scalability for arbitrary distributions is severely limited [4, 5, 17, 42]. In this context, we advocate the usage of knowledge compilation-based techniques, which not only achieve significant scalability but supports efficient adaptive sampling.

## 6 CONCLUSION

Design of test generation techniques to achieve higher $t-$wise coverage is a fundamental challenge in the context of testing of configurable systems. In this work, we propose a Baital approach for construction of test sample sets that dynamically modifies the probability distribution of configuration to be chosen in order to improve the t-wise coverage of the test set. We showed on a large set of benchmarks used in prior works that our approach generates achieves higher t-wise coverage than commonly used uniform sampling. We explored several ways to update the weight function and their effect on the resulted coverage.

Since our approach generates samples for combinatorial interaction testing, an interesting direction of future work would be to study the impact of improved t-wise coverage on fault detection.

## REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 421–432.

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated

software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[3] Mihir Bellare, Oded Goldreich, and Erez Petrank. 2000. Uniform Generation of NP-witnesses using an NP-oracle. *Information and Computation* 163, 2 (2000), 510–526.

[4] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-Aware Sampling and Weighted Model Counting for SAT. In *Proc. of AAAI*. 1722–1730.

[5] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Proc. of TACAS*. 304–319.

[6] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *Proc. of CAV*. 608–623.

[7] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2014. Balancing Scalability and Uniformity in SAT Witness Generator. In *Proc. of DAC*. 1–6.

[8] Nie Changhai, Xu Baowen, Shi Liang, and Wang Ziyuan. 2006. A new heuristic for test suite generation for pair-wise testing. *SEKE 2006* 1 (2006), 517.

[9] Tsong Yueh Chen, Hing Leung, and IK Mak. 2004. Adaptive random testing. In *Annual Asian Computing Science Conference*. Springer, 320–329.

[10] Anastasia Cmyrev and Ralf Reissing. 2014. Efficient and effective testing of automotive software product lines. *King Mongkuts University of Technology North Bangkok International Journal of Applied Science and Technology* 7, 2 (2014), 53–57.

[11] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2006. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*. 53–63.

[12] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264.

[13] Alexis de Colnet and Kuldeep S Meel. 2019. Dual Hashing-Based Algorithms for Discrete Integration. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 161–176.

[14] Rina Dechter, K. Kask, E. Bin, and R. Emek. 2002. Generating random solutions for constraint satisfaction problems. In *Proc. of AAAI*. 15–21.

[15] Joe W Duran and Simeon C Ntafos. 1984. An evaluation of random testing. *IEEE transactions on Software Engineering* 4 (1984), 438–444.

[16] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 549–559.

[17] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2013. Embed and Project: Discrete Sampling with Universal Hashing. In *Proc. of NIPS*. 2085–2093.

[18] Vibhav Gogate and Rina Dechter. 2006. A new algorithm for sampling CSP solutions uniformly at random. In *CP*. Springer, 711–715.

[19] Loreto Gonzalez-Hernandez, Nelson Rangel-Valdez, and Jose Torres-Jimenez. 2010. Construction of mixed covering arrays of variable strength using a tabu search approach. In *International Conference on Combinatorial Optimization and Applications*. Springer, 51–64.

[20] Rahul Gupta, Shubham Sharma, Subhajit Roy, and Kuldeep S. Meel. 2019. WAPS: Weighted and Projected Sampling. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[21] Mahesh A. Iyer. 2003. RACE: A word-level ATPG-based constraints solver system for smart random simulation. In *Proc. of ITC*. 299–308.

[22] Mark R. Jerrum and Alistair Sinclair. 1996. The Markov chain Monte Carlo method: an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems* (1996), 482–520.

[23] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science* 43, 2-3 (1986), 169–188.

[24] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 46–55.

[25] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1084–1094.

[26] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.

[27] Nathan Kitchen. 2010. *Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation.* Ph.D. Dissertation. University of California, Berkeley.

[28] Nathan Kitchen and Andreas Kuehlmann. 2007. Stimulus generation for constrained random simulation. In *Proc. of ICCAD*. 258–265.

[29] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 291–302.

[30] D Richard Kuhn, Raghu N Kacker, and Yu Lei. 2010. Practical combinatorial testing. *NIST special Publication* 800, 142 (2010), 142.

[31] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler.. In *IJCAI*. 667–673.

[32] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, 549–556.

[33] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability* 18, 3 (2008), 125–148.

[34] Yu Lei and Kuo-Chung Tai. 1998. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*. IEEE, 254–261.

[35] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. Sat-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, 91–100.

[36] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. 2015. TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 494–505.

[37] Roberto E Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba. 2013. Multi-objective optimal test suite computation for software product line pairwise testing. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 404–407.

[38] Neal Madras. 2002. Lectures on Monte Carlo Methods, Fields Institute Monographs 16. *American Mathematical Society* (2002).

[39] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. 2013. Practical pairwise testing for software product lines. In *Proceedings of the 17th international software product line conference*. 227–235.

[40] James D McCaffrey. 2009. Generation of pairwise test sets using a genetic algorithm. In *2009 33rd annual IEEE international computer software and applications conference*, Vol. 1. IEEE, 626–631.

[41] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 643–654.

[42] Kuldeep S. Meel. 2014. *Sampling Techniques for Boolean Satisfiability*. Rice University. M.S. Thesis.

[43] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functionsâĂŤI. *Mathematical programming* 14, 1 (1978), 265–294.

[44] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 1–29.

[45] Kari J Nurmela. 2004. Upper bounds for covering arrays by tabu search. *Discrete applied mathematics* 138, 1-2 (2004), 143–152.

[46] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 61–71.

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

[48] Jeho Oh, Paul Gazzillo, and Don S. Batory. 2019. *t*-wise coverage by uniform sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*. 15:1–15:4.

[49] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and scalable t-wise test case generation strategies for software product lines. In *2010 Third international conference on software testing, verification and validation*. IEEE, 459–468.

[50] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. [n.d.]. Product Sampling for Product Lines: The Scalability Challenge. ([n. d.]).

[51] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform sampling of sat solutions for configurable systems: Are we there yet?. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 240–251.

[52] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 342–352.

[53] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge Compilation meets Uniform Sampling. In *Proc. of LPAR-22*. 620–636.

[54] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 284–294.

[55] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 167–177.

[56] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. 1–5.

[57] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 1–45.

[58] Ziyuan Wang, Baowen Xu, and Changhai Nie. 2008. Greedy heuristic algorithms to generate variable strength combinatorial test suite. In *2008 The Eighth International Conference on Quality Software*. IEEE, 155–160.

[59] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. 2016. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 614–624.