A Scalable t-wise Coverage Estimator: Algorithms and Applications

Eduard Baranov, Sourav Chakraborty, Axel Legay, Kuldeep S. Meel, and N. Variyam Vinodchandran

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 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.

We designed 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, ApproxCov takes in a test set $\mathcal{U}$ 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 $1 - \delta$ for a given tolerance parameter ε and a confidence parameter δ. A scalable framework ApproxMaxCov for a given formula F outputs an approximation which is guaranteed to be within $(1 \pm \varepsilon)$ factor of the maximum achievable t-wise coverage under F, with probability $\geq 1 - \delta$ for a given tolerance parameter ε and a confidence parameter δ. Our comprehensive evaluation demonstrates that ApproxCov and ApproxMaxCov can handle benchmarks that are beyond the reach of current state-of-the-art approaches.

In this paper we present proofs of correctness of ApproxCov, ApproxMaxCov, and of their generalizations. We show how the algorithms can improve the scalability of a test suite generator while maintaining its effectiveness. In addition, we compare several test suite generators on different feature combination sizes t.

Index Terms—Configurable software, t-wise coverage, Approximation

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 [1]. 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 a 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 [2] 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 [3] 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 the embedded Linux kernel for microcontrollers has over 7.7×10^{417} configurations [6].

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 [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. 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 [3]. Another study showed that discovered variability bugs in the Linux kernel involve up to $5-$wise feature interactions [17]. 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

1. defined formally in Section 3

E. Baranov and A. Legay are with Universite catholique de Louvain.
S. Chakraborty is with Indian Statistical Institute.
K. S. Meel is with University of Toronto.
N. V. Vinodchandran is with University of Nebraska-Lincoln.
Manuscript received 29 November 2023; revised 8 May 2024; accepted 11 June 2024. Date of publication 27 June 2024. Digital Object Identifier 10.1109/TSE.2024.3419919.

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. 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 $\Omega(2^t \log n)$ [18]. The presence of constraints brings additional complications to the design of a t-covering array.

Given the practical importance of 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 greedy algorithms [19], [20], [21] to constraint-based systems [22], [23] 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 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 F. For the former computation, the state-of-the-art techniques maintain a hash map of size $\mathcal{O}(\binom{n}{t}2^t)$, and the map is updated for every element of $\mathcal{U}$. Furthermore, to compute the possible t-combinations over the solutions of F, the best-known algorithms check whether F conjuncted with a t-combination (expressed as a conjunction of literals) is satisfiable[2]. Unfortunately, for most practical instances, the computation of both quantities is beyond the reach of 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

This paper is an extension of our previous work [24] where we gave an affirmative answer to the above question. We designed the first scalable technique that provides rigorous estimates of t-wise coverage. In particular, we designed:

1) A scalable Monte Carlo-based algorithm ApproxCov that takes in a test suite $\mathcal{U}$, a coverage parameter t, a tolerance parameter ε, a confidence parameter δ 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-sized combinations of elements in $\mathcal{U}$ and $|\cdot|$ is a set cardinality. ApproxCov takes only $\mathcal{O}(2^t \cdot t \log n)$ (for a constant ε and δ, and bounded $|\mathcal{U}|$) space in contrast to the space requirement of $\mathcal{O}(\binom{n}{t}2^t)$ for the existing techniques. 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 a formula F, a coverage parameter t, a tolerance parameter ε, and a confidence parameter δ 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 $1-\delta$, where $\mathsf{Sol}(\mathsf{F})$ is a set of all satisfying assignments of F. 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 [25], [26], [27], [28], [29], and we employ state-of-the-art hashing-based approximate model counter ApproxMC4 [30].
3) We demonstrated 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 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 observed significant runtime improvement: in particular, the comparison with the corresponding algorithms in a current state-of-the-art test generator suite BAITAL[4] showed that ApproxCov achieves from 2 to 294 factor speedup over the BAITAL implementation, while ApproxMaxCov achieves from 6 to 131 factor speedup over the BAITAL implementation.
4) We showed how the algorithms can be generalized to estimate t-wise coverage on configurable systems where each feature can take values from a discrete domain. We demonstrated with experimental evaluation that the generalized algorithms are effective and can provide a close estimation of t-wise coverage.

In this work, we extend the previous results with the following additional contributions:

1) We provide a detailed description of a reduction of $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ computation to the problem of projected model counting used in ApproxMaxCov.
2) We provide proofs of correctness for both ApproxCov and ApproxMaxCov.
3) We provide a detailed description of generalized algorithms ApproxCovGeneral and ApproxMaxCovGeneral that estimate t-wise coverage on configurable systems where each feature can take values from a discrete domain.
4) We integrate the algorithms ApproxCov and ApproxMaxCov into adaptive weighted sampling and demonstrate with a set of benchmarks the greatly improved scalability of the approach without reducing the achievable t-wise coverage.
5) We perform the comparison of several tools for test suite generation and show that ApproxCov and ApproxMaxCov can handle the comparison for $t \in$

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
4. https://github.com/meelgroup/baital

$\{2, 3, 4, 5, 6\}$ while previous works have never considered the comparison for $t \geq 3$.

A 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 [27], [28], [29], [30].

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 and ApproxMaxCov provide the combinatorial testing community sound tools to compare different test suite generation techniques for large benchmarks. 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 related work in Section 2. We present notation and preliminaries in Section 3. We then present the proposed algorithms 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. In Section 6 we generalize the algorithms to the case where each feature can take a finite number of values. We present an application of the algorithms to the test suite design in Section 7. We then present an empirical comparison of several tools for the test suite design in Section 8. Finally, in Section 9, we give concluding remarks.

2 Related Work

2.1 Combinatorial Testing

Since the introduction of combinatorial testing in the 1980s as an effective option for testing configurable systems [7], [8], steady progress has been reported on this topic. We refer the reader to [9], [10], [11] 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 suite can be represented as a covering array [31]. Covering arrays are arrays of k vectors of length n with the property that the projection onto any t columns contains all 2^t possibilities [32]. Over the decades, the construction of covering arrays has witnessed a wide variety of approaches including greedy search [33], [31], [34], [35], [36], [20], [37], [38], [39], divide-and-compose [40], genetic algorithms [41], and tabu search [42], [43].

Modern software systems have a large number of features, and the design of a covering array is often impractical for $t > 2$. Furthermore, modern systems 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 [44]. 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 [45], random seeding of DPLL-based SAT solvers [46], Markov Chain Monte Carlo-based methods [47], [48], [49], [50], interval propagation and belief networks-based methods [51], [52], MaxSAT-based techniques such as Quicksampler [53], hashing-based approaches [54], [55], [30], knowledge compilation-based approaches such as KUS [56], WAPS [57], Baital [58], and LS-Sampling [59].

2.2 Model Counting

Valiant initiated the complexity theoretic study of model counting and showed that the problem of model counting for a CNF formula is #P-complete [60]. The problem of projected model counting which we employ in this paper is shown to be #·NP-complete [61]. Given the computational intractability of (projected) model counting, we are often interested in (ε, δ)-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 (ε, δ)-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, and 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 [25], subsequently pursued by Gomes, Sabharwal, and Selman [62].

Chakraborty, Meel, and Vardi proposed the first scalable approximate model counter, ApproxMC, which invoked the underlying SAT solver, CryptoMiniSat, $\mathcal{O}(n)$ times (where n is the number of variables in the original formula). Subsequently, Chakraborty et al [28] reduced the number of SAT calls from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$; the corresponding counter was called ApproxMC2. Soos and Meel [29] 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 [30], [63][5].

2.3 t-wise Coverage Estimation

While combinatorial testing has witnessed over four decades of sustained interest 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 π. Given such a π, the state-of-the-art techniques simply invoke a SAT solver to check whether $\mathsf{F} \wedge \pi$ is

5. While beta version of ApproxMC5 is released; ApproxMC's developer recommends the usage of ApproxMC4

satisfiable. In our work, we use the implementation of these techniques due to Baranov, Legay, and Meel [58], which was used in the evaluation of the current state-of-the-art-test generator suite, Baital. We use BLMCov and BLMMaxCov to denote the implementations for computations of $\mathsf{Cov_t}(\mathcal{U})$ and $\mathsf{Cov_t}(\mathsf{Sol(F)})$ respectively. To the best of our knowledge, BLMCov and BLMMaxCov represent the current state of the art; considering other recent tools Smarch [64] and LS-Sampler [59] that compute t-wise coverage, both use the same idea for the $\mathsf{Cov_t}(\mathsf{Sol(F)})$ computation, while for the $\mathsf{Cov_t}(\mathcal{U})$ computation, Smarch requires a set of precomputed valid combinations, and LS-Sampler utilizes the same approach but do not have a dedicated option to compute the number of combinations for a given set.

3 Notations and Preliminaries

3.1 Boolean Formulas

A literal is a Boolean variable or its negation. A clause is a disjunction of a set of literals. A propositional formula F in conjunctive normal form (CNF) is a conjunction of clauses. $\mathsf{Vars(F)}$ denotes the set of variables appearing in F. The $\mathsf{Vars(F)}$ is also called the *support* of F. A *satisfying assignment* or *witness* of F, denoted by σ, is an assignment of truth values to variables in its support such that 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 an 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 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 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 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 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

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

3.2 t-wise Coverage

The formulation of combinatorial interaction testing (CIT) assigns a variable corresponding to every feature of a software system. To simplify the presentation, we assume that each feature can take two states: *True* or *False*. The general case where features can take a finite number of values is described in Section 6. Let $X = \{x_1, \cdots, x_n\}$ be the set of all the variables (corresponding to n features). Then a configuration σ 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 σ represented as a set of literals, we define the t-wise coverage of σ denoted by $\mathsf{Cov_t}(\sigma) = \{T \subseteq \sigma \mid |T| = t\}$, the set of all subsets of literals of the size t in σ. $\mathsf{Cov_t}(\sigma)$ represents the set of t-sized feature combinations due to σ. 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 σ, $|\mathsf{Cov_t}(\sigma)| = \binom{|X|}{t} = \binom{n}{t}$. 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 Ω. The above discussion leads to the following observation which is crucial for the proof of correctness of our algorithm.

Observation 3.1. *For any $\mathcal{U} \neq \emptyset$ over a set of variables X and any $1 \leq t \leq |X|$,*

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

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 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 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 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					
$\mathsf{Sol(F)}$	$(1,2)$	$(1,3)$	$(1,4)$	$(2,3)$	$(2,4)$	$(3,4)$
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

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 $\mathcal{U} \subset \prod_i \{x_i, \neg x_i\}$, a coverage parameter t, 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, a coverage parameter t, 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 $(1+\epsilon)|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$.

Before we present those algorithms, we discuss how to estimate the fractional coverage. 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 $\mathcal{U} \subset \mathsf{Sol}(\mathsf{F})$, we use $\mathsf{ApproxCov}(\mathcal{U}, \epsilon_1, \delta_1)$ and $\mathsf{ApproxMaxCov}(\mathsf{F}, \epsilon_2, \delta_2)$ to obtain estimates Out_1 and Out_2 for $|\mathsf{Cov_t}(\mathcal{U})|$ and $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ respectively. So we have with probability at least $(1-\delta_1)$ the following Equation 1 and with probability $(1-\delta_2)$ the Equation 2 holds.

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

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

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

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

Thus given a Boolean formula F, a set $\mathcal{U} \subset \mathsf{Sol}(\mathsf{F})$, an error parameter ϵ and a confidence parameter δ, if we set ϵ_1, ϵ_2, δ_1 and δ_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 $(1-\delta)$

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

4.1 Counting the Coverage of a Test Suite

The intuition behind the ApproxCov is similar to the Monte-Carlo method. Given a set of variables $X = \{x_1, \ldots, x_n\}$, consider the universe set $\Omega := \mathsf{Cov_t}(\prod_i \{x_i, \neg x_i\})$ with all possible t-wise coverage tuples and a test set $\mathcal{U} \subseteq \prod_i \{x_i, \neg x_i\}$. The algorithm picks a set $\mathcal{S}$ of size ℓ of random t-wise coverage tuples from Ω (ℓ 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 $\frac{m}{\ell} \cdot |\Omega|$.

ApproxCov shown in Algorithm 1 starts by setting the size ℓ of sample set depending on the parameters ε and δ. In the **For** loop between line 3 and line 7 it picks ℓ samples uniformly at random from Ω. The sampling is done in two steps. For each sample, at first ApproxCov selects uniformly t variables to be used in a sample at line 4. In the second step, it randomly selects a value for each variable at line 5. Note that this procedure gives a random element of Ω 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.

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

1: $\ell \leftarrow \left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta) \right\rceil$
2: Initialise $\mathcal{M} = \emptyset$
3: **for** $k = 1; k \leq \ell; k++$ **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 $w_k \to 0$ into $\mathcal{M}$
7: **end for**
8: **for** $\sigma \in \mathcal{U}$ **do**
9: **for** $k = 1; k \leq \ell; k++$ **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 $\frac{\binom{n}{t}2^t}{\ell}\sum_{k=1}^{\ell}\mathcal{M}[w_k]$

For establishing correctness of ApproxCov, we need concentration inequality due to Chernoff. For a random variable V, let $\mathbb{E}[V]$ denote its expected value.

Theorem 4.1 (Chernoff's Bound [65])**.** *Suppose $v_1, ..., v_n$ are independent random variables taking values in $\{0, 1\}$. Let $V = \sum_{i=1}^{n} v_i$ and $\mu = \mathbb{E}[V]$ then for any $0 < \delta < 1$*

$$\Pr\left(|V - \mu| \geq \delta\mu\right) \leq 2e^{-\frac{\delta^2\mu}{3}}$$

We state the efficiency and correctness guarantees of ApproxCov as a theorem.

Theorem 4.2. *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* $(1-\delta)$,

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

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

Proof. Let $W_1,\ldots,W_\ell$ be $\{0,1\}$-random variables where $W_k = 1$ iff $\mathcal{M}[w_k]$ is assigned 1 by ApproxCov. Note that for all $1 \leq k \leq \ell$ the set w_k is uniformly drawn from the set Ω, therefore the random variables $W_1,\ldots,W_\ell$ are independent. Clearly for all k

$$\mathbb{E}(W_k) = \frac{|\mathsf{Cov_t}(\mathcal{U})|}{|\Omega|} = \frac{|\mathsf{Cov_t}(\mathcal{U})|}{\binom{n}{t}2^t}.$$

So by Chernoff's bound (Theorem 4.1) we have that

$$\Pr\left[\left|\sum_{k=1}^{\ell}\mathcal{M}[w_k]-\mu\right| \geq \epsilon\mu\right] \leq 2e^{-\frac{\epsilon^2\mu}{3}},$$

where, $\mu = \mathbb{E}(\sum_{k=1}^{\ell}W_k)$ which by linearity of expectation is equal to $\ell\frac{|\mathsf{Cov_t}(\mathcal{U})|}{\binom{n}{t}2^t}$.

Now, because $|\mathsf{Cov_t}(\mathcal{U})| \geq \binom{n}{t}$, by the choice of ℓ we have $\mu \geq \frac{3}{\epsilon^2}\ln(2/\delta)$ and hence $2e^{-\frac{\epsilon^2\mu}{3}} \leq \delta$. So with probability at least $(1-\delta)$ we have

$$(1-\epsilon)\ell\frac{|\mathsf{Cov_t}(\mathcal{U})|}{\binom{n}{t}2^t} \leq \sum_{k=1}^{\ell}W_k \leq (1+\epsilon)\ell\frac{|\mathsf{Cov_t}(\mathcal{U})|}{\binom{n}{t}2^t}.$$

Therefore, the output of ApproxCov, which is $\frac{\binom{n}{t}2^t}{\ell}\sum_{k=1}^{\ell}\mathcal{M}[w_k]$, is between $(1-\epsilon)|\mathsf{Cov_t}(\mathcal{U})|$ and $(1+\epsilon)|\mathsf{Cov_t}(\mathcal{U})|$.

For proving the bound on space complexity of ApproxCov, first note that since the set of variables X has n variables, to store a variable (or rather the index of the variable) requires $O(\lceil\log_2 n\rceil)$ number of bits. Since each w that is sampled in line 5 is a set of t literals, $O(t\lceil\log_2 n\rceil)$ space is required for storing each w in $\mathcal{M}$. So to store the map $\mathcal{M}$ ApproxCov uses $O(\ell t\lceil\log_2 n\rceil) = O\left(\left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta)\right\rceil t\lceil\log_2 n\rceil\right)$ bits. In addition to storing $\mathcal{M}$, $O(t\lceil\log_2 n\rceil + \lceil\log_2\ell\rceil + \lceil\log_2|\mathcal{U}|\rceil)$ amount of space is needed to store local variables such as ℓ and loop iterators. So overall space required is $O\left(\left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta)\right\rceil t\lceil\log_2 n\rceil + \lceil\log_2|\mathcal{U}|\rceil\right)$.

To analyse the runtime of ApproxCov first note that in the **For** loop between line 3 and line 7 the total time taken for choosing the map $\mathcal{M}$ is $O(\ell t\lceil\log_2 n\rceil)$, following the same arguments as we did for the space complexity. Now in the **For** loop between line 9 and line 13 for each round of the **For** loop, the only non-trivial step is checking the **If** condition in line 10, where it is checked if $w \subseteq \sigma$, which can take at most $O(t\lceil\log_2 n\rceil)$ steps (we assume random access to the indices of σ). So the **For** loop between line 8 and line 14 take at most $O(|\mathcal{U}|\ell t\lceil\log_2 n\rceil)$. So overall the time complexity of ApproxCov is $O(\ell t\lceil\log_2 n\rceil) + O(|\mathcal{U}|\ell t\lceil\log_2 n\rceil)$ which is $O\left(\left\lceil 3\frac{2^t}{\epsilon^2}\ln(2/\delta)\right\rceil t\lceil\log_2 n\rceil|\mathcal{U}|\right)$. □

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 $\mathsf{Sol}(\mathsf{F})$ 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

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

Then it will use an approximate projected model counting algorithm[6] (which we call ApproxCount) on $\mathsf{G^F}$ to output an (ϵ,δ) estimate of $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$. Since the output of ApproxCount is guaranteed to be within $(1\pm\varepsilon)$ of $|\mathsf{Sol}(\mathsf{G^F})_{\downarrow\mathsf{S}}|$ 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$(\mathsf{F}, t, \varepsilon, \delta)$

1: $(\mathsf{G^F}, S) \leftarrow$ ConstructGFormula(F)
2: $c \leftarrow$ 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. We now present the construction of $\mathsf{G^F}$.

Construction of $\mathsf{G^F}$: $\mathsf{G^F}$ encodes F and all the t-wise coverage tuples contained in $\mathsf{Sol}(\mathsf{F})$. Given natural numbers n and t, and a Boolean formula F on n variables $\{x_0,\ldots,x_{n-1}\}$[7] we will define a new Boolean formula $\mathsf{G^F}$ 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: $Y_1,\ldots,Y_t$ of size $\lceil\log_2 n\rceil$ and $Z = \{z_1,\ldots,z_t\}$ of size t. Intuitively, S encodes a t-wise coverage tuple: Y_i is a bit-vector that encodes the index of the i^{th} variable in the tuple 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, $\mathsf{G^F}$ adds additional constraints to the formula F. To formally define the formula $\mathsf{G^F}$ we will need to first introduce some notations and define certain basic Boolean formulae.

For an integer $0 \leq j \leq (n-1)$, let $\textsc{Bin}(j)$ denote the $\lceil\log_2 n\rceil$ binary representation of j with $True$ representing 1 and $False$ representing 0. Similarly, we will view a $\{True, False\}$ assignment to a set $\lceil\log_2 n\rceil$ variables (say $\{a_1,\ldots,a_{\lceil\log_2 n\rceil}\}$) as an integer between 0 and $2^{\lceil\log_2 n\rceil}-1$.

For any set of n variables $A = \{a_1, a_2, \cdots, a_n\}$, let $A^{(i)}$ denote the set of first i variables $\{a_1,\cdots,a_i\}$. For two sets of variables $A = \{a_1,\ldots,a_k\}$ and $B = \{b_1,\ldots,b_k\}$ of the same size, $\textsc{Eq}(A,B)$ be the formula that is *True* iff A and B represents same integers, $\textsc{Less}(A,B)$ be the formula that is *True* iff the integer represented by A is strictly less than the integer represented by B, and $\textsc{Leq}(A,B)$ be the formula

6. In our implementation we use ApproxMC [66]
7. Choosing $\{x_0,\ldots,x_{n-1}\}$ instead of $\{x_1,\ldots,x_n\}$ is for the ease of presentation.

that is *True* iff the integer represented by A is less than or equal to the integer represented by B.

$$\begin{aligned}\mathrm{EQ}(A,B) &:= \bigwedge_{i=1}^{n}(a_i \Leftrightarrow b_i)\\ \mathrm{LESS}(A,B) &:= \bigvee_{i=0}^{k-1}\Big(\mathrm{EQ}(A^{(i)},B^{(i)}) \wedge \neg a_{(i+1)} \wedge b_{(i+1)}\Big)\\ \mathrm{LEQ}(A,B) &:= \mathrm{LESS}(A,B) \vee \mathrm{EQ}(A,B)\end{aligned}$$

To encode t-wise coverage tuples we use the following constraints: (i) variable indexes in a tuple are in $\{0,\ldots,n-1\}$, (ii) the ordering ensures a single representation of each tuple, and (iii) the binary assignment variables $Y \cup Z$ encodes a t-wise coverage tuple contained in the assignment to F. Constraints encoding is shown below.

$$\begin{aligned}\mathrm{C}(X,Y_1,Y_2,\cdots Y_t,Z) \quad :=\\ (i)\quad &\bigwedge_{i=1}^{t}\mathrm{LEQ}(Y_i,\textsc{Bin}(n-1)) \wedge\\ (ii)\quad &\bigwedge_{i=1}^{t-1}\mathrm{LESS}(Y_i,Y_{i+1}) \wedge\\ (iii)\quad &\bigwedge_{i=0}^{n-1}\bigwedge_{j=1}^{t}(\mathrm{EQ}(Y_j,\textsc{Bin}(i)) \Rightarrow (x_i \Leftrightarrow z_j))\end{aligned}$$

Finally, we define the formula G^{F} as

$$\mathsf{G}^{\mathsf{F}}(X,Y_1,\ldots,Y_t,Z) \quad := \quad \mathsf{F}(X) \wedge \mathrm{C}(X,Y_1,Y_2,\cdots Y_t,Z).$$

With the intuition behind that each valid assignment to the set $S = \bigcup_{i=1}^{t} Y_i \cup Z$ represents a distinct t-wise coverage tuple, it is easy to see that the number of different valid assignments to S is the desired value $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$.

Theorem 4.3.

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

Proof. The theorem follows from the fact that for any Boolean formula F, the sets $\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))$ and $\mathsf{Sol}(\mathsf{G}^{\mathsf{F}})_{\downarrow(\mathsf{Y_1}\cup\mathsf{Y_2}\cup\cdots\cup\mathsf{Y_t}\cup\mathsf{Z})}$ are identical. Recall that $\mathsf{Sol}(\mathsf{F})$ is the set of all satisfying assignments of F. For any satisfying assignment $\sigma \in \mathsf{Sol}(\mathsf{F})$ let $\mathsf{G}^{\mathsf{F}}(\sigma)$ is the formula obtained by setting the X variables in G^{F} according to σ. Since $G^{\mathsf{F}}(X,Y_1,\ldots,Y_t,Z) := \mathsf{F}(X) \wedge \mathrm{C}(X,Y_1,\ldots,Y_t,Z)$ and by definition of σ, $\mathsf{F}(\sigma)$ is $True$, so $\mathsf{G}^{\mathsf{F}}(\sigma) = \mathrm{C}(\sigma)$. Note that set of solutions $\mathsf{Sol}(\mathsf{G}^{\mathsf{F}})_{\downarrow(\mathsf{Y_1}\cup\mathsf{Y_2}\cup\cdots\cup\mathsf{Y_t}\cup\mathsf{Z})}$ is identical to the set of solutions $\bigcup_{\sigma\in\mathsf{Sol}(\mathsf{F})}\left(\mathsf{Sol}(\mathrm{C}(\sigma))_{\downarrow(\mathsf{Y1}\cup\cdots\cup\mathsf{Y_t}\cup\mathsf{Z})}\right)$. Since $\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))$ is same as $\bigcup_{\sigma\in\mathsf{Sol}(\mathsf{F})}\mathsf{Cov_t}(\sigma)$, it is enough to prove that for any $\sigma \in \mathsf{Sol}(\mathsf{F})$ the sets $\mathsf{Sol}(\mathrm{C}(\sigma))_{\downarrow(\mathsf{Y_1}\cup\cdots\cup\mathsf{Y_t}\cup\mathsf{Z})}$ and $\mathsf{Cov_t}(\sigma)$ are identical.

Let $(\sigma,\gamma_1,\ldots,\gamma_t,\zeta)$ be a satisfying assignment of the formula $\mathrm{C}(\sigma,Y_1,\ldots,Y_t,Z)$. Then the following two condition holds:

1) The $\gamma_1,\ldots,\gamma_t$ encodes (when looked as a binary string) indices $j_1,\ldots,j_t$ such that $0 \le j_1 < \cdots < j_t \le (n-1)$.
2) the variables $z_1,\ldots,z_t$ is assigned the same Boolean value as $x_{j_1},\ldots,x_{j_t}$ in σ or in other words $\zeta = \sigma \cap T$ where $T = \{j_1,\ldots,j_t\}$.

So the set $\mathsf{Sol}(\mathrm{C}(\sigma))_{\downarrow(\mathsf{Y_1}\cup\mathsf{Y_2}\cup\cdots\cup\mathsf{Y_t}\cup\mathsf{Z})}$ comprises of all the restriction of σ to sets of size t and thus the set $\mathsf{Sol}(\mathrm{C}(\sigma))_{\downarrow(\mathsf{Y_1}\cup\mathsf{Y_2}\cup\cdots\cup\mathsf{Y_t}\cup\mathsf{Z})}$ is exactly same as $\mathsf{Cov_t}(\sigma)$. This proves the theorem.

□

5 Experiments

We have implemented both algorithms ApproxCov and ApproxMaxCov in Python 3 and we have performed a set of experiments to evaluate their precision and efficiency[8]. In particular, 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 outputs of ApproxCov and ApproxMaxCov close to the correct values and within the PAC guarantees boundary?
- **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?

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 [67], [68], [69], [6], [58]. The benchmarks have between 565 and 11254 variables, between 1164 and 62183 clauses, and between 9.7×10^{13} and 7.7×10^{417} solutions. Unfortunately, existing approaches are not capable of computing $|\mathsf{Cov_t}(\mathcal{U})|$ and $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ for $t \ge 3$ on large benchmarks. Therefore, we added 111 extra benchmarks from [70] that have between 10 and 500 variables.

ApproxCov takes configuration sets as inputs, therefore we used 3 sampling tools —WAPS [57], Quicksampler [71], and Baital [58] —to generate sets of 1000 configurations for each of the 234 benchmarks. On several benchmarks Quicksampler failed to generate 1000 configurations and we excluded those incomplete configuration sets from the experiments. The total number of configuration sets we used for the evaluation is 674.

All experiments were conducted on a high performance computer cluster with Intel Skylake 16-cores Xeon 6142 processors at 2.6 GHz.

8. Instructions for the replication of all experiments in the paper can be found in supplemental materials

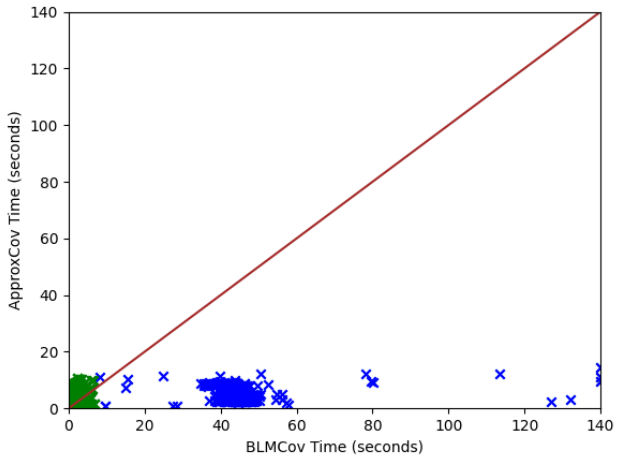

Fig. 1: Comparison of execution time of ApproxCov and BLMCov on 674 sample sets for $t = 2$. X and Y axes show time in seconds for BLMCov and ApproxCov, respectively. In this and the following figures, green points correspond to benchmarks with less than 500 variables and blue are for benchmarks with more than 500 variables.

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 configuration sets described above. We used ε and δ equal to 0.05. ApproxCov computations have been performed 10 times with different random seeds 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 performed the same computations with BLMCov [58]. The timeout was set to 3600 seconds and the memory limit was set to 4Gb for both 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 $\delta = \varepsilon = 0.05$. As in the previous experiment, all computations with ApproxMaxCov have been performed 10 times with different random seeds. For comparison, we performed the same computations with BLMMaxCov [58]. Originally, the timeout was set to 3600 seconds for both tools. Unfortunately, BLMMaxCov failed to finish half of the 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 $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ 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 $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ are derived from the selection of ε and δ in the computation of ApproxCov and ApproxMaxCov: $\delta = 0.0975$, $\varepsilon = 0.105$.

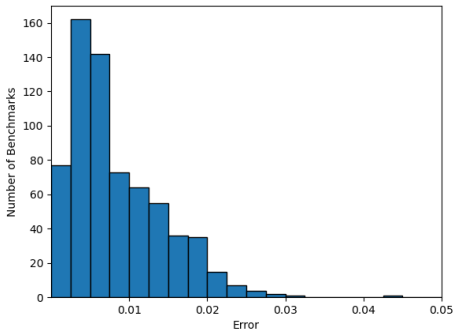

Fig. 2: Approximation error of ApproxCov for $t = 2$ on 674 sample sets computed as $max(abs(result_i - |\mathsf{Cov_2}(\mathcal{U})|)/|\mathsf{Cov_2}(\mathcal{U})|)$, where $result_i$ is approximation returned by ApproxCov on the i^{th} run.

5.3 ApproxCov Results

In the first experiment we computed $|\mathsf{Cov_t}(\mathcal{U})|$ with ApproxCov and BLMCov. For $t = 2$, ApproxCov has successfully terminated on all sample sets within 25 seconds. For BLMCov the longest computation took 3000 seconds. BLMCov was faster on 237 benchmarks though all such benchmarks have less than 500 variables except one. Among large sample sets with more than 500 variables, only 1 was faster with BLMCov, while on the rest of the 367 sets ApproxCov showed at least 5x speedup on 80% of benchmarks and at least 10x speedup on 45% of benchmarks. Time comparison is shown Figure 1; 3 points on the right border correspond to higher than 2800 seconds time for BLMCov. Slower performance on smaller benchmarks is explained by the fact that the number of picked elements depends on the selected ε and δ 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 the implementation, this case is optimized: instead of randomly selecting the combinations, we take the set of all the different combinations as a key set of the map $\mathcal{M}$ and, as a result, the sum of values in $\mathcal{M}$ at the end of the algorithm is the exact value of $|\mathsf{Cov_t}(\mathcal{U})|$.

To check the accuracy of ApproxCov approximation, we compared the approximation results with the correct value of $|\mathsf{Cov_2}(\mathcal{U})|$ computed by BLMCov. For each sample set we took the maximal value of $abs(result - |\mathsf{Cov_2}(\mathcal{U})|)/|\mathsf{Cov_2}(\mathcal{U})|$ among 10 $results$ of ApproxCov. Figure 2 shows the histogram of errors, the largest error was 0.0435 which is smaller than selected ε.

For $t \geq 3$, BLMCov has successfully terminated on few 'small' benchmarks within the given time and memory budget: 315 benchmarks for $t = 3$, 106 for $t = 4$, 48 for $t = 5$, and 23 for $t = 6$. In comparison, ApproxCov terminated on all benchmarks within 420 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 0.0364. 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 $|\mathsf{Cov_2}(\mathcal{U})|$ and within the selected boundary

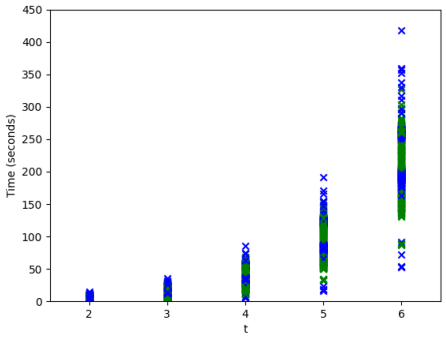

Fig. 3: ApproxCov execution time on 674 sample sets for $t \in [2, 6]$. X axis shows the value of t, Y axis shows time in seconds.

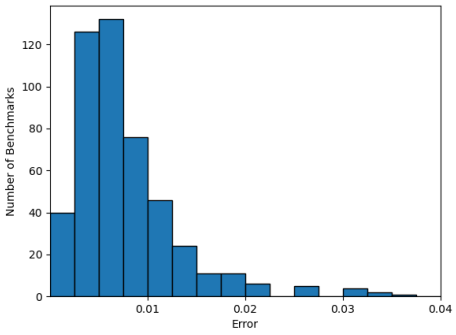

Fig. 4: Approximation error of ApproxCov for $t \in [3, 6]$ on all benchmarks for which BLMCov has successfully terminated - 492 elements in total. The error is computed as $max(abs(result_i - |\mathsf{Cov_t}(\mathcal{U})|)/|\mathsf{Cov_t}(\mathcal{U})|)$, where $result_i$ is approximation returned by ApproxCov on the i^{th} run.

from PAC guarantees, thus answering **RQ1**. Comparison of execution time allows us to give a positive answer to **RQ3**: ApproxCov can estimate $|\mathsf{Cov_t}(\mathcal{U})|$ for $t = 6$, while existing method fails on half of the sample sets for $t = 3$.

5.4 ApproxMaxCov **Results**

ApproxMaxCov has successfully terminated on all benchmarks for $t = 2$ within 210 seconds except 1 benchmark that required 879 seconds. For comparison, BLMMaxCov has successfully terminated on 2 benchmarks with more than 500 variables for $t = 2$ with a 3600 seconds timeout. After raising the timeout to 28800 seconds it succeeded on 223 benchmarks out of 234. The speed up factor of ApproxMaxCov on benchmarks with more than 500 variables has a range between 6 and 131.

To check the accuracy of ApproxMaxCov approximation, we compared it with $\mathsf{Cov_2(Sol(F))}$. For each benchmark, we took the maximal value of $abs(result - |\mathsf{Cov_2(Sol(F))}|)/|\mathsf{Cov_2(Sol(F)}|$ among 10 $results$ of ApproxMaxCov. Figure 5 shows the histogram of errors, the largest error was 0.0055 which is smaller than selected ε.

For larger values of t, ApproxMaxCov timed out only on the largest benchmark for $t = 6$. The execution time for various values of t is shown in Figure 6. BLMMaxCov was able to compute 81 benchmarks for $t = 3$, 28 for $t = 4$,

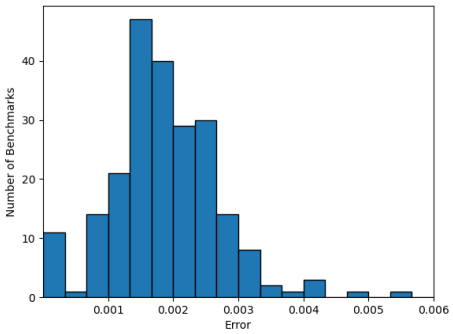

Fig. 5: Approximation error of ApproxMaxCov for $t = 2$ on 223 benchmarks for which BLMMaxCov terminated. The error is computed as $max(abs(result_i - |\mathsf{Cov_2(Sol(F))}|)/|\mathsf{Cov_2(Sol(F))}|)$, where $result_i$ the approximation returned by ApproxMaxCov on i^{th} run.

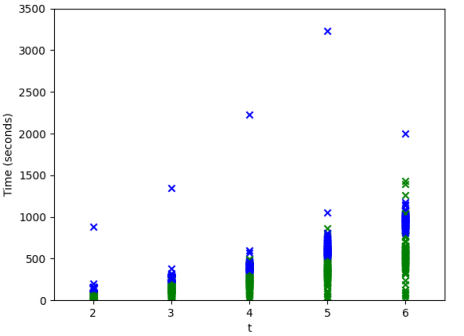

Fig. 6: ApproxMaxCov execution time on 234 benchmarks for $t \in [2, 6]$. X axis shows the value of t, Y axis shows time in seconds. Point on the top border corresponds to timeout.

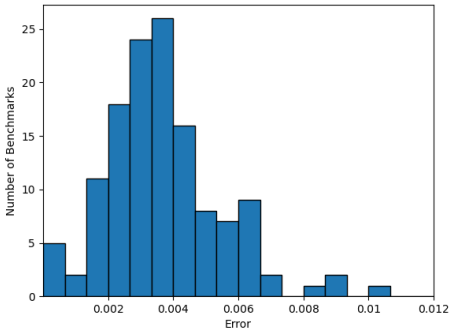

Fig. 7: Approximation error of ApproxMaxCov for $t \in [3, 6]$ on all benchmarks on which BLMMaxCov has successfully terminated - 132 elements in total. The error is computed as $max(abs(result_i - |\mathsf{Cov_t(Sol(F))}|)/|\mathsf{Cov_t(Sol(F))}|)$, where $result_i$ the approximation returned by ApproxMaxCov on i^{th} run.

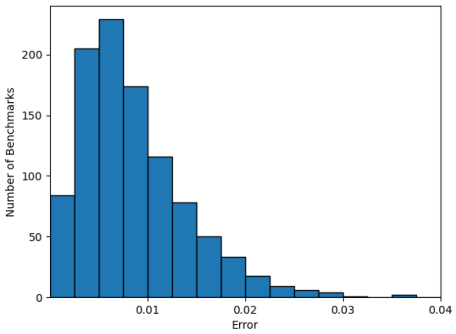

Fig. 8: Approximation error of $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ for $t \in [2, 6]$ on all benchmarks on which both BLMCov and BLMMaxCov have successfully terminated - 1009 benchmarks in total. It has been computed as $max(abs(result_i - \mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}))/\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}))$, 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.

Benchmark	Ground Truth	Approximation
baital_busybox_1_28_0	0.994542	0.996919
waps_busybox_1_28_0	0.985651	0.988336
quick_busybox_1_28_0	0.253179	0.249201
baital_ecos-icse11	0.975684	0.970069
waps_ecos-icse11	0.790062	0.784191
quick_ecos-icse11	0.282384	0.273483
baital_mpc50	0.980516	0.975899
waps_mpc50	0.797631	0.806473
quick_mpc50	0.538494	0.531343
baital_phycore	0.965864	0.962150
waps_phycore	0.774117	0.769050
quick_phycore	0.513699	0.506364
baital_psim	0.975527	0.968054
waps_psim	0.792510	0.785722
quick_psim	0.494380	0.484784

TABLE 2: Approximation of $\mathsf{FracCov_2}(\mathcal{U}, \mathsf{F})$. Confidence parameters are $\delta = 0.0975$, $\varepsilon = 0.105$.

15 for $t = 5$, and only 8 for $t = 6$ within 28800 seconds. Comparison of approximation with $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ showed the largest error of 0.0103 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 almost all benchmarks up to $t = 6$, whereas BLMMaxCov timed out on several benchmarks even with a large time budget on $t = 2$, thus allowing us to give a positive 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 2 and the histogram of errors is shown in Figure 8. The largest error was 0.035 which is smaller than the derived ε.

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 3 for $t = 6$, while existing methods succeeded only on 23, thus giving us a positive answer to **RQ5**.

5.6 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 provided theoretical proofs that bound the potential error, and in the experiments we ran each benchmark multiple times. In the obtained results the difference between multiple runs was below 0.07 for ApproxCov and below 0.02 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 [67], [68], [69], [64], [58], [70]. These benchmarks cover a wide range in the number of variables, clauses, and configurations.

6 Generalization of the Algorithms

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. Results for empirical evaluation are provided at the end of the section.

6.1 Generalization of ApproxCov

We start with the formalization of the problem for the general case. Let $X = \{x_1, \ldots, x_n\}$ be the set of all the variables (corresponding to n features) where each x_i takes values in an alphabet Σ_i with d_i elements. A *configuration* σ is a string of the form $\alpha_1 \cdots \alpha_n$ where $\alpha_i \in \Sigma_i$. Let $\mathcal{C}$ denote the set of all configurations over $\Sigma_1, \cdots, \Sigma_n$. We call the alphabet size sequence $\vec{d} = \langle d_1, \ldots, d_n \rangle$ the *signature of* $\mathcal{C}$.

Given a configuration $\sigma = \alpha_1 \cdots \alpha_n$, the *t-wise coverage* of σ is defined as

$$\mathsf{Cov_t}(\sigma) = \{\langle (i_1, \alpha_{i_1}), \cdots (i_t, \alpha_{i_t}) \rangle \mid 1 \leq i_1 < \cdots < i_t \leq n\}.$$

Note that for any configuration σ, $|\mathsf{Cov_t}(\sigma)| = \binom{|X|}{t} = \binom{n}{t}$. As in the binary case, for a set $\mathcal{U} \subseteq \mathcal{C}$ of configurations, we can extend the notion of $\mathsf{Cov_t}$ to as

$$\mathsf{Cov_t}(\mathcal{U}) = \bigcup_{\sigma \in \mathcal{U}} \mathsf{Cov_t}(\sigma)$$

.

Let $\Omega = \mathsf{Cov_t}(\mathcal{C})$ denote the set of all t-wise coverage tuples. Computing the size of $\mathsf{Cov_t}(\mathcal{C})$ is a bit more involved than in the binary case. We have the following proposition.

Proposition 6.1. *For any set of n variables $X = \{x_1, \ldots, x_n\}$ where each x_i takes values in Σ_i with size d_i, and for any t where $1 \leq t \leq n$;*

1)

$$|\Omega| = \sum_{I \subseteq [n], |I|=t} \prod_{i \in I} d_i$$

2) For any $\mathcal{U} \subseteq \mathcal{C}$ and any $1 \leq t \leq n$,

$$\binom{n}{t} \leq |\mathsf{Cov_t}(\mathcal{U})| \leq |\Omega|$$

ApproxCovGeneral is the generalization of the algorithm ApproxCov for the non-binary case. It takes as input a set $\mathcal{U} \subseteq \mathcal{C}$, a number of variables n, a signature vector $\vec{d} = \langle d_1, \cdots, d_n \rangle$, a coverage parameter t, 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 $(1+\epsilon)|\mathsf{Cov_t}(\mathcal{U})|$.

There are two elements of the original algorithm that require modification. The first one is the computation of $|\Omega|$. We fill in a 1-based $n \times t$ matrix N[9] with the $N_{i,j}$ entry containing the size of the set of all j-wise tuples for variables $x_1, \cdots, x_i$ with alphabet size $d_1, \cdots, d_i$. Matrix cells have the following recurrence:

$$N_{i,j} = N_{i-1,j} + d_i \cdot N_{i-1,j-1}.$$

The value in the cell $N_{n,t}$ is equal to $|\Omega|$. The above recursion can be used to develop a dynamic programming algorithm to fill in the entries of the matrix N.

The second element is the selection of random t-wise coverage tuples from Ω. In order to ensure uniformity, the selection of t variables must consider the correct probabilities that are proportional to the number of tuples involving each variable. Instead of computing all these probabilities, we propose an algorithm Sample that reuses the matrix N and selects an ordered t-wise coverage tuple with the order imposed by variable indexes. First, the last variable of the tuple is selected. For any i, $N_{i,t}$ is the number of t-sized tuples with variables $x_1, \ldots, x_i$, and therefore $N_{i,t} - N_{i-1,t}$ is the number of t-sized tuples with the last variable x_i. These differences are the required proportions for the uniform selection. The procedure repeats for each element of the tuple considering at each step the corresponding column of the matrix N. After the selection of variables, the algorithm randomly assigns values to each selected variable.

ApproxCovGeneral pseudo-code is provided in Algorithm 3. Two procedures for the matrix N computation and for uniform sampling are shown in Algorithms 4 and 5, respectively.

Lemma 6.2 (Correctness of Sample)**.** *The algorithm* Sample(N, n, t) *samples an element uniformly at random from* Ω*.*

Proof. We use induction on the number of rounds of the outer for loop to show the correctness of the algorithm Sample(N, n, t). We need to prove that in the round j of the outer **for** loop for all $1 \leq k \leq z$, probability that k is picked is

$$\Pr_{\langle (a_1, \alpha_{a_1}), \cdots (a_t, \alpha_{a_t}) \rangle \leftarrow \Omega} [a_j = k \mid a_{j+1} = z+1].$$

9. We define N with indexes starting from 1 for the ease of presentation.

Algorithm 3 ApproxCovGeneral$(\mathcal{U}, n, \vec{d}, t, \epsilon, \delta)$

```
N ← FillMatrix(d, n, t)
|Ω| = N_{n,t}
ℓ ← ⌈3 (|Ω| / (n choose t)·ε²) ln(2/δ)⌉
Initialise M = ∅
for k = 1; k ≤ ℓ; k++ do
    w_k ← Sample(N, n, t)
    Put w_k → 0 into M
end for
for σ ∈ U do
    for k = 1; k ≤ ℓ; k++ do
        if w_k ⊆ σ then
            Update M[w_k] to 1
        end if
    end for
end for
Output (|Ω|/ℓ) Σ_{k=1}^{ℓ} M[w_k]
```

Algorithm 4 FillMatrix$(\vec{d}, n, t)$

```
Initialise n × t 1-based matrix N
N_{1,1} = d_1
for i = 2 to n do
    N_{i,1} = d_i + N_{i-1,1}
end for
for j = 2 to t do
    N_{1,j} = 0
end for
for i = 2 to n do
    for j = 2 to t do
        N_{i,j} = N_{i-1,j} + d_i N_{i-1,j-1}
    end for
end for
Output N
```

This follows because of the contents of the matrix N. Note that $N_{i,j}$ contains the size of the set of all j-wise tuples for variables $x_1, \cdots, x_i$ with alphabet size $d_1, \cdots, d_i$. Therefore, $N_{i,j} - N_{(i-1),j}$ is the number of j-wise tuples for variables $x_1, \cdots, x_i$ with alphabet size $d_1, \cdots, d_i$ such that the variable x_i is present in the tuple. Thus, for all $1 \leq k \leq z$

$$\Pr_{\langle (a_1, \alpha_{a_1}), \cdots (a_t, \alpha_{a_t}) \rangle \leftarrow \Omega} [a_j = k \mid a_{j+1} = z+1] = \frac{N_{k,j} - N_{(k-1),j}}{N_{z,j}} = \frac{p_{k,j}}{N_{z,j}},$$

and this is indeed the probability that k is picked in the round j of the outer **for** loop. □

Theorem 6.3. *Let $X = \{x_1, \ldots, x_n\}$ be the set of n variables where each x_i takes values in an alphabet Σ_i with d_i elements. Let $\mathcal{U} \subset \mathcal{C}$, where $\mathcal{C}$ is the set of all configurations over $\Sigma_1, \ldots, \Sigma_n$. Then for any positive integer t and positive real numbers $0 < \epsilon, \delta < 1$, with probability at least $(1-\delta)$ the output of* ApproxCovGeneral *is an $(1 \pm \epsilon)$-multiplicative approximation*

Algorithm 5 Sample(N, n, t)

$z = n$
Initialize $w \leftarrow \emptyset$
for $j = t$ to 1 **do**
  $p_{1,j} = N_{1,j}$
  for $i = 2$ to z **do**
    $p_{i,j} = N_{i,j} - N_{(i-1),j}$
  end for
  Pick a element $k \in \{1, \ldots, z\}$ with probability proportional to $\{p_{1,j}, \ldots, p_{z,j}\}$
  Pick a random element α_k from Σ_k
  $w = \{(k, \alpha_k)\} \cup w$
  $z = k - 1$
end for
Output w

of the $|\mathsf{Cov_t}(\mathcal{U})|$. *That is, with probability at least* $(1-\delta)$,

$$(1-\epsilon)|\mathsf{Cov_t}(\mathcal{U})| \leq \frac{|\Omega|}{\ell}\sum_{k=1}^{\ell}\mathcal{M}[w_k] \leq (1+\epsilon)|\mathsf{Cov_t}(\mathcal{U})|,$$

where $\Omega = \mathsf{Cov_t}(\mathcal{C})$ *t-wise tuples. Moreover, the amount of space needed is* $O\left(nt + \left\lceil 3\frac{|\Omega|}{\binom{n}{t}\epsilon^2}\ln(2/\delta)\right\rceil t\lceil\log_2 n\rceil + \lceil\log_2|\mathcal{U}|\rceil\right)$ *and the run time is* $O\left(nt + \left\lceil 3\frac{|\Omega|}{\binom{n}{t}\epsilon^2}\ln(2/\delta)\right\rceil t\lceil\log_2 n\rceil|\mathcal{U}|\right)$.

The proof is similar to Theorem 4.2. The additional term nt in the run time and space complexity corresponds to the computation and storage of the matrix N.

6.2 Generalization of ApproxMaxCov

Let $X = \{x_1, \ldots, x_n\}$ be the set of all the variables (corresponding to n features) where each x_i takes values in an alphabet Σ_i with d_i elements. In this work we assume that constraints over the features are encoded with Quantifier-Free Bit-Vector logic (QF_BV). A satisfying assignment of a formula F, denoted by $\sigma = \alpha_1 \ldots \alpha_n$, is an assignment of values to variables such that $\alpha_i \in \Sigma_i$ for all $1 \leq i \leq n$ and F evaluates to true. We denote the set of all satisfying assignments of F as $\mathsf{Sol}(\mathsf{F})$.

The generalization of ApproxMaxCov estimates $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ and is shown in Algorithm 6. The algorithm uses the standard reduction to the Boolean case with bit-blasting. For a QF_BV formula F on variable set X, it will first construct a new QF_BV formula $\mathsf{G^F}$ on variable set $X \cup S$, where S is an additional set of variables, such that

$$|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))| = |\mathsf{Sol}(\mathsf{G^F})_{\downarrow \mathsf{S}}| \tag{5}$$

The set S contains t bit-vector variables $y_1, \ldots, y_n$ of size $\lceil\log_2 n\rceil$ (corresponding to the groups $Y_1, \ldots, Y_n$ in ApproxMaxCov) and t variables $z_1, \ldots, z_n$ of size $\lceil\log_2 max(d_i)\rceil$. The additional constraints in $\mathsf{G^F}$ are similar to the ones described in Section 4.2. Note that the constraints use the comparison operators (EQ, LEQ, LESS) on bit-vectors that are defined in QF_BV.

The second step involves conversion of $\mathsf{G^F}$ into a CNF formula. In our implementation we use the SMT solver z3 with the following tactics: Then(simplify, bit-blast, tseitin-cnf). The final step uses an approximate projected model counting algorithm to estimate the size of $|\mathsf{Sol}(\mathsf{G^F_{CNF}})_{\downarrow \mathsf{S_{CNF}}}|$.

Algorithm 6 ApproxMaxCovGeneral$(\mathsf{F}, t, \varepsilon, \delta)$

$(\mathsf{G^F}, S) \leftarrow \mathsf{ConstructGFormulaGeneral}(F)$
$(\mathsf{G^F_{CNF}}, S_{\mathsf{CNF}}) \leftarrow \mathsf{QF\_BV2CNF}(\mathsf{G^F}, S)$
$c \leftarrow \mathsf{ApproxCount}(\mathsf{G^F_{CNF}}, S_{\mathsf{CNF}}, \varepsilon, \delta)$
return c

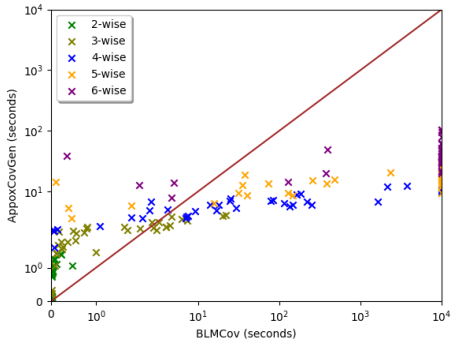

Fig. 9: Comparison of execution time of ApproxCovGeneral and BLMCov on 35 benchmarks for different values of t. Points on the right border correspond to timeouts.

The Equality 5 and the correctness of the algorithm are proved similarly to the binary case.

6.3 Empirical Results

For the empirical evaluation of the generalized algorithms we experimented with the implementation on a set of benchmarks. The research question we pose is similar to the ones in Section 5:

- **RQ6** Are algorithms ApproxCovGeneral and ApproxMaxCovGeneral fast, scalable, and provide a close approximation to the correct result?

Our benchmark suite consisted of 35 feature models and 35 sample sets from [72]. 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 ApproxCovGeneral and ApproxMaxCovGeneral. For comparison we used modified versions of BLMCov and BLMMaxCov: both have been adapted to new inputs and in the latter we replaced calls to a SAT solver with calls to an SMT solver (z3).

The timeouts were 3600 seconds for BLMCov, ApproxCovGeneral, and ApproxMaxCovGeneral, and 14400 seconds for BLMMaxCov. Memory limit was 4Gb. Both ApproxCovGeneral and ApproxMaxCovGeneral 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.

ApproxCovGeneral has successfully terminated on all benchmarks for $t \in [2, 6]$ within 110 seconds. BLMCov has failed 3 benchmarks for $t = 4$, 19 for $t = 5$, and 28 for $t = 6$. Figure 9 shows the comparison of execution times in the log scale. Note that the run time of BLMCov is proportional to $\binom{n}{t}$, where n is the number of variables, while run time of ApproxCovGeneral is proportional to $\frac{|\Omega| t\lceil\log_2 n\rceil}{\binom{n}{t}}$. Due to the low number of variables, BLMCov was faster on all benchmarks for $t = 2$ and on 26 benchmarks for $t = 3$. For higher values of t ApproxCovGeneral is faster and scales

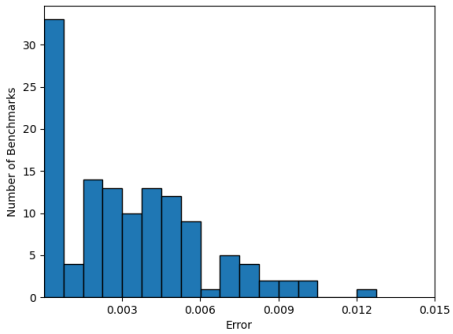

Fig. 10: Approximation error of ApproxCovGeneral for $t \in [2, 6]$ on 125 sample sets on which BLMCov have successfully terminated. The error is computed as $max(abs(result_i - |\mathsf{Cov_t}(\mathcal{U})|)/|\mathsf{Cov_t}(\mathcal{U})|)$, where $result_i$ is approximation returned by ApproxCovGeneral on the i^{th} run.

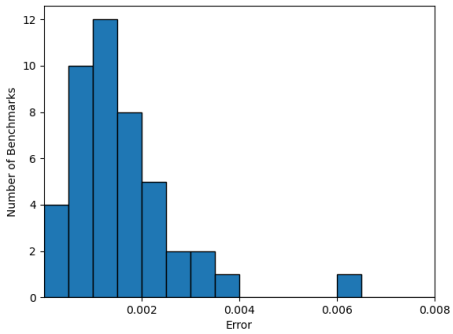

Fig. 11: Approximation error of ApproxMaxCovGeneral for $t \in [2, 4]$ on 45 benchmarks on which BLMMaxCov terminated. The error is computed as $max(abs(result_i - |\mathsf{Cov_t}(\mathsf{Sol(F)})|)/|\mathsf{Cov_t}(\mathsf{Sol(F)})|)$, where $result_i$ the approximation returned by ApproxMaxCovGeneral on i^{th} run.

better. The approximation error is shown in Figure 10, the largest value was 0.0125.

ApproxMaxCovGeneral has successfully terminated on all benchmarks for $t \in [2, 5]$ within the given timeout of 3600 seconds. 10 benchmarks for $t = 6$ have timed out. BLMMaxCov has managed to terminate on 4 benchmarks for $t = 3$ and on 3 benchmarks for $t = 4$ within 14400 seconds timeout. On all benchmarks ApproxMaxCovGeneral was at least 6 times faster than BLMMaxCov. The approximation error is shown in 11, the largest value was 0.006.

From these results we can answer **RQ6**: on the benchmarks from [72] ApproxCovGeneral and ApproxMaxCovGeneral provide approximations close to the correct value and have good scalability. However, it is important to note that on benchmarks with few variables or with large alphabets to take values from and for low values of t, ApproxCovGeneral performs slower than BLMCov.

7 Application to Adaptive Weighted Sampling

The effectiveness of the combinatorial testing depends on the quality of a test suite and t-wise coverage is one of the commonly used metrics to evaluate the test suite. High t-wise coverage is one of the major objectives for test suite generators. One of the recently proposed techniques, *adaptive weighted sampling* [58], performs the generation in multiple rounds and between rounds it explores previously generated configurations to tune up the generation for the following round. Such adaptation at run time allows the technique to achieve higher t-wise coverage than other sampling techniques such as uniform sampling. The exploration between rounds is closely related to the t-wise coverage computation, therefore fast estimation is capable of improving performance and scalability of the technique.

BAITAL [73] is a sampling tool that uses adaptive weighted sampling. The selection of configurations is guided by a distribution defined with a weight function. BAITAL follows a multi-round architecture wherein the initial weight function is uniform, and then the weight function is updated after each round. Several *strategies* to perform the update have been considered in [58], but one of the best options requires computing the t-wise fractional coverage of a generated configuration set for each literal. Due to computation complexity, this update has only been used for $t = 2$. In this section, we propose to utilize ApproxCov and ApproxMaxCov in the weight function update.

7.1 Approximation in Weight Function

Given a configuration σ and a literal x, we define t-wise coverage of σ for x by $\mathsf{Cov_t^x}(\sigma) = \{T \subseteq \sigma \mid |T| = t \wedge x \in T\}$. It is easy to see that $|\mathsf{Cov_t^x}(\sigma)| = \binom{n-1}{t-1}$ for $x \in \sigma$, where n is the number of variables. Similarly to Section 3, for a set of configurations $\mathcal{U}$ we define $\mathsf{Cov_t^x}(\mathcal{U}) = \bigcup_{\sigma \in \mathcal{U}} \mathsf{Cov_t^x}(\sigma)$ and $\mathsf{FracCov_t^x}(\mathcal{U}, \mathsf{F}) = \frac{\mathsf{Cov_t^x}(\mathcal{U})}{\mathsf{Cov_t^x(Sol(F))}}$.

Weight function $W(l)$ in BAITAL assigns a value in the interval $[0.01, 0.99]$ for each literal l such that $W(l) + W(\neg l) = 1$. Initially all weights are set to 0.5 which corresponds to the uniform distribution. BAITAL generates configurations in multiple rounds, after each round the weight function is updated. The considered strategy uses the Algorithm 7 for weight function update with $\mathcal{U}$ being the current set of configurations. Line 4 ensures that weights stay within the desired interval. The algorithm involves computation of $\mathsf{FracCov_t^x}(\mathcal{U}, \mathsf{F})$ for each literal.

Algorithm 7 $WeightUpdate(\mathcal{U}, \mathsf{F}, t)$

1: **for** $x \in Vars(\mathsf{F})$ **do**
2: $\quad \mathsf{diff} \leftarrow \mathsf{FracCov_t^x}(\mathcal{U}, \mathsf{F}) - \mathsf{FracCov_t^{\neg x}}(\mathcal{U}, \mathsf{F})$
3: $\quad W(x) \leftarrow 0.5 \times (1 - \mathsf{sign(diff)} \times \mathsf{sqrt}(|\mathsf{diff}|))$
4: $\quad W(x) \leftarrow \mathsf{max}(0.01,\ \mathsf{min}(0.99,\ W(x)))$
5: $\quad W(\neg x) \leftarrow 1 - W(x)$
6: **end for**
7: Output W

We propose a new strategy for BAITAL that replaces the computation of $\mathsf{FracCov_t^x}(\mathcal{U}, \mathsf{F})$ by its estimation with ApproxCov and ApproxMaxCov. In ApproxCov we select a single sample set (lines 3-7 of Algorithm 1) for all variables that is used in all rounds of BAITAL. This allows us to reuse the map $\mathcal{M}$ from the earlier rounds, thus map updates (lines 8-14 of Algorithm 1) are performed only on the

configurations generated during the last round. The output value is a list of values for each literal where for a literal l we use a subset of the sample set that includes l.

$|\mathsf{Cov}_t^x(\mathsf{Sol}(\mathsf{F}))|$ is estimated with ApproxMaxCov. For a literal l, we set it to *True*, simplify $\mathsf{F}[l \leftarrow True]$, and compute the approximation with ApproxMaxCov for $t-1$. $\mathsf{Cov}_t^x(\mathsf{Sol}(\mathsf{F}))$ can be obtained by adding l to all elements of $\mathsf{Cov}_{t-1}(\mathsf{Sol}(\mathsf{F}[l \leftarrow True]))$, therefore these sets have equal sizes. Note that $|\mathsf{Cov}_t^x(\mathsf{Sol}(\mathsf{F}))|$ does not change between the BAITAL rounds and it is computed a single time.

7.2 Experimental Results

We implemented the new strategy for BAITAL[10] where the computation of $\mathsf{FracCov}_t^x(\mathcal{U}, \mathsf{F})$ has been replaced with its approximation and compared the original and the new strategies. We posed the following research question:

- **RQ7** Has the proposed strategy a negative impact on the achievable 2-wise coverage in comparison with the original strategy?
- **RQ8** Is the proposed strategy faster, more scalable than the original strategy, and capable of generating test suites for $t \geq 3$?

As a benchmark set, we took 126 feature models from [58] that include the set of large benchmarks used in Section 5. The experiment has been designed as follows. As a preliminary step, for every benchmark, we computed $|\mathsf{Cov}_2^x(\mathsf{Sol}(\mathsf{F}))|$ and its estimation for every literal x in the benchmark, i.e. for each variable and for their negations. For this step, we used 18000 seconds timeout. These values do not change between BAITAL runs and their computation is slow, therefore we pre-compute them a single time and reuse them in the following steps. Additionally, since the approximation is significantly faster, the inclusion of this computation would strongly affect the time comparison between strategies. In the remaining steps, we consider the benchmarks on which both exact and approximate computations have terminated.

In the first step, we generated 10 configuration sets with both strategies for $t = 2$. We compared mean running time (excluding the preliminary computation) and achieved 2-wise coverage. In the next step, we randomly selected 10 feature models from the first step and additionally generated another 40 configuration sets with both strategies. For the comparison of the strategies, we run a Welch's t-test with a null hypothesis that their mean values of the 2-wise coverage are equal. The considered p-value is equal to 0.05. Finally, we attempted to generate configuration sets with the new strategy for $t \in [3, 6]$. We used 18000 seconds timeout for the computation of $|\mathsf{Cov}_t^x(\mathsf{Sol}(\mathsf{F}))|$ and 18000 seconds timeout for the following configuration sets generation.

The original strategy failed to finish the computation of $|\mathsf{Cov}_2^x(\mathsf{Sol}(\mathsf{F}))|$ for all literals x within the timeout on 11 benchmarks while the new strategy failed only on a single benchmark. On all except the 2 smallest benchmarks the new strategy was faster; the speedup factor was above 5 on 106 benchmarks.

The results of the configuration set generation are the following. The run time of two strategies excluding the computation of $|\mathsf{Cov}_2^x(\mathsf{Sol}(\mathsf{F}))|$ was close: the relative difference of mean time was below 0.05 on 95% of benchmarks with the approximated version being faster on 60% of benchmarks. Mean time on different benchmarks is shown in Figure 12. Note that considering the full time including the computation of $|\mathsf{Cov}_2^x(\mathsf{Sol}(\mathsf{F}))|$, the new strategy is at least 2 times faster than the original one on all but 4 benchmarks.

10. https://github.com/meelgroup/baital

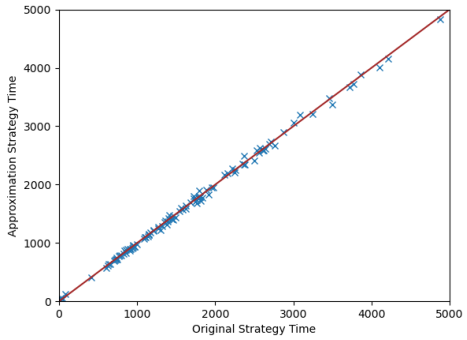

Fig. 12: Mean time comparison of two BAITAL strategies for $t = 2$. The time does not include the computation of $|\mathsf{Cov}_2^x(\mathsf{Sol}(\mathsf{F}))|$.

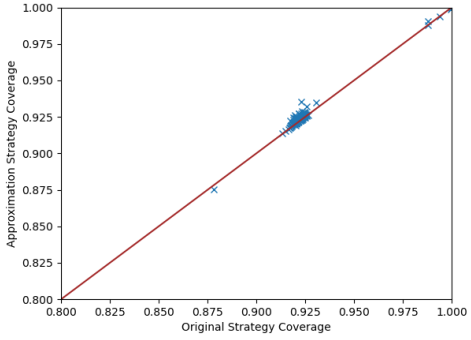

Fig. 13: Mean coverage comparison of two BAITAL strategies.

The comparison of mean and maximal 2-wise coverages between 10 runs showed the relative difference between the two strategies below 0.025 on all benchmarks. Moreover, the strategy with approximation achieved slightly higher mean and maximal coverages on 80% of benchmarks. The mean coverages on different benchmarks are illustrated in Figure 13.

On 10 randomly selected benchmarks we ran a Welch's t-test with 50 configuration sets. On 6 benchmarks the null hypothesis has not been rejected, on the remaining benchmarks the strategy with approximation had a slightly higher mean value by 0.002-0.004.

The final step is the generation of configuration sets with the new strategy for $t \geq 3$. It has successfully terminated on 123 benchmarks for $t = 3$, 122 benchmarks for $t = 4$, 116 benchmarks for $t = 5$, and only 6 benchmarks for $t = 6$. The original strategy and the computation of $\mathsf{FracCov}_t(\mathcal{U}, \mathsf{F})$ time out, therefore we cannot compare these results with the original strategy, but can estimate the coverage with ApproxCov and ApproxMaxCov algorithms. We used $\epsilon = \delta = 0.05$ for both algorithms. On all benchmarks except 1 mean t-wise coverage is above 0.8. However, we noticed that with higher values of t the achieved coverage may significantly

differ between runs. The interquartile range for the coverage was above 0.01 on the majority of benchmarks for $t = 5$ with the highest value 0.067 and the difference between the best and the worst results exceeded 0.1 on 17 benchmarks.

From the experiment we can negatively answer **RQ7**: the 2-wise coverage obtained with the new strategy for BAITAL utilizing ApproxCov and ApproxMaxCov is close to the coverage obtained with the original BAITAL strategy on the majority of benchmarks and is statistically indistinguishable on 6 out of 10 benchmarks, moreover on several benchmarks the new strategy achieved better results. The new strategy is capable of generating sample sets for $t = 5$ on almost all benchmarks, while the original one times out for $t = 3$, thus giving us a positive answer to **RQ8**.

8 COMPARISON OF SAMPLING TOOLS

Multiple tools exist for building a test suite for combinatorial testing that use some form of random sampling. Authors of each tool perform comparisons with competitors. However, due to the computation complexity, the comparison is always limited to $t = 2$ or 3. The efficient approximation of t-wise coverage allows us to compare different tools on higher values of t. In this experiment we used a set of benchmarks and compared achievable t-wise coverage of publicly available tools on different values of t. We selected a set of tools and used their recommended parameters unless stated otherwise:

- WAPS [57] set up for uniform sampling.
- CMSGen [74].
- LS-Sampling [59].
- SamplingCA [75] modified to output a specified number of configurations. This tool builds a sample set with $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F}) = 1$ in two phases: sampling and full covering. For this comparison, we stopped the tool after the first phase.
- Baital [58] with several sets of parameters. We used two sampling strategies 3 and 5 and two sampling engines WAPS and CMSGen resulting in 4 configurations that we denote with Baital[3|5][W|C]. Strategy 3 is presented in Section 7 and its output depends on the value of t, therefore we generated 10 configuration sets for each value of t with this strategy. For the CMSGen engine, we used an additional option "–extra-samples" that generates a larger number of configurations and selects the best ones among them.

We used the 123 largest benchmarks from Section 5 and generated 10 configuration sets with 1000 configurations for each benchmark with each tool. We compared $|\mathsf{Cov_t}(\mathcal{U})|$ on different values of t; for $t = 2$ we used BLMCov that provides exact value and for $t \in [3, 6]$ we estimated the value with ApproxCov. Initially we used $\delta = \varepsilon = 0.05$, however in many cases the obtained coverages are close and we recomputed the coverage with parameters set to 0.01 for all tools except WAPS and Baital with the WAPS engine.

Note that since $|\mathsf{Cov_t}(\mathsf{Sol}(\mathsf{F}))|$ is computed on a formula representing benchmark constraints its value is independent from the sampling tool. Therefore, the comparison on $|\mathsf{Cov_t}(\mathcal{U})|$ is valid for $\mathsf{FracCov_t}(\mathcal{U}, \mathsf{F})$ as well.

In this comparison, we are interested in achievable t-wise coverage and do not report the results for the run time. For $t = 2$ we computed an exact value of $|\mathsf{Cov_t}(\mathcal{U})|$ for the generated configuration sets and for $t \geq 3$ we computed an approximation with ApproxCov. In the comparison we considered an interval for each benchmark and each tool that is expected to contain all 10 values of $|\mathsf{Cov_t}(\mathcal{U})|$. For $t = 2$ we took the minimum and maximum values of $|\mathsf{Cov_t}(\mathcal{U})|$ since the exact values are computed. For higher values of t, the interval is $[min_{i\in[1,10]}(result_i/(1+\varepsilon)), max_{i\in[1,10]}(result_i/(1-\varepsilon))]$, the additional factor is used to include the approximation error: each value of $|\mathsf{Cov_t}(\mathcal{U})|$ is in the interval with the probability at least $1 - \delta$. We say that one tool has better t-wise coverage than another on a benchmark if the left endpoint of its computed interval is greater than the right endpoint of the second tool interval.

In the second step, we randomly selected 10 benchmarks and computed 40 additional configuration sets for these benchmarks with each tool. For each pair of tools, for each t, and for each benchmark, we run a Welch's t-test with a null hypothesis that their mean values of the t-wise coverage are equal. For each sample there were 50 independent observations, however we could not assume that variances for different tool samples are identical, therefore the Welch's t-test was selected. We used a p-value equal to 0.05.

8.1 Comparison Results

Out of 123 benchmarks the sampling did not terminate after 12 hours on 1 benchmark with LS-Sampler, 2 benchmarks for Baital3C, and 3 benchmarks with Baital3W. Other tools successfully generated 10 configuration sets for all benchmarks.

Uniform sampling with WAPS achieved the worst results, it had worse t-wise coverage than any other tool on at least 117 out of 123 benchmarks. The remaining tools can be divided into 3 groups. The first group includes 2 versions of Baital with the WAPS engine. For $t = 2$, Baital5W was better than Baital3W, however we cannot distinguish them on almost all benchmarks for $t \geq 3$. Other tools got better coverage than Baital with the WAPS engine. Baital5C and LS-Sampling also showed close results and are indistinguishable for $t \geq 3$. The remaining 3 tools showed the best results. For $t = 2$, SamplingCA had the best coverage on almost all benchmarks, yet for $t \geq 3$ the results were close. Only for $t = 6$, it had better coverage than CMSGen. SamplingCA was indistinguishable from Baital3C for $t \geq 3$. Table 3 shows the number of benchmarks each tool has better coverage than other ones for t equal 2 and 6.

Note that Baital3W and Baital3C attempt to adapt the selection of configurations to a particular value of t. This adaptation explains their relative performance improvement with the increase of t. For example, for $t = 2$ Baital3C is worse than LS-Sampling on 70% of benchmarks, while for $t = 6$ it is better on almost all benchmarks.

Welch's t-test results provide additional details to the comparison. The number of benchmarks with rejected null hypothesis for t equal 2 and 6 is shown in Table 4. Uniform sampling has the lowest mean value. The null hypothesis can be rejected on the majority of benchmarks for Baital5W and Baital3W, however we can notice that the former is expected to achieve higher coverage on low values of t while the latter works better on higher values of t. Two other tools LS-Sampling and Baital5C that were indistinguishable

t=2	Uniform	Baital3W	Baital5W	Baital5C	LS-Sampling	CMSGen	Baital3C	SamplingCA
Uniform		2	0	0	1	0	1	0
Baital3W	117		1	0	0	0	0	0
Baital5W	121	118		0	1	0	1	0
Baital5C	122	122	121		5	2	2	0
LS-Sampling	121	121	121	56		2	87	0
CMSGen	122	122	122	36	5		59	0
Baital3C	121	121	121	0	4	1		0
SamplingCA	122	122	122	121	122	121	122	

t=6	Uniform	Baital3W	Baital5W	Baital5C	LS-Sampling	CMSGen	Baital3C	SamplingCA
Uniform		3	0	0	1	0	2	0
Baital3W	119		1	0	0	0	0	0
Baital5W	120	3		0	1	0	2	0
Baital5C	121	120	118		3	1	2	0
LS-Sampling	120	119	117	0		1	1	0
CMSGen	121	120	117	116	115		2	0
Baital3C	119	118	118	116	117	2		0
SamplingCA	121	120	119	117	119	115	2	

TABLE 3: Number of benchmarks a tool in a row has higher coverage than a tool in a column for $t = 2$ and 6.

t=2	Uniform	Baital5W	Baital3W	Baital5C	LS-Sampling	CMSGen	Baital3C	SamplingCA
Uniform		0	0	0	0	0	0	0
Baital5W	10		9	0	0	0	0	0
Baital3W	10	0		0	0	0	0	0
Baital5C	10	10	10		0	0	1	0
LS-Sampling	10	10	10	7		0	6	0
CMSGen	10	10	10	6	0		6	0
Baital3C	10	10	10	1	0	0		0
SamplingCA	10	10	10	7	0	0	8	

t=6	Uniform	Baital5W	Baital3W	Baital5C	LS-Sampling	CMSGen	Baital3C	SamplingCA
Uniform		0	0	0	0	0	0	0
Baital5W	10		1	0	0	0	0	0
Baital3W	10	6		0	0	0	0	0
Baital5C	10	10		0	0	0	0	
LS-Sampling	10	10	10	10		0	0	0
CMSGen	10	10	10	10	10		0	0
Baital3C	10	10	10	10	10	10		0
SamplingCA	10	10	10	10	10	10	10	

TABLE 4: Number of benchmarks null hypothesis can be rejected.

with a conservative comparison of intervals have the null hypothesis rejected on almost all benchmarks. Among the tools in the last group, SamplingCA has the highest expected t-wise coverage, while the choice between Baital3C and CMSGen should be based on the value of t. In 97% of cases where we rejected the null hypothesis p-value was below 0.01, so the results are valid even with a higher significance level.

The experiment shows that we can obtain a comparison of different sampling tools on higher values of t with ApproxCov.

9 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 test 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 for a given set of tests within $(1\pm\varepsilon)$-factor of the ground truth with probability at least $1-\delta$ for given ε and δ; (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 ε and δ. The approach has 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 the estimation of 6-wise coverage of features. We also extended the frameworks to also include non-binary domains. We demonstrate how the algorithms can be used in test suite generation and improve its scalability. Finally, we showed that our work opens the possibility to compare various test set generators by estimating the maximum achievable t-wise coverage. For future work, we would like to apply our approximation algorithms to improve scalability of other test scenario generators, for example apply them to Feature-based Context-oriented Programming [76].

References

[1] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on software engineering*, no. 1, pp. 1–9, 1976.

[2] I. Rodrigues, M. Ribeiro, F. Medeiros, P. Borba, B. Fonseca, and R. Gheyi, "Assessing fine-grained feature dependencies," *Inf. Softw. Technol.*, vol. 78, pp. 27–52, 2016. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.05.006

[3] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," *NIST special Publication*, vol. 800, no. 142, p. 142, 2010.

[4] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, 2013, pp. 1–8.

[5] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[6] T. Pett, T. Thüm, T. Runge, S. Krieter, M. Lochau, and I. Schaefer, "Product sampling for product lines: The scalability challenge."

[7] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[8] K. Tatsumi, "Test case design support system," in *Proc. International Conference on Quality Control (ICQC'87)*, 1987, pp. 615–620.

[9] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[10] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to combinatorial testing*. CRC press, 2013.

[11] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 1–45, 2014.

[12] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "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, 2015, pp. 540–550.

[13] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 643–654.

[14] C. Yilmaz, S. Fouché, M. B. Cohen, A. A. Porter, G. Demiröz, and U. Koc, "Moving forward with combinatorial interaction testing," *Computer*, vol. 47, no. 2, pp. 37–45, 2014.

[15] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pp. 38–48.

[16] P. Martou, B. Duhoux, K. Mens, and A. Legay, "Beyond combinatorial interaction testing: On the need for transition testing in dynamically adaptive context-aware systems," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 100–104.

[17] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 421–432.

[18] G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *IEEE Transactions on Information Theory*, vol. 34, no. 3, pp. 513–522, 1988.

[19] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[20] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog/ipog-d: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[21] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing, Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

[22] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[23] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Annual Asian Computing Science Conference*. Springer, 2004, pp. 320–329.

[24] E. Baranov, S. Chakraborty, A. Legay, K. S. Meel, and V. N. Variyam, "A scalable t-wise coverage estimator," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 36–47.

[25] L. Stockmeyer, "The complexity of approximate counting," in *Proc. of STOC*, 1983, pp. 118–126.

[26] C. P. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using XOR constraints," in *Proc. of NIPS*, 2007, pp. 670–676.

[27] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A Scalable Approximate Model Counter," in *Proc. of CP*, 2013, pp. 200–216.

[28] ——, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls," in *Proc. of IJCAI*, 2016.

[29] M. Soos and K. S. Meel, "Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting," in *Proc. of AAAI*, 2019.

[30] M. Soos, S. Gocht, and K. S. Meel, "Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling," in *International Conference on Computer Aided Verification (CAV)*, 2020.

[31] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "Incling: efficient product-line testing using incremental pairwise sampling," *ACM SIGPLAN Notices*, vol. 52, no. 3, pp. 144–155, 2016.

[32] N. J. Sloane, "Covering arrays and intersecting codes," *Journal of combinatorial designs*, vol. 1, no. 1, pp. 51–63, 1993.

[33] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, 2011, pp. 1–5.

[34] A. Cmyrev and R. Reissing, "Efficient and effective testing of automotive software product lines," *King Mongkuts University of Technology North Bangkok International Journal of Applied Science and Technology*, vol. 7, no. 2, pp. 53–57, 2014.

[35] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *Proceedings of the 17th international software product line conference*, 2013, pp. 227–235.

[36] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "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, 2007, pp. 549–556.

[37] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 614–624.

[38] Z. Wang, B. Xu, and C. Nie, "Greedy heuristic algorithms to generate variable strength combinatorial test suite," in *2008 The Eighth International Conference on Quality Software*. IEEE, 2008, pp. 155–160.

[39] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*, 2012, pp. 46–55.

[40] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "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, 2010, pp. 459–468.

[41] J. D. McCaffrey, "Generation of pairwise test sets using a genetic algorithm," in *2009 33rd annual IEEE international computer software and applications conference*, vol. 1. IEEE, 2009, pp. 626–631.

[42] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete applied mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

[43] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *International Conference on Combinatorial Optimization and Applications*. Springer, 2010, pp. 51–64.

[44] A. Calvagna and A. Gargantini, "A formal logic approach to constrained combinatorial testing," *Journal of Automated Reasoning*, vol. 45, no. 4, pp. 331–358, 2010.

[45] J. H. Kukula and T. R. Shiple, "Building circuits from relations," in *Proc. of CAV*, 2000.

[46] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. of DAC*, 2001.

[47] N. Kitchen, "Markov chain monte carlo stimulus generation for constrained random simulation," Ph.D. dissertation, UC Berkeley, 2010.

[48] W. Wei and B. Selman, "A new approach to model counting," in *Proc. of SAT*, 2005.

[49] R. Karp and M. Luby, "Monte-carlo algorithms for enumeration and reliability problems," *Proc. of FOCS*, 1983.

[50] R. M. Karp, M. Luby, and N. Madras, "Monte-Carlo approximation algorithms for enumeration problems," *Journal of Algorithms*, vol. 10, no. 3, pp. 429–448, 1989.
[51] R. Dechter, K. Kask, E. Bin, and R. Emek, "Generating Random Solutions for Constraint Satisfaction Problems," in *Proc. of AAAI/IAAI*, 2002, pp. 15–21.
[52] V. Gogate and R. Dechter, "A New Algorithm for Sampling CSP Solutions Uniformly at Random," in *Proc. of CP*, 2006, pp. 711–715.
[53] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of sat solutions for testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 549–559.
[54] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A Scalable and Nearly Uniform Generator of SAT Witnesses," in *Proc. of CAV*, 2013, pp. 608–623.
[55] ——, "Balancing scalability and uniformity in SAT witness generator," in *Proc. of DAC*, 2014, pp. 1–6.
[56] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge compilation meets uniform sampling," in *Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 11 2018, pp. 620–636.
[57] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "Waps: Weighted and projected sampling," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2019.
[58] E. Baranov, A. Legay, and K. S. Meel, "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*, 2020, pp. 1114–1126.
[59] C. Luo, B. Sun, B. Qiao, J. Chen, H. Zhang, J. Lin, Q. Lin, and D. Zhang, "Ls-sampling: An effective local search based sampling approach for achieving high t-wise coverage," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1081–1092.
[60] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410–421, 1979.
[61] A. Durand, M. Hermann, and P. G. Kolaitis, "Subtractive reductions and complete problems for counting complexity classes," *Theoretical Computer Science*, vol. 340, no. 3, pp. 496–513, 2005.
[62] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *Proc. of AAAI*, 2006, pp. 54–61.
[63] K. S. Meel and S. Akshay, "Sparse hashing for scalable approximate model counting: Theory and practice," in *Proc. of LICS*, 2020.
[64] J. Oh, P. Gazzillo, and D. S. Batory, "*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*, 2019, pp. 15:1–15:4.
[65] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
[66] M. Soos and K. S. Meel, "Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1592–1599.
[67] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "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, 2017, pp. 291–302.
[68] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "Sat-based analysis of large real-world feature models is easy," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 91–100.
[69] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "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, 2019, pp. 240–251.
[70] K. S. Meel, "Model counting and uniform sampling instances," May 2020.
[71] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proc. of ICSE*, 2018.
[72] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *2009 1st International Symposium on Search Based Software Engineering*. IEEE, 2009, pp. 13–22.
[73] E. Baranov and A. Legay, "Baital: an adaptive weighted sampling platform for configurable systems," in *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*, 2022, pp. 46–49.
[74] P. Golia, M. Soos, S. Chakraborty, and K. S. Meel, "Designing samplers is easy: The boon of testers," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 8 2021.
[75] C. Luo, Q. Zhao, S. Cai, H. Zhang, and C. Hu, "SamplingCA: effective and efficient sampling-based pairwise testing for highly configurable software systems," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1185–1197.
[76] P. Martou, K. Mens, B. Duhoux, and A. Legay, "Test scenario generation for feature-based context-oriented software systems," *Journal of Systems and Software*, vol. 197, p. 111570, 2023.

ACKNOWLEDGMENTS

This work was supported by Walloon Region (Belgium) under convention 8560. Computational resources have been provided by the supercomputing facilities of the Université catholique de Louvain (CISM/UCL) and the Consortium des Équipements de Calcul Intensif en Fédération Wallonie Bruxelles (CÉCI) funded by the Fond de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under convention 2.5020.11 and by the Walloon Region.