# A Scalable t-wise Coverage Estimator

Eduard Baranov
Universite catholique de Louvain
Belgium
eduard.baranov@uclouvain.be

Sourav Chakraborty
Indian Statistical Institute
India
sourav@isical.ac.in

Axel Legay
Universite catholique de Louvain
Belgium
axel.legay@uclouvain.be

Kuldeep S. Meel
National University of Singapore
Singapore
meel@comp.nus.edu.sg

Vinodchandran N. Variyam
University of Nebraska-Lincoln
USA
vinod@unl.edu

## ABSTRACT

Owing to the pervasiveness of software in our modern lives, software systems have evolved to be highly configurable. Combinatorial testing has emerged as a dominant paradigm for testing highly configurable systems. Often constraints are employed to define the environments where a given system under test (SUT) is expected to work. Therefore, there has been a sustained interest in designing constraint-based test suite generation techniques. A significant goal of test suite generation techniques is to achieve $t$-wise coverage for higher values of $t$. Therefore, designing scalable techniques that can estimate $t$-wise coverage for a given set of tests and/or the estimation of maximum achievable $t$-wise coverage under a given set of constraints is of crucial importance. The existing estimation techniques face significant scalability hurdles.

The primary scientific contribution of this work is the design of scalable algorithms with mathematical guarantees to estimate (i) $t$-wise coverage for a given set of tests, and (ii) maximum $t$-wise coverage for a given set of constraints. In particular, we design a scalable framework ApproxCov that takes in a test set $\mathcal{U}$, a coverage parameter $t$, a tolerance parameter $\varepsilon$, and a confidence parameter $\delta$, and returns an estimate of the $t$-wise coverage of $\mathcal{U}$ that is guaranteed to be within $(1 \pm \varepsilon)$-factor of the ground truth with probability at least $1 - \delta$. We design a scalable framework ApproxMaxCov that, for a given formula $\mathsf{F}$, a coverage parameter $t$, a tolerance parameter $\varepsilon$, and a confidence parameter $\delta$, outputs an approximation which is guaranteed to be within $(1 \pm \varepsilon)$ factor of the maximum achievable $t$-wise coverage under $\mathsf{F}$, with probability $\geq 1 - \delta$. Our comprehensive evaluation demonstrates that ApproxCov and ApproxMaxCov can handle benchmarks that are beyond the reach of current state-of-the-art approaches. We believe that the availability of ApproxCov and ApproxMaxCov will enable test suite designers to evaluate the effectiveness of their generators and thereby significantly impact the development of combinatorial testing techniques.

*ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510218

## KEYWORDS

Configurable software, $t$-wise coverage, Approximation

**ACM Reference Format:**
Eduard Baranov, Sourav Chakraborty, Axel Legay, Kuldeep S. Meel, and Vinodchandran N. Variyam. 2022. A Scalable t-wise Coverage Estimator. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510218

## 1 INTRODUCTION

For the past 50 years, software systems have permeated nearly all aspects of our lives, including critical domains such as autonomous driving, criminal sentencing, surveillance, and healthcare. Given the diversity of application scenarios, a software system is designed to be highly configurable to allow widespread adoption [50]. Economic factors also support the design of highly configurable systems. It is desirable for a software vendor to develop a general-purpose software with a large number of configurations to allow client-level customization without necessarily requiring redesign of the underlying software architecture. At the same time, given the usage of software in critical domains such as healthcare, software failures can have serious adverse effects. Therefore, the testing of software systems is of paramount interest.

A configuration of a system refers to the assignment of values to all the configurable features of the system. Configurability does not come without a price: feature dependencies are common [54] and could lead to variability bugs appearing only in some configurations. A straightforward testing strategy would be to check whether the system behaves as intended for every possible configuration. However, such an approach is not practical for real-life systems [33] as it is common to have thousands of features in modern software systems resulting in a prohibitively large number of possible configurations [4, 5]. The combinatorial explosion of the possible set of configurations is perhaps best illustrated by the observation that embedded Linux kernel for micro-controllers has over $7.7 \times 10^{417}$ configurations [52].

The curse of configuration explosion has been well known for over three decades, and consequently, the area of *combinatorial testing* has emerged as the dominant paradigm for testing of highly configurable systems [15, 27, 34, 40, 43, 47, 62, 63, 68]. The development of combinatorial testing techniques, in large part, has been motivated by the observation that for most systems, interactions among a small number of features are sufficient to trigger the buggy behavior. An influential study by NIST observed that up to 6−wise, interactions among parameters are responsible for most of the bugs [33]. Another study showed that discovered variability bugs in Linux kernel involve up to 5−wise feature interactions [1]. In combinatorial testing, we are often interested in maximizing *$t$-wise coverage*[1], which measures the fraction of $t$-sized combinations of features appearing in the test suite over all possible $t$-sized combinations of features. A test suite that can achieve $t$-wise coverage of 1 is also called a $t$-covering array in the literature, i.e., for $n$ binary features, a $t$-covering array $\mathcal{U}$ has all the $\binom{n}{t}2^t$ combinations appearing in itself.

The complexity of test suite design is exacerbated by the observation that not every configuration is typically allowed by the system, and often constraints are employed to describe the valid set of configurations. In the real world, these constraints represent scenarios for which the systems are expected to work correctly. For example, one of the widely used SPLC challenge benchmark uLinux consists of constraints to capture the variability model in KConfig [52]. Therefore, given a set of constraints, the possible $t$-wise combinations of features are defined only over the satisfying assignments of these constraints. At this point, it is perhaps worth emphasizing that even for the case when there are no constraints, the size of a $t$-covering array for $n$ features is $\Omega(2^t \log n)$ [55]. The presence of constraints brings additional complications to the design of $t$-covering array.

Given the practical importance of the combinatorial testing, the problem of efficient design of the test suite has witnessed a sustained interest for over three decades, evidenced by the diverse set of techniques ranging from evolutionary algorithms [6, 14, 37] to constraint-based systems [12, 17] proposed over the years. The proposal of these techniques is often accompanied by measurement of $t$-wise coverage over benchmarks with small $n$ as the computation of $t$-coverage for large values of $n$ is considered impractical. Given a test suite $\mathcal{U}$ and a set of constraints $\mathsf{F}$ over the parameters, the estimation of $t$-wise coverage requires us to estimate the number of $t$-combinations of features appearing in the test suite $\mathcal{U}$ and the number of possible $t$-combinations over the solutions of $\mathsf{F}$. For the former computation, the state-of-the-art techniques maintain a hash map of size $O(\binom{n}{t}2^t)$, and the map is updated for every element of $\mathcal{U}$. Furthermore, to compute the possible $t$-combinations over the solutions of $\mathsf{F}$, the best-known algorithms check whether $\mathsf{F}$ conjuncted with a $t$-combination (expressed as a conjunction of literals) is satisfiable[2]. Unfortunately, for most practical instances, the computation of both the quantities is beyond the reach of the state-of-the-art techniques. The limitations of the current techniques for $t$-wise coverage estimations are illustrated in reliance on small benchmarks or extremely small-sized test suites whenever comparisons across different test suite generation methodologies are presented. In this context, we ask: *Can we design scalable algorithms to closely estimate $t$-wise coverage with rigorous guarantees?*

## 1.1 Our Contribution

The primary contribution of our work is an affirmative answer to the above question. We design, to the best of our knowledge, the first scalable technique that provides rigorous estimates of $t$-wise coverage. In particular, we present:

(1) A scalable Monte Carlo-based algorithm ApproxCov that takes in a test suite $\mathcal{U}$, a coverage parameter $t$, a tolerance parameter $\varepsilon$, a confidence parameter $\delta$ as input, and returns an estimate of $|\mathsf{Cov_t}(\mathcal{U})|$ that is mathematically guaranteed to be within $(1\pm\varepsilon)$-factor of the ground truth with probability at least $1-\delta$, where $\mathsf{Cov_t}(\mathcal{U})$ is the set of all $t$-wise coverages of elements in $\mathcal{U}$ and $|\cdot|$ is a set cardinality. ApproxCov takes only $O(2^t \cdot t \log n)$ (for a constant $\varepsilon$ and $\delta$) space in contrast to the space requirement of $O(\binom{n}{t}2^t)$ for the existing techniques. Therefore, for small $t < 6$, we achieve a reduction from $O(n^t)$ to $O(t \log n)$. The running time of the algorithm is also scalable for instances with parameters that are currently used in practice.
(2) A scalable counting-based algorithm ApproxMaxCov that takes in formula $\mathsf{F}$, a coverage parameter $t$, a tolerance parameter $\varepsilon$, and a confidence parameter $\delta$ as input, and returns an estimate of $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ that is guaranteed to be within $(1 \pm \varepsilon)$-factor of the ground truth with probability at least $1-\delta$. ApproxMaxCov reduces computation of $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ to the problem of projected model counting. Our reduction allows us to build on the recent advances in hashing-based paradigm for projected model counting [9, 11, 24, 59, 60], and we employ state-of-the-art hashing-based approximate model counter ApproxMC4 [58].
(3) We demonstrate the effectiveness of ApproxCov and ApproxMaxCov via implementations in Python[3] and a comprehensive experimental study. We observe that while the current state of the art techniques fail to compute coverage for beyond $t = 2$, ApproxCov and ApproxMaxCov can efficiently handle $t \in \{2, 3, 4, 5, 6\}$ (and beyond $t = 6$). Furthermore, for $t = 2$ on feature models with thousands of features, we observe significant runtime improvement: in particular, ApproxCov achieves from 2 to 136 factor speedup over prior state of the art BLMCov, while ApproxMaxCov achieves from 6 to 86 factor speedup over prior state of the art technique BLMMaxCov.
(4) We show generalizations of ApproxCov and ApproxMaxCov to estimate $t$-wise coverage on configurable systems where each feature can take values from a discrete domain. We demonstrate with experimental evaluation that the generalized algorithms are effective and can provide close estimation of $t$-wise coverage.

Few words are in order to explain the critical enabler for the scalability of ApproxCov and ApproxMaxCov: From our viewpoint, the scalability of ApproxCov owes to the simple but creative usage of the Monte Carlo-based strategy, while for ApproxMaxCov, the reduction to projected model counting allows us to employ and reap the benefits of the recent progress in the development of hashing-based techniques [9, 11, 58, 59].

**Significance of our contribution:** Combinatorial testing is a dominant testing methodology in large and complex software systems. Therefore it is vital to have tools that allow us to compare different test suite generation techniques. Our algorithms ApproxCov

[1]defined formally in Section 2
[2]An alternate approach would be to enumerate all the solutions of $F$, but for most formulas of interest, the number of solutions is too large to enumerate.

[3]https://github.com/meelgroup/approxcov

and ApproxMaxCov provide the combinatorial testing community sound tools to compare different test suite generation techniques for large benchmarks. As an immediate impact, we expect our techniques to allow a thorough comparison of the recently proposed techniques in CIT community [3, 39] for large instances, which was not feasible with the state of the art. The fact that our algorithms are grounded on guarantees that are mathematically proved makes them an attractive toolset in downstream applications.

**Organization:** The rest of the paper is organized as follows. We present notation and preliminaries in Section 2. We present related work in Section 3. We then present the technical contribution of the paper in Section 4: ApproxCov in Section 4.1 and ApproxMaxCov in Section 4.2. In Section 5 we present experimental results over a comprehensive set of benchmarks. Finally, in Section 6, we give concluding remarks.

## 2 NOTATIONS AND PRELIMINARIES

### 2.1 Boolean Formulas

A literal is a Boolean variable or its negation. A clause is a disjunction of a set of literals. A propositional formula $\mathsf{F}$ in conjunctive normal form (CNF) is a conjunction of clauses. $\mathsf{Vars(F)}$ denotes the set of variables appearing in $\mathsf{F}$. The $\mathsf{Vars(F)}$ is also called the *support* of $\mathsf{F}$. A *satisfying assignment* or *witness* of $\mathsf{F}$, denoted by $\sigma$, is an assignment of truth values to variables in its support such that $\mathsf{F}$ evaluates to true. We often represent an assignment by the set of literals that make the variables true. That is, an assignment of *True* to variable $x$ is represented as $x$ and assignment of *False* to $x$ is represented as $\neg x$. We also use the binary bit 1 (0) to represent *True* (respectively, *False*) and binary strings to represent an assignment. We denote the set of all satisfying assignments of $\mathsf{F}$ as $\mathsf{Sol(F)}$. Given a set of variables $S \subseteq \mathsf{Vars(F)}$, we use $\mathsf{Sol(F)}_{\downarrow S}$ to denote the projection of $\mathsf{Sol(F)}$ on $S$.

*Example.* Consider the formula $\mathsf{F}$ over 4 variables $\{x_1, x_2, x_3, x_4\}$ given by:

$$\mathsf{F} = (x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_4).$$

$\mathsf{Sol(F)} = \{0010, 0011, 1000, 1100\}$. In the literal representation the assignment 0010 is represented as $\{\neg x_1, \neg x_2, x_3, \neg x_4\}$. Let $S = \{x_1, x_2\}$, then $|\mathsf{Sol(F)}_{\downarrow S}| = \{00, 10, 11\}$.

The *propositional model counting problem* is to compute $|\mathsf{Sol(F)}_{\downarrow S}|$ for a given CNF formula $\mathsf{F}$ and projection set $S \subseteq \mathsf{Vars(F)}$. A *probably approximately correct* (or *PAC*) counter for Boolean formulas is a probabilistic algorithm that takes as inputs a formula $\mathsf{F}$, a sampling set $S \subseteq \mathsf{Vars(F)}$, a tolerance parameter $\varepsilon \in (0, 1)$, and a confidence parameter $\delta \in (0, 1]$, and returns a count $c$ such that

$$\Pr\left[(1-\epsilon)|\mathsf{Sol(F)}_{\downarrow S}| \le c \le (1+\varepsilon)|\mathsf{Sol(F)}_{\downarrow S}|\right] \ge 1 - \delta.$$

### 2.2 t-wise Coverage

The formulation of combinatorial interaction testing (CIT) assigns a variable corresponding to every feature of a software system. While, in practice each feature can take a finite number of values, in the the paper we consider the binary version, where each feature can take two states: *True* or *False*. Let $X = \{x_1, \cdots, x_n\}$ be the set of all the variables (corresponding to $n$ features). Then a configuration $\sigma$ of the system can be represented as an element of the set $\prod_i \{x_i, \neg x_i\}$. For example, for $X = \{x_1, x_2, x_3\}$, $\sigma = \{x_1, \neg x_2, x_3\}$ is an example of a configuration.

Given a configuration $\sigma$ represented as a set of literals, we define the $t$-wise coverage of $\sigma$ denoted by $\mathsf{Cov_t}(\sigma) = \{T \subseteq \sigma \mid |T| = t\}$, the set of all subsets of literals of the size $t$ in $\sigma$. $\mathsf{Cov_t}(\sigma)$ represents the set of $t$-sized feature combinations due to $\sigma$. We can extend the notion of $\mathsf{Cov_t}$ to a set $\mathcal{U} \subseteq \prod_i \{x_i, \neg x_i\}$ of configurations as $\mathsf{Cov_t}(\mathcal{U}) = \bigcup_{\sigma \in \mathcal{U}} \mathsf{Cov_t}(\sigma)$. For a given $\sigma$, $|\mathsf{Cov_t}(\sigma)| = \binom{|X|}{t} = \binom{n}{t}$. However, this does not imply $|\mathsf{Cov_t}(\mathcal{U})| = |U| \times \binom{|X|}{t}$ since $|\mathsf{Cov_t}(\sigma_1) \cup \mathsf{Cov_t}(\sigma_2)|$ is not necessarily equal to $|\mathsf{Cov_t}(\sigma_1)| + |\mathsf{Cov_t}(\sigma_2)|$ due to non-empty intersection of $\mathsf{Cov_t}(\sigma_1)$ and $\mathsf{Cov_t}(\sigma_2)$. Also note that, for $\prod_i \{x_i, \neg x_i\}$, the set of all possible configurations $|\mathsf{Cov_t}(\prod_i \{x_i, \neg x_i\})| = 2^t \binom{|X|}{t}$. We will call $\mathsf{Cov_t}(\prod_i \{x_i, \neg x_i\})$ the *universe* and denote it by $\Omega$. The above discussion leads to the following observation which is crucial for the proof of correctness of our algorithm.

Observation 2.1. *For any $\mathcal{U} \neq \emptyset$ over a set of variables $X$ and any $1 \le t \le |X|$,*

$$\binom{|X|}{t} \le |\mathsf{Cov_t}(\mathcal{U})| \le 2^t \binom{|X|}{t}.$$

We will be interested in the coverage of a set of configurations that satisfy certain constraints over the features. We will focus on constraints represented by a Boolean formula $\mathsf{F}$. For a set $\mathcal{U} \subseteq \mathsf{Sol(F)}$, the $t$-wise fractional coverage of a set $\mathcal{U}$ with respect to a formula $\mathsf{F}$, denoted by $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ is defined as follows:

$$\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}) = \frac{|\mathsf{Cov_t}(\mathcal{U})|}{|\mathsf{Cov_t}(\mathsf{Sol(F)})|}$$

*Example.* To illustrate the notions, let us consider again the CNF formula $\mathsf{F}$

$$\mathsf{F} = (x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_4).$$

The following table lists $\mathsf{Cov_t}(\mathsf{Sol(F)})$ for $t = 2$. For compactness, we use the bit representation of the assignments and coverage. For example, a set of literals $\{\neg x_1, \neg x_2\}$ is listed as 00 in the column indexed $(1, 2)$. $|\mathsf{Cov_t}(\mathsf{Sol(F)})| = 17$. For $U = \{0010, 0011\}$ (shaded in the table), $|\mathsf{Cov_t}(\mathcal{U})| = 9$. $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}) = 9/17$.

| | 2-tuples | | | | | |
|---|---|---|---|---|---|---|
| Sol(F) | (1, 2) | (1, 3) | (1, 4) | (2, 3) | (2, 4) | (3, 4) |
| 0010 | 00 | 01 | 00 | 01 | 00 | 10 |
| 0011 | 00 | 01 | 01 | 01 | 01 | 11 |
| 1000 | 10 | 10 | 10 | 00 | 00 | 00 |
| 1100 | 11 | 10 | 10 | 10 | 10 | 00 |
| Total | 3 | 2 | 3 | 3 | 3 | 3 |
| For $\mathcal{U}$ | 1 | 1 | 2 | 1 | 2 | 2 |

**Table 1: Coverage for example formula F**

## 3 RELATED WORK

### 3.1 Combinatorial Testing

Since the introduction of combinatorial testing in the 1980s as an effective option for configurable systems [40, 62], steady progress has been reported on this topic. We refer the reader to [34, 47, 63] for a detailed overview of the topic. In the classical combinatorial testing, the goal is to design a test suite $\mathcal{U}$ such that $|\mathsf{Cov_t}(\mathcal{U})| = \binom{n}{t}2^t$. Such a test suit is also known as a covering array formally defined in [57]. Covering arrays are orthogonal arrays or matrices with rows representing configurations of a system and every possible t-sized combination of variables appears at least in one column. Over the decades, the construction of covering arrays has witnessed a wide variety of approaches including greedy search [2, 13, 28, 36, 37, 41, 61, 65, 67], divide-and-compose [51], genetic algorithms [42], and tabu search [25, 48].

Modern software systems have a large number of features, and the design of covering array is often impractical for $t > 2$. Furthermore, the modern softwares have associated variability models, and not every configuration is valid, and therefore, cannot act as a test. In this context, constraints are employed to capture the associated variability models or the scenarios under which a software is expected to behave as per specifications. Combinatorial testing in such a constraint setting is called constrained combinatorial testing [7]. The presence of constraints has led to the development of techniques that rely on the progress in combinatorial solvers over the past three decades. Several approaches have been proposed that seek to sample solutions subject to constraints. These approaches resulted in BDD-based techniques [35], random seeding of DPLL-based SAT solvers [46], Markov Chain Monte Carlo-based methods [29–31, 66], interval propagation and belief networks-based methods [16, 22], MaxSAT-based techniques such as Quicksampler [19], hashing-based approaches [8, 10, 58], knowledge compilation-based approaches such as KUS [56], WAPS [26], and Baital [3].

### 3.2 Model Counting

Valiant initiated the complexity theoretic study of model counting and showed that the problem of model counting for CNF formula is #P-complete [64]. The problem of projected model counting which we employ in this paper is shown to be #·NP-complete [18]. Given the computational intractability of (projected) model counting, we are often interested in $(\varepsilon, \delta)$-approximations of the exact count, where the goal is to obtain an $(1 \pm \epsilon)$ multiplicative approximation of the exact count with probability at least $(1 - \delta)$. Hashing-based techniques have emerged as a dominant paradigm seeking scalability while providing $(\varepsilon, \delta)$-approximation guarantees. The core idea of hashing-based techniques is to employ pairwise independent hash functions to partition the solution space into *roughly equal* small cells of solutions. Then, we randomly choose one of the small cells, enumerate all the solutions using a SAT solver one by one. The number of solutions is estimated to be simply the number of solutions in the cell multiplied by the total number of cells. The pairwise independent hash functions can be realized using XOR-based hash functions. The hashing-based techniques trace their origin to Stockmeyer's seminal work [60], subsequently pursued by Gomes, Sabharwal, and Selman [23].

Chakraborty, Meel, and Vardi proposed the first scalable approximate model counter, ApproxMC, which invoked the underlying SAT solver, CryptoMiniSat, $O(n)$ times (where $n$ is the number of variables in the original formula). Subsequently, Chakraborty et al [11] reduced the number of SAT calls from $O(n)$ to $O(\log n)$; the corresponding counter was called ApproxMC2. Soos and Meel [59] sought to improve the underlying SAT solver's architecture; their new architecture, called BIRD, achieved significant performance improvement. ApproxMC is currently in its fourth generation, ApproxMC5 [45, 58][4].

### 3.3 $t$-wise Coverage Estimation

While combinatorial testing has witnessed over four decades of sustained interests from theoreticians and practitioners, the techniques to estimate $|\mathsf{Cov}_{\mathcal{U}}|$ and $|\mathsf{Cov}_{\mathsf{Sol(F)}}|$ have rather been largely underexplored. Given a set $\mathcal{U}$, the state of the art technique maintains a map of the $t$-wise combinations seen in $\mathcal{U}$. In the case of a given formula F, observe that every $t$-combination can be expressed as a conjunction of $t$ literals, say $\pi$. Given such a $\pi$, the state of the art techniques simply invoke a SAT solver to check whether $\mathsf{F} \wedge \pi$ is satisfiable. In our work, we use the implementation of these techniques due to Baranov, Legay, and Meel [3], which was used in the evaluation of the current state of the art test generator suite, Baital[5]. We use BLMCov and BLMMaxCov to denote the implementations for computations of $\mathsf{Cov_t}(\mathcal{U})$ and $\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))$ respectively. To the best of our knowledge, BLMCov and BLMMaxCov represent the current state of the art; an assertion confirmed by the authors of both the recently published studies in SE community [3, 39].

## 4 ALGORITHMS

In this section we present two algorithms. The first algorithm ApproxCov (presented in Section 4.1) estimates $|\mathsf{Cov_t}(\mathcal{U})|$. More precisely ApproxCov takes as input a set $\mathcal{U} \subset \prod_i\{x_i, \neg x_i\}$, error parameter $0 < \epsilon < 1$, and a confidence parameter $0 < \delta < 1$, and outputs a number that, with probability at least $(1 - \delta)$, is between $(1 - \epsilon)|\mathsf{Cov_t}(\mathcal{U})|$ and $(1 + \epsilon)|\mathsf{Cov_t}(\mathcal{U})|$.

In Section 4.2 we present another algorithm ApproxMaxCov that given a Boolean formula F, estimates $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$. More precisely, ApproxMaxCov takes as input F, an error parameter $0 < \epsilon < 1$ and a confidence parameter $0 < \delta < 1$ and outputs a number that, with probability at least $(1 - \delta)$, is between $(1 - \epsilon)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ and $(1 + \epsilon)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$.

Using ApproxCov and ApproxMaxCov we can estimate the value of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$. Given a Boolean formula F and a set $\mathcal{U} \subset \mathsf{Sol}(\mathsf{F})$, say we use ApproxCov$(\mathcal{U}, \epsilon_1, \delta_1)$ and ApproxMaxCov$(\mathsf{F}, \epsilon_2, \delta_2)$ to obtain estimates for $|\mathsf{Cov_t}(\mathcal{U})|$ and $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ respectively. Let $\mathsf{Out}_1$ be the output of ApproxCov$(\mathcal{U}, \epsilon_1, \delta_1)$ and $\mathsf{Out}_2$ be the output of ApproxMaxCov$(\mathsf{F}, \epsilon_2, \delta_2)$. So we have with probability at least $(1 - \delta_1)$ the following Equation 1 and with probability $(1 - \delta_2)$ the Equation 2 holds.

$$(1 - \epsilon_1)|\mathsf{Cov_t}(\mathcal{U})| \le \mathsf{Out}_1 \le (1 + \epsilon_1)|\mathsf{Cov_t}(\mathcal{U})|, \quad (1)$$

$$(1 - \epsilon_2)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))| \le \mathsf{Out}_2 \le (1 + \epsilon_2)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|. \quad (2)$$

[4]While beta version of ApproxMC5 is released; ApproxMC4's developer recommend usage of ApproxMC4

[5]We use the implementation available at https://github.com/meelgroup/baital

So, by union bound, with probability at least $(1-\delta_1-\delta_2)$ both the above equations work. Thus with probability $(1-\delta_1-\delta_2)$ we have

$$\frac{(1-\epsilon_1)}{(1+\epsilon_2)}\frac{|\mathsf{Cov_t}(\mathcal{U})|}{|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|} \le \frac{\mathsf{Out}_1}{\mathsf{Out}_2} \le \frac{(1+\epsilon_1)}{(1-\epsilon_2)}\frac{|\mathsf{Cov_t}(\mathcal{U})|}{|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|}. \quad (3)$$

Thus given a Boolean formula F, a set $\mathcal{U} \subset \mathsf{Sol}(\mathsf{F})$, an error parameter $\epsilon$ and a confidence parameter $\delta$, if we set $\epsilon_1, \epsilon_2, \delta_1$ and $\delta_2$ appropriately we will be able to give an $(1 \pm \epsilon)$-multiplicative estimate of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ with probability $(1-\delta)$. For example, let us set $\epsilon_1 = \epsilon_2 = \frac{\epsilon}{2+\epsilon}$ and $\delta_1 = \delta_2 = \delta/2$. Note that in that case $(1+\epsilon_1)/(1-\epsilon_2)$ is at most $(1+\epsilon)$ and $(1-\epsilon_1)/(1+\epsilon_2)$ is at least $(1-\epsilon)$. Then by using $\mathsf{ApproxCov}(\mathcal{U}, \epsilon_1, \delta_1)$ and $\mathsf{ApproxMaxCov}(\mathsf{F}, \epsilon_2, \delta_2)$ to estimate $|\mathsf{Cov_t}(\mathcal{U})|$ and $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ respectively, from Equation 3, we have with probability at least $(1-\delta)$

$$(1-\epsilon)\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}) \le \frac{\mathsf{Out}_1}{\mathsf{Out}_2} \le (1+\epsilon)\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}).$$

## 4.1 Counting the Coverage of a Test Suit

The intuition behind the ApproxCov is similar to the Monte-Carlo method. Given a set of variables $X = \{x_1, \ldots, x_n\}$, consider a test set $\mathcal{U} \subseteq \prod_i \{x_i, \neg x_i\}$ and the universe set $\Omega := \mathsf{Cov_t}(\prod_i \{x_i, \neg x_i\})$ with all possible $t$-wise coverage tuples (the set with all combinations of size $t$ that can be obtained with variables from $X$). The algorithm picks a set $\mathcal{S}$ of size $\ell$ of random $t$-wise coverage tuples from $\Omega$ ($\ell$ is appropriately chosen based on the input parameters). Then it counts the number of elements from $\mathcal{S}$ that is realizable by at least one of the tests in $\mathcal{U}$ (membership in $\mathsf{Cov_t}(\mathcal{U})$). Let this number be $m$. Then the output of the algorithm is $\frac{m}{\ell} \cdot |\Omega|$.

In detail, ApproxCov shown in Algorithm 1 starts by setting the size of sample set that it would be picking depending on the parameters $\varepsilon$ and $\delta$. In the **For** loop between line 3 and line 7 we pick $\ell$ samples uniformly at random from $\Omega$. The sampling is done in two steps. For each sample, at first we select uniformly $t$ variables to be used in a sample at line 4. At the second step we select a value for each variable at random at line 5. Note that this procedure gives a random element of $\Omega$ as the universe set can also be written as

$$\Omega = \left\{ w \in \prod_{i \in T} \{x_i, \neg x_i\} \mid T \subseteq \binom{[n]}{t} \right\},$$

where $\binom{[n]}{t}$ is the set of all subsets of the set $\{1, \ldots, n\}$ of size $t$.

Samples are stored in a map $\mathcal{M}$, where the values indicate whether the sample is realizable by at least one test in $\mathcal{U}$, initialized to 0. Note that there may be some elements that are picked multiple times. In that case we will keep all the copies in the map $\mathcal{M}$, in other words, $\mathcal{M}$ is actually a multi-map. In the nested **for** loops between line 8 and line 14 we update values of the map $\mathcal{M}$ if a sampled element is a subset of any of the $\sigma \in \mathcal{U}$. Finally, in line 15 we sum the variables corresponding to the elements in $\mathcal{M}$ and output the value multiplied with an appropriate scaling number.

Theorem 4.1. *Let $X = \{x_1, \ldots, x_n\}$ be the set of $n$ variables and let $\mathcal{U}$ be any subset of $\prod_i \{x_i, \neg x_i\}$. Then for any positive integer $t$ and positive real numbers $0 < \epsilon, \delta < 1$, with probability at least $(1-\delta)$ the output of* ApproxCov *is an $(1 \pm \epsilon)$-multiplicative approximation of the $|\mathsf{Cov_t}(\mathcal{U})|$. That is, with probability at least $(1-\delta)$,*

$$(1-\epsilon)|\mathsf{Cov_t}(\mathcal{U})| \le \frac{\binom{n}{t}2^t}{\ell}\sum_{k=1}^{\ell} \mathcal{M}[w_k] \le (1+\epsilon)|\mathsf{Cov_t}(\mathcal{U})|.$$

*Moreover, the amount of space needed is $O\left(\left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta)\right\rceil t\lceil \log_2 n\rceil\right)$ and the run time is $O\left(\left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta)\right\rceil t\lceil \log_2 n\rceil|\mathcal{U}|\right)$.*

**Algorithm 1** ApproxCov($\mathcal{U}, \epsilon, \delta$)

```
1:  $\ell \leftarrow \left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta)\right\rceil$
2:  Initialise $\mathcal{M} = \emptyset$
3:  for $k = 1; k \le \ell; k++$ do
4:      Pick a random $T_k$ from $\binom{[n]}{t}$
5:      Pick a random $w_k$ from $\prod_{i \in T_k} \{x_i, \neg x_i\}$
6:      Put $w_k \rightarrow 0$ into $\mathcal{M}$
7:  end for
8:  for $\sigma \in \mathcal{U}$ do
9:      for $k = 1; k \le \ell; k++$ do
10:         if $w_k \subseteq \sigma$ then
11:             Update $\mathcal{M}[w_k]$ to 1
12:         end if
13:     end for
14: end for
15: Output $\frac{\binom{n}{t}2^t}{\ell}\sum_{k=1}^{\ell}\mathcal{M}[w_k]$
```

## 4.2 Counting the Coverage with Constraints

In this section we present the algorithm ApproxMaxCov for estimating $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$. Notice that it is not straightforward to use ApproxCov to estimate this quantity as Sol(F) is not given explicitly. Hence we design a new algorithm ApproxMaxCov that uses a projected model counting algorithm on a related formula. It is a two-step algorithm shown in Algorithm 2. For a Boolean formula F on variable set $X$, it will first construct a new Boolean formula $\mathsf{G^F}$ on variable set $X \cup S$, where $S$ is an additional set of variables, such that

$$|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))| = |\mathsf{Sol}(\mathsf{G}^{\mathsf{F}}_{\downarrow\{\mathsf{S}\}})| \quad (4)$$

Then it will use an approximate projected model counting algorithm (which we call ApproxCount) on $\mathsf{G^F}$ to output an $(\epsilon, \delta)$ estimate of $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$. Several algorithms are known for the approximate model counting problem and we discuss the one we use in Section 5. Since the output of ApproxCount is guaranteed to be within $(1 \pm \varepsilon)$ of $|\mathsf{Sol}(\mathsf{G}^{\mathsf{F}}_{\downarrow\{\mathsf{S}\}})|$ with probability at least $(1-\delta)$, the output of ApproxMaxCov is also between $(1-\epsilon)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ and $(1+\epsilon)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$, with probability at least $(1-\delta)$.

**Algorithm 2** ApproxMaxCov($F, t, \varepsilon, \delta$)

```
1: $(\mathsf{G^F}, S) \leftarrow \mathsf{ConstructGFormula}(F)$
2: $c \leftarrow \mathsf{ApproxCount}(\mathsf{G^F}, S, \varepsilon, \delta)$
3: return $c$
```

Thus the correctness of the algorithm ApproxMaxCov follows once we show that the formula $\mathsf{G^F}(X, S)$ satisfies the condition in Equation 4.

Lemma 4.2.

$$|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))| = |\mathsf{Sol}(\mathsf{G}^{\mathsf{F}}_{\downarrow\{S\}})|$$

We now present the main idea of the construction of $\mathsf{G}^{\mathsf{F}}$.

**Construction of $\mathsf{G}^{\mathsf{F}}$:** Intuitively, $\mathsf{G}^{\mathsf{F}}$ encodes F and all the $t$-wise coverage tuples contained in Sol(F). Given natural numbers $n$ and $t$, and a Boolean formula F on $n$ variables $X = \{x_0, \dots, x_{n-1}\}$[6] we will define a new Boolean formula $\mathsf{G}^{\mathsf{F}}$ on $n + t\lceil \log_2 n \rceil + t$ variables.

First $n$ variables are the variables of $F$. The remaining set of variables $S$ can be partitioned into $t+1$ groups: $Y_1, \dots, Y_t$ of size $\lceil \log_2 n \rceil$ and $Z = \{z_1, \dots, z_t\}$. Intuitively, $S$ encodes a subset of indices $\{1, \cdots, n\}$ of size $t$: $Y_i$ is a bit-vector that encodes the $i^{th}$ integer in the subset and $z_i$ is its value in $X$. To ensure that only elements of $\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))$ can be assigned to the set $S$, we extend the formula F with additional constraints that is encoded in a formula $F'$ and the formula $\mathsf{G}^{\mathsf{F}}$ is constructed as a conjunction of $F$ and $F'$. For a binary string $y_i$ of length $\lceil \log_2 n \rceil$, let $\mathsf{val}(y_i)$ denote the number encoded by a binary assignment to the variables in $Y$. $F'$ is a formula over variable $Y \cup Z$ that encodes the following requirements.

(1) $0 \le \mathsf{val}(y_i) < n$ for all $1 \le i \le t$.
(2) $\mathsf{val}(y_i) < \mathsf{val}(y_{i+1})$ for all $1 \le i \le t-1$. This constraint together with (1) ensures that the tuple $\langle y_1, \cdots, y_t \rangle$ encodes a subset of indices of size $t$ (same index cannot appear in a combination multiple times and each set can have only one representation).
(3) $x_{\mathsf{val}(y_i)} = z_i$ for all $1 \le i \le t$. This constraint ensures that the binary assignment variables $Y \cup Z$ encodes a $t$-wise coverage tuple contained in the assignment to F.

With this intuition it is easy to see that the number of different valid assignments to the set $S = Y \cup Z$ is the desired value $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$.

## 5 EXPERIMENTS

In this section we evaluate the precision and efficiency of algorithms ApproxCov and ApproxMaxCov. Both algorithms have been implemented in Python 3. ApproxMaxCov implementation uses a state-of-the-art tool ApproxMC4 [11, 58, 59] for ApproxCount subroutine (for approximate projected model counting). In the evaluation of the algorithms, we want to validate that the implementation achieves the theoretical results on the estimation accuracy and to compare the performance of the algorithms with the existing approaches. Therefore, we pose the following research questions:

- **RQ1** and **RQ2**: Are approximations of ApproxCov and ApproxMaxCov close to the correct values and within the boundary provided by PAC guarantees?
- **RQ3** and **RQ4**: Are ApproxCov and ApproxMaxCov faster and more scalable than the existing approaches?

As discussed in Section 4, an estimate of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ for a given sample set $\mathcal{U}$ and a formula F can be computed with ApproxCov and ApproxMaxCov. Such an estimate informs us of the $t$-coverage achieved by a given test suite with respect to the maximum possible $t$-wise coverage subject to the set of constraints F. Evaluation of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ is covered by the following research question:

- **RQ5** Can algorithms ApproxCov and ApproxMaxCov be used to estimate $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ and provide a close approximation to the correct result?

As noted earlier, it is well known [3, 39] that typically techniques used for binary domains can be scalably lifted to discrete domains. We describe the extensions of ApproxCov and ApproxMaxCov for the general case along with empirical studies in the end of the section.

### 5.1 Benchmarks & Experimental Setup

For the experiments we selected a large number of publicly available feature models from real-world configurable systems that were used in the literature for the evaluation of sampling tools. In particular, we took 123 benchmarks appeared in [3, 32, 38, 52, 53]. The benchmarks have between 565 and 11254 variables, between 1164 and 62183 clauses, and between $9.7 \times 10^{13}$ and $7.7 \times 10^{417}$ solutions. Unfortunately, existing approaches are not capable to compute $|\mathsf{Cov_t}(\mathcal{U})|$ and $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ for $t \ge 3$ on large benchmarks. Therefore, in order to check approximations for $t \ge 3$ we have selected smaller benchmarks from [44]. In particular we have sorted the benchmarks by the number of variables and randomly selected 1 for each value between 10 and 500 providing 111 extra benchmarks. In the remainder of the section we would reference the first 123 benchmarks as 'large' and the last 111 as 'small'.

The evaluation of ApproxCov requires sample sets, therefore we generated them with 3 publicly available tools —WAPS [26], Quicksampler [20], and Baital [3] —for each of the 234 benchmarks. Each sample set contains 1000 samples. Since Quicksampler can generate unsatisfiable samples that are filtered afterwards, we generated more samples with this tool and selected the first 1000 valid ones. Note that for several benchmarks, even after requesting 100000 samples, Quicksampler output contained less than 1000 valid samples, therefore we did not consider such sample sets in the experiments. Thus, we used 674 sample sets for the experiment. Similarly to the feature models, we would call sample sets originating from the first 123 feature models as 'large' and the remaining would be 'small'.

All experiments were conducted on a high performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2x12 real cores and 96GB of RAM.

### 5.2 Methodology

The evaluation of ApproxCov is focused on **RQ1** and **RQ3**. In our experiment we used ApproxCov to approximate $|\mathsf{Cov_t}(\mathcal{U})|$ for $t \in [2, 6]$ on 674 sample sets described above. We used $\varepsilon$ and $\delta$ equal to 0.05. ApproxCov computations have been performed 10 times without fixed random seed and in the results we report the mean running time and the worst-case output; i.e. the output farthest from the $|\mathsf{Cov_t}(\mathcal{U})|$. For comparison, we have performed the same computations with BLMCov [3]. The timeout was set to 3600 seconds and the memory limit was set to 4Gb for both the tools.

The evaluation of ApproxMaxCov is focused on **RQ2** and **RQ4**. In our experiments we approximated $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ on 234 benchmarks for $t \in [2, 6]$. We used $\delta = \varepsilon = 0.05$. Similarly to the previous experiment, all computations with ApproxMaxCov have been performed 10 times without fixed random seed. For comparison, we have performed the same computations with BLMMaxCov [3].

[6]Choosing $\{x_0, \dots, x_{n-1}\}$ instead of $\{x_1, \dots, x_n\}$ is for easiness of presentation.

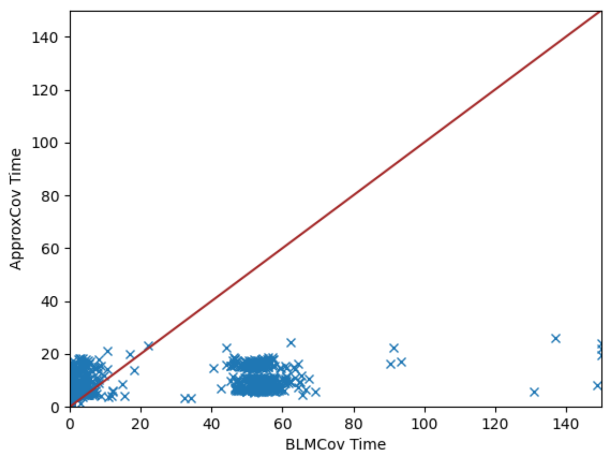

**Figure 1: Comparison of execution time of** ApproxCov **and** BLMCov **on 674 sample sets for** $t = 2$**. X axis shows** BLMCov **time in seconds, Y axis shows** ApproxCov **time in seconds. Red line indicates equal time. Among 3 points at the right border, 2 correspond to timeout of** BLMCov **and 1 corresponds to 3295 seconds for** BLMCov**.**

Originally, the timeout was set to 3600 seconds for both tools. Unfortunately, BLMMaxCov is able to compute only 2 'large' benchmarks within this timeout for $t = 2$. Therefore, we raised the timeout for BLMMaxCov to 28800 seconds. The memory limit was set to 4Gb.

For the evaluation of **RQ5** we used the results from the two previous experiments: the estimation of FracCov$_t(\mathcal{U}, F)$ is obtained by dividing the $i^{th}$ result of ApproxCov by the $i^{th}$ result of ApproxMaxCov on the corresponding feature model. The 10 generated approximations have been compared with the correct values computed with BLMCov and BLMMaxCov. Confidence parameters of FracCov$_t(\mathcal{U}, F)$ are derived from the selection of $\varepsilon$ and $\delta$ in the computation of Approx- Cov and ApproxMaxCov: $\delta = 0.0975$, $\varepsilon = 0.105$.

## 5.3 Results for ApproxCov

In the first experiment we computed $|\text{Cov}_t(\mathcal{U})|$ with ApproxCov and BLMCov. For $t = 2$, ApproxCov have successfully terminated on all sample sets within 25 seconds. BLMCov has timed out on 2 benchmarks. Comparing the computation time for $t = 2$ between BLMCov and ApproxCov, BLMCov was faster on sample sets with few variables. Among large sample sets with more than 500 variables, only 3 were faster with BLMCov, while on the rest of the sets ApproxCov was from 2 to 136 times faster. Time comparison is shown Figure 1; among 3 points on the right border, 2 correspond to timeouts of BLMCov and the remaining corresponds to 3295 seconds for BLMCov. Slower performance on smaller benchmarks is explained by the fact that the number of picked elements depends on the selected $\varepsilon$ and $\delta$ parameters rather than the number of variables. Note that on the smallest benchmarks the number of combinations to pick is greater than the total number of different combinations. In such cases all combinations are selected and the result is an exact value rather than approximation.

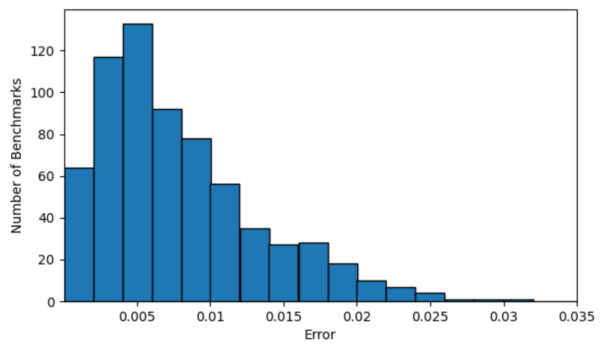

**Figure 2: Approximation error of** ApproxCov **for** $t = 2$ **on 672 sample sets computed as** $max(abs(result_i - |\text{Cov}_2(\mathcal{U})|)/|\text{Cov}_2(\mathcal{U})|)$**, where** $result_i$ **is approximation returned by** ApproxCov **on the** $i^{th}$ **run. 2 sample sets that timed out with** BLMCov **are not used in this figure.**

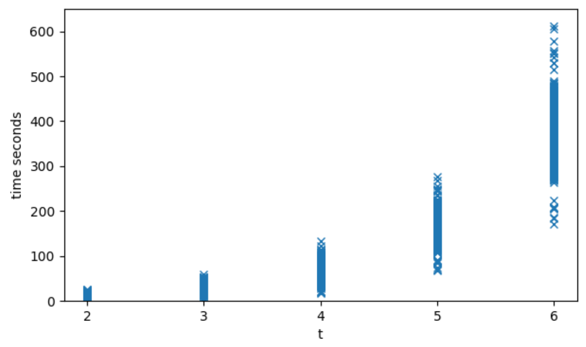

**Figure 3:** ApproxCov **execution time on 674 sample sets for** $t \in [2, 6]$**. X axis shows the value of** $t$**, Y axis shows time in seconds.**

To check the accuracy of ApproxCov approximation, we compared the approximation results with the correct value of $|\text{Cov}_2(\mathcal{U})|$ computed by BLMCov. For each sample set we took the maximal value of $abs(result - |\text{Cov}_2(\mathcal{U})|)/|\text{Cov}_2(\mathcal{U})|$ among 10 *results* of ApproxCov. Figure 2 shows the histogram of errors, the largest error was 0.0315 which is smaller than selected $\varepsilon$.

For $t \geq 3$, BLMCov has successfully terminated only on few 'small' benchmarks within the given time and memory budget: 292 benchmarks for $t = 3$, 76 for $t = 4$, 36 for $t = 5$, and 22 for $t = 6$. In comparison, ApproxCov terminated on all benchmarks within 650 seconds for all $t \leq 6$. The execution time for various values of $t$ is shown in Figure 3. The approximation accuracy was within PAC guarantees: the largest error was 0.0369. The histogram of errors on benchmarks computed by BLMCov is shown in Figure 4.

Our experiment shows that the results of the ApproxCov are close to the $|\text{Cov}_2(\mathcal{U})|$ and within the selected boundary from PAC guarantees, thus answering **RQ1**. Comparison of execution time allows us to give a positive answer to **RQ3**: ApproxCov is can compute $|\text{Cov}_t(\mathcal{U})|$ for $t = 6$, while existing method fails on half of the sample sets for $t = 3$.

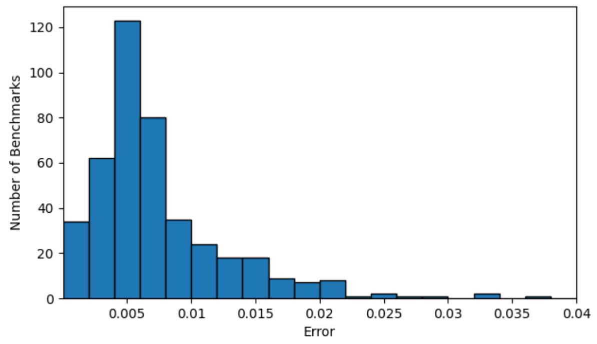

**Figure 4: Approximation error of** ApproxCov **for** $t \in [3,6]$ **on all benchmarks on which** BLMCov **has successfully terminated - 426 elements in total. The error is computed as** $max(abs(result_i - |Cov_t(\mathcal{U})|)/|Cov_t(\mathcal{U})|)$**, where** $result_i$ **is approximation returned by** ApproxCov **on the** $i^{th}$ **run.**

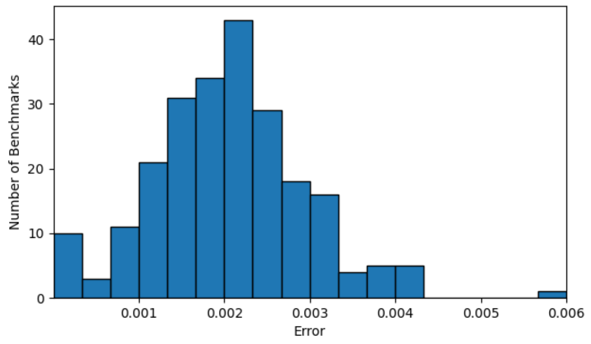

**Figure 5: Approximation error of** ApproxMaxCov **for** $t = 2$ **on 221 benchmarks on which** BLMMaxCov **terminated. The error is computed as** $max(abs(result_i - |Cov_2(Sol(F))|)/|Cov_2(Sol(F))|)$**, where** $result_i$ **the approximation returned by** ApproxMaxCov **on** $i^{th}$ **run.**

## 5.4 Results for ApproxMaxCov

ApproxMaxCov have successfully terminated on all benchmarks for $t = 2$ within 360 seconds except 1 benchmark that required 1720 seconds. For comparison, BLMMaxCov has successfully terminated only on 2 'large' benchmarks for $t = 2$ with 3600 seconds timeout. After raising timeout to 28800 seconds it succeeded on 221 benchmarks out of 234. The speed up factor of ApproxMaxCov on large benchmarks have range between 6 and 86.

To check the accuracy of ApproxMaxCov approximation, we compared it with $Cov_2(Sol(F))$. For each benchmark, we took the maximal value of $abs(result - |Cov_2(Sol(F))|)/|Cov_2(Sol(F)|$ among 10 $results$ of ApproxMaxCov. Results for 8 benchmarks are shown in Table 2. Figure 5 shows the histogram of errors, the largest error was 0.0058 which is smaller than selected $\varepsilon$.

For larger values of $t$, ApproxMaxCov timed out of 3600 seconds on the largest benchmark for $t \geq 4$ and on one more benchmark for $t = 6$. Further exploration of these 2 benchmarks showed that ApproxMaxCov can generate all results within 8500 seconds. The execution time for various values of $t$ is shown in Figure 6.

| Benchmark | Ground Truth | ApproxMaxCov | Error |
|---|---|---|---|
| buildroot | 621270 | 622592 | 0.0021 |
| busybox_1_28_0 | 1965023 | 1967616 | 0.0013 |
| ecos-icse11 | 2910229 | 2913280 | 0.0010 |
| financial | 917150 | 919040 | 0.0021 |
| mpc50 | 2719748 | 2713600 | 0.0023 |
| phycore | 3008140 | 3015680 | 0.0025 |
| psim | 2591638 | 2597888 | 0.0024 |
| sleb | 2624832 | 2630656 | 0.0022 |

**Table 2: Comparison of** ApproxMaxCov **approximations with** $|Cov_2(Sol(F))|$**. First column shows the benchmark, second and third columns shows** $|Cov_2(Sol(F))|$ **and worst** ApproxMaxCov **result from 10 runs, and the last column is computed as** $(abs(column3 - |Cov_2(Sol(F))|)/|Cov_2(Sol(F))|)$**.**

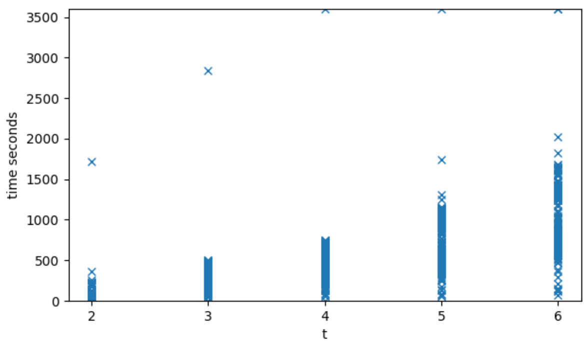

**Figure 6:** ApproxMaxCov **execution time on 234 benchmarks for** $t \in [2,6]$**. X axis shows the value of** $t$**, Y axis shows time in seconds. Points on the top border correspond to timeouts.**

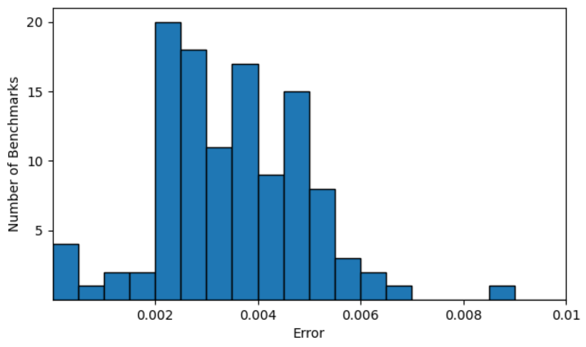

**Figure 7: Approximation error of** ApproxMaxCov **for** $t \in [3,6]$ **on all benchmarks on which** BLMMaxCov **has successfully terminated - 114 elements in total. The error is computed as** $max(abs(result_i - |Cov_t(Sol(F))|)/|Cov_t(Sol(F))|)$**, where** $result_i$ **the approximation returned by** ApproxMaxCov **on** $i^{th}$ **run.**

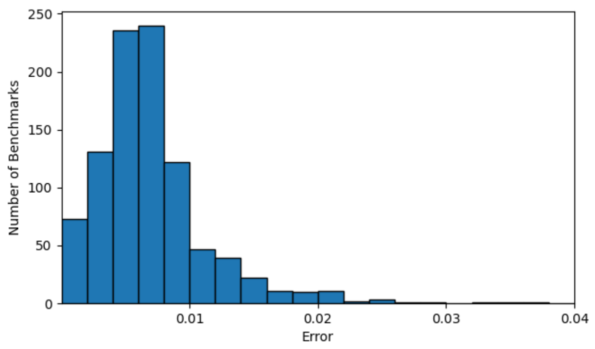

**Figure 8: Approximation error of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ for $t \in [3, 6]$ on all benchmarks on which both BLMCov and BLMMaxCov have successfully terminated - 952 elements in total. It has been computed as $max(abs(result_i - \mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}))/\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}))$, where $result_i$ is a quotient of approximations returned by ApproxCov on the $i^{th}$ run and of approximation returned by ApproxMaxCov on the $i^{th}$ run on the corresponding feature model.**

BLMMaxCov was able to compute 81 benchmarks for $t = 3$, 19 for $t = 4$, 9 for $t = 5$, and only 5 for $t = 6$. Comparison of approximation with $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ showed the largest error of 0.0086 which is within PAC guarantees. The error histogram is shown in Figure 7

The experiment shows that results computed by ApproxMaxCov are close to $\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))$ and within the selected boundary from PAC guarantees (**RQ2**). ApproxMaxCov was significantly faster than the existing methods on large benchmarks and was able to output the result on the benchmarks upto $t = 6$, where BLMMaxCov timed out on several benchmarks even with a large time budget on $t = 2$, thus allowing us to positively answer to **RQ4**.

## 5.5 Approximation of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$

The approximations of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ have been obtained by taking pairs of results of ApproxCov and ApproxMaxCov. We report the worst approximation obtained by this method. Results for 15 sample sets are shown in Table 3 and the histogram of errors is shown in Figure 8. The largest error was 0.038 which is smaller than the derived $\varepsilon$.

This result shows that the approximation of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ obtained with ApproxCov and ApproxMaxCov is close to the correct value. Moreover, considering the scalability of both approximation algorithms, the approximation can be computed on all benchmarks except 6 for $t = 6$, while existing methods succeeded only on 13, thus positively answering **RQ5**.

## 5.6 Extension to the General Case

In this section we describe how algorithms ApproxCov and ApproxMaxCov can be generalized to arbitrary alphabet size: the case where each feature can take a finite number of values. In the second part of this subsection we provide results for empirical evaluation.

*5.6.1 Generalization of Algorithms.* The generalization of ApproxCov requires a few changes in Algorithm 1, yet the general idea of checking inclusion into $\mathsf{Cov_t}(\mathcal{U})$ of randomly selected combinations remains the same. In particular, in lines 1 and 15 we need to change the number of selected combinations and the final multiplier to ensure that the approximation is within the PAC guarantees. The second change is the uniform selection of combinations in the set taking into account the number of values each feature can take. Indeed, the sampling shall choose a feature with multiple values more often than binary features.

| Benchmark | Ground Truth | Approximation |
|---|---|---|
| baital_busybox_1_28_0 | 0.994542 | 0.993025 |
| waps_busybox_1_28_0 | 0.985651 | 0.983794 |
| quick_busybox_1_28_0 | 0.253179 | 0.256468 |
| baital_ecos-icse11 | 0.975684 | 0.975041 |
| waps_ecos-icse11 | 0.790062 | 0.788512 |
| quick_ecos-icse11 | 0.282384 | 0.277069 |
| baital_mpc50 | 0.980516 | 0.974432 |
| waps_mpc50 | 0.797631 | 0.789341 |
| quick_mpc50 | 0.538494 | 0.546906 |
| baital_phycore | 0.965864 | 0.961093 |
| waps_phycore | 0.774117 | 0.782127 |
| quick_phycore | 0.513699 | 0.519799 |
| baital_psim | 0.975527 | 0.971872 |
| waps_psim | 0.792510 | 0.785056 |
| quick_psim | 0.494380 | 0.498117 |

**Table 3: Approximation of $\mathsf{FracCov_2}(\mathcal{U}, \mathsf{F})$. Confidence parameters are $\delta = 0.0975$, $\varepsilon = 0.105$.**

The generalization of ApproxMaxCov is re-implemented, as follows: we start with a formula F encoded with Quantifier-Free Bit-Vector logic (QF_BV) representing constraints of the configurable system. The first step is to construct a QF_BV formula $\mathsf{G^F}$ by extending F with constraints presented in Section 4.2. At the next step the formula $\mathsf{G^F}$ is converted into CNF. We use the SMT solver z3 with the following tactics: Then(simplify, bit-blast, tseitin-cnf) (Boolean variables are added to keep track of the original variables). In the resulted CNF formula we approximately compute the number of solutions projected to the set $S$ with ApproxMC.

*5.6.2 Empirical Results.* To illustrate that ApproxCov can directly operate on instances without reduction to Boolean values, we experimented with the implementation of ApproxCov for feature models without binarization. Our benchmark suite consisted of 35 feature models and 35 sample sets from [21]. The feature models have up to 200 variables having between 2 and 6 values. We used the same cluster, $\varepsilon = \delta = 0.05$ for both ApproxCov and ApproxMaxCov. For comparison we used slight modifications of BLMCov and BLMMaxCov: the former just needed to be capable to read the new inputs, while in the latter we additionally replaced calls to SAT solver with calls to SMT solver (z3).

The timeouts were 3600 seconds for BLMCov, ApproxCov, and ApproxMaxCov, and 14400 seconds for BLMMaxCov. Memory limit was 4Gb. Both ApproxCov and ApproxMaxCov have been run 10 times, we report mean time among 10 runs and the furthest result from the exact value computed with BLMCov and BLMMaxCov.

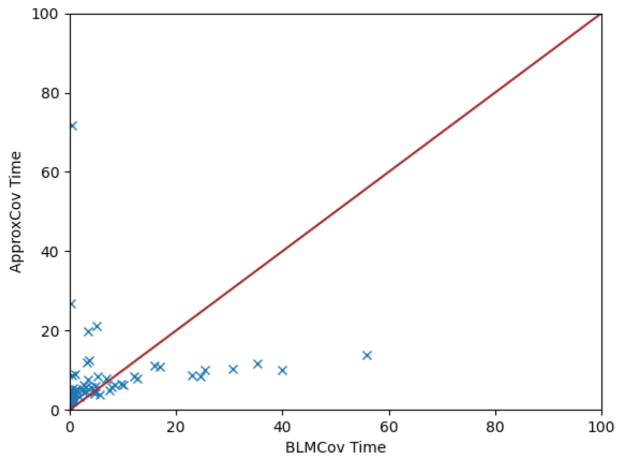

**Figure 9: Comparison of execution time of** ApproxCov **and** BLMCov **for** $t \in [2, 6]$ **on sample sets on which** BLMCov **have successfully terminated - 99 points in total. X axis shows** BLMCov **time in seconds, Y axis shows** ApproxCovGeneral **time in seconds. Red line indicates equal time.**

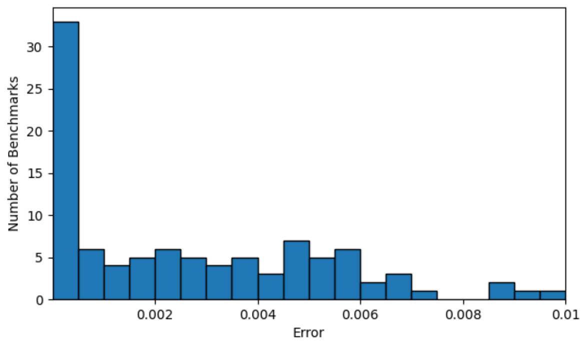

**Figure 10: Approximation error of** ApproxCov **for** $t \in [2, 6]$ **on 99 sample sets on which** BLMCov **have successfully terminated. The error is computed as** $max(abs(result_i - |\mathrm{Cov_t}(\mathcal{U})|)/|\mathrm{Cov_t}(\mathcal{U})|)$, **where** $result_i$ **is approximation returned by** ApproxCov **on the** $i^{th}$ **run.**

ApproxCov have successfully terminated on all benchmarks for $t \in [2, 6]$ within 150 seconds. BLMCov has failed 15 benchmarks for $t = 4$, 30 for $t = 5$, and 31 for $t = 6$ with Out-of-Memory error. The comparison of execution times is shown on a Figure 9. Due to the small size of benchmarks, BLMCov was faster on many of them. The approximation error is shown in Figure 10, the largest value was 0.0098.

ApproxMaxCov have successfully terminated on all benchmarks for $t \in [2, 4]$ within within the given timeout of 3600 seconds. 2 benchmarks for $t = 5$ and 16 benchmarks for $t = 6$ have timed out. BLMMaxCov has managed to terminate on 4 benchmarks for $t = 3$ and only on 1 benchmark for $t = 4$ within 14400 seconds timeout. The approximation error is shown in 11, the largest value was 0.005.

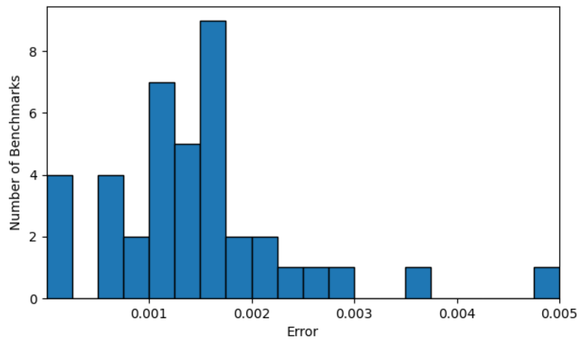

**Figure 11: Approximation error of** ApproxMaxCov **for** $t \in [2, 3]$ **on 40 benchmarks on which** BLMMaxCov **terminated. The error is computed as** $max(abs(result_i - |\mathrm{Cov_t}(\mathrm{Sol(F)})|)/|\mathrm{Cov_t}(\mathrm{Sol(F)})|)$, **where** $result_i$ **the approximation returned by** ApproxMaxCov **on** $i^{th}$ **run.**

## 5.7 Threats to Validity

*Internal Validity.* Our algorithms provide an estimation of the result with *Probably Approximately Correct* guarantees, and several runs may not yield identical results. To mitigate this threat we provide theoretical proofs that bound the potential error, and in the experiments we run each benchmark multiple times. In the obtained results the difference between multiple runs was below 0.06 for ApproxCov and below 0.021 for ApproxMaxCov, and on all benchmarks the worst approximation was always within PAC guarantees interval for both algorithms.

*External Validity.* To mitigate the threat of non-generalizability of our study we have used a large number of benchmarks used before in several prior studies [3, 32, 38, 44, 49, 53]. These benchmarks cover a wide range in the number of variables, clauses, and configurations.

## 6 CONCLUSION

Scalable and efficient computation of $t$-wise coverage is of pivotal importance for Combinatorial testing. In this work, we propose algorithms for estimating $t$-wise coverage for a given set of tests and also for tests sets with a given set of constraints. In particular, we present (1) a scalable Monte-Carlo based framework ApproxCov that is guaranteed to estimate the size of the coverage of for a given set of tests within $(1 \pm \varepsilon)$-factor of the ground truth with probability at least $1 - \delta$ for given $\varepsilon$ and $\delta$; (2) a scalable counting-based framework ApproxMaxCov that estimates maximal achievable coverage for a given formula and guarantees it to be within $(1 \pm \varepsilon)$-factor with probability at least $1 - \delta$ for given $\varepsilon$ and $\delta$. The approach have been evaluated on a large set of benchmarks involving up to 11000 variables and we have shown that both frameworks can provide highly accurate results even for estimation of 6-wise coverage of features. We also extended the frameworks to also include non-binary domains. Our work opens the possibility to compare various test set generators with the presented frameworks. An important direction for future work would be to perform an extensive evaluation of the existing test set generators by estimating the maximum achievable $t$-wise coverage and exploring the possibility to improve them based on the evaluation results.

## ACKNOWLEDGEMENTS

This work was supported in part by EU H2020 project Serums (826278-SERUMS-H2020-SC1-FA-DTS-2018-2020), by National Science Foundation USA (NSF HDR:TRIPODS-1934884), and by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005]. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

## 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] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Notices* 52, 3 (2016), 144–155.

[3] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: an adaptive weighted sampling approach for improved t-wise coverage. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 1114–1126. https://doi.org/10.1145/3368089.3409744

[4] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. 1–8.

[5] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[6] Renée C Bryce and Charles J Colbourn. 2009. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability* 19, 1 (2009), 37–53.

[7] Andrea Calvagna and Angelo Gargantini. 2010. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning* 45, 4 (2010), 331–358.

[8] 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.

[9] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable Approximate Model Counter. In *Proc. of CP*. 200–216.

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

[11] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proc. of IJCAI*.

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

[13] 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.

[14] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.

[15] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. [n.d.]. Constructing Test Suites for Interaction Testing. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. 38–48.

[16] R. Dechter, K. Kask, E. Bin, and R. Emek. 2002. Generating Random Solutions for Constraint Satisfaction Problems. In *Proc. of AAAI/IAAI*. 15–21.

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

[18] Arnaud Durand, Miki Hermann, and Phokion G Kolaitis. 2005. Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science* 340, 3 (2005), 496–513.

[19] 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.

[20] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *Proc. of ICSE*. 549–559.

[21] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. 2009. An improved meta-heuristic search for constrained interaction testing. In *2009 1st International Symposium on Search Based Software Engineering*. IEEE, 13–22.

[22] V. Gogate and R. Dechter. 2006. A New Algorithm for Sampling CSP Solutions Uniformly at Random. In *Proc. of CP*. 711–715.

[23] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*. 54–61.

[24] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2007. Near-Uniform sampling of combinatorial spaces using XOR constraints. In *Proc. of NIPS*. 670–676.

[25] 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.

[26] 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)*.

[27] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 540–550.

[28] 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.

[29] R.M. Karp and M. Luby. 1983. Monte-Carlo algorithms for enumeration and reliability problems. *Proc. of FOCS* (1983).

[30] Richard M Karp, Michael Luby, and Neal Madras. 1989. Monte-Carlo Approximation Algorithms for Enumeration Problems. *Journal of Algorithms* 10, 3 (1989), 429–448.

[31] Nathan Kitchen. 2010. *Markov Chain Monte Carlo stimulus generation for constrained random simulation*. Ph.D. Dissertation. UC Berkeley.

[32] 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.

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

[34] D Richard Kuhn, Raghu N Kacker, and Yu Lei. 2013. *Introduction to combinatorial testing*. CRC press.

[35] James H Kukula and Thomas R Shiple. 2000. Building circuits from relations. In *Proc. of CAV*.

[36] 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.

[37] 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.

[38] 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.

[39] Chuan Luo, Binqi Sun, Bo Qiao, Junjie Chen, Hongyu Zhang, Jinkun Lin, Qingwei Lin, and Dongmei Zhang. 2021. LS-Sampling: An Effective Local Search Based Sampling Approach for Achieving High t-Wise Coverage *(ESEC/FSE 2021)*. 1081–1092.

[40] Robert Mandl. 1985. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM* 28, 10 (1985), 1054–1058.

[41] 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.

[42] 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.

[43] 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.

[44] Kuldeep S. Meel. 2020. Model counting and uniform sampling instances. https://doi.org/https:/doi.org/10.5281/zenodo.3793090

[45] Kuldeep S Meel and S Akshay. 2020. Sparse hashing for scalable approximate model counting: Theory and practice. In *Proc. of LICS*.

[46] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC*.

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

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

[49] 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.

[50] David Lorge Parnas. 1976. On the design and development of program families. *IEEE Transactions on software engineering* 1 (1976), 1–9.

[51] 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.

[52] 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.]).

[53] 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.

[54] Iran Rodrigues, Márcio Ribeiro, Flávio Medeiros, Paulo Borba, Baldoino Fonseca, and Rohit Gheyi. 2016. Assessing fine-grained feature dependencies. *Inf. Softw. Technol.* 78 (2016), 27–52. https://doi.org/10.1016/j.infsof.2016.05.006

[55] Gadiel Seroussi and Nader H Bshouty. 1988. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory* 34, 3 (1988), 513–522.

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

[57] Neil JA Sloane. 1993. Covering arrays and intersecting codes. *Journal of combinatorial designs* 1, 1 (1993), 51–63. https://doi.org/10.1002/jcd.3180010106

[58] Mate Soos, Stephan Gocht, and Kuldeep S Meel. 2020. Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling. In *International Conference on Computer Aided Verification (CAV)*.

[59] Mate Soos and Kuldeep S Meel. 2019. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *Proc. of AAAI*.

[60] Larry Stockmeyer. 1983. The complexity of approximate counting. In *Proc. of STOC*. 118–126.

[61] 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.

[62] Keizo Tatsumi. 1987. Test case design support system. In *Proc. International Conference on Quality Control (ICQC'87)*. 615–620.

[63] 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.

[64] Leslie G. Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.

[65] 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.

[66] Wei Wei and Bart Selman. 2005. A new approach to model counting. In *Proc. of SAT*.

[67] 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.

[68] Cemal Yilmaz, Sandro Fouché, Myra B. Cohen, Adam A. Porter, Gülsen Demiröz, and Ugur Koc. 2014. Moving Forward with Combinatorial Interaction Testing. *Computer* 47, 2 (2014), 37–45.