A Scalable t-wise Coverage Estimator

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

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

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

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

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

ABSTRACT

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

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

KEYWORDS

Configurable software, t-wise coverage, Approximation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

1 INTRODUCTION

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

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

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

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

Given the practical importance of the combinatorial testing, the problem of efficient design of the test suite has witnessed a sustained interest for over three decades, evidenced by the diverse set of techniques ranging from evolutionary algorithms [6, 14, 37] to constraint-based systems [12, 17] proposed over the years. The proposal of these techniques is often accompanied by measurement of the benefits of the recent progress in the development of hashing-based techniques [9, 11, 58, 59]. The scalability of \(\varepsilon\)-covering array in the literature, i.e., \(1−\delta\) of the ground truth with probability at least \(1−\delta\), ApproxMaxCov reduces computation of \(\text{Cov}(\text{Sol}(F))\) to the problem of projected model counting. Our reduction allows us to build on the recent advances in hashing-based paradigm for projected model counting [9, 11, 24, 59, 60], and we employ state-of-the-art hashing-based approximate model counter ApproxMC4 [58].

\(^1\)defined formally in Section 2

\(^2\)An alternate approach would be to enumerate all the solutions of \(F\), but for most formulas of interest, the number of solutions is too large to enumerate.

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

1. A scalable Monte Carlo-based algorithm ApproxCov that takes in a test suite \(U\), a coverage parameter \(t\), a tolerance parameter \(\varepsilon\), a confidence parameter \(\delta\) as input, and returns an estimate of \(\text{Cov}(\text{Sol}(F))\) that is mathematically guaranteed to be within \((1±\varepsilon)\)-factor of the ground truth with probability at least \(1−\delta\), where \(\text{Cov}(\text{Sol}(F))\) is the set of all t-wise coverages of elements in \(U\) and \(|\cdot|\) is a set cardinality. ApproxCov takes only \(O(2^t \cdot t \log n)\) (for a constant \(t\) and \(\delta\) space in contrast to the space requirement of \(O(2^n)\) for the existing techniques. Therefore, for small \(t<6\), we achieve a reduction from \(O(n^t)\) to \(O(t \log n)\). The running time of the algorithm is also scalable for instances with parameters that are currently used in practice.

2. A scalable counting-based algorithm ApproxMaxCov that takes in formula \(F\), a coverage parameter \(t\), a tolerance parameter \(\varepsilon\), and a confidence parameter \(\delta\) as input, and returns an estimate of \(\text{Cov}(\text{Sol}(F))\) that is guaranteed to be within \((1±\varepsilon)\)-factor of the ground truth with probability at least \(1−\delta\). ApproxMaxCov achieves speedup over prior state of the art techniques by \(2\) to 136 factor speedup over prior state of the art BLMCov, while ApproxMaxCov achieves from 6 to 86 factor speedup over prior state of the art technique BLMMaxCov.

3. We demonstrate the effectiveness of ApproxCov and ApproxMaxCov via implementations in Python\(^3\) and a comprehensive experimental study. We observe that while the current state of the art techniques fail to compute coverage for beyond \(t=2\), ApproxCov and ApproxMaxCov can efficiently handle \(t\in\{2,3,4,5,6\}\) (and beyond \(t=6\)). Furthermore, for \(t=2\) on feature models with thousands of features, we observe significant runtime improvement: in particular, ApproxCov achieves from 2 to 136 factor speedup over prior state of the art BLMCov, while ApproxMaxCov achieves from 6 to 86 factor speedup over prior state of the art technique BLMMaxCov.

4. We show generalizations of ApproxCov and ApproxMaxCov to estimate t-wise coverage on configurable systems where each feature can take values from a discrete domain. We demonstrate with experimental evaluation that the generalized algorithms are effective and can provide close estimation of t-wise coverage.

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

\(^3\)https://github.com/meelgroup/approxcov

**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. As an immediate impact, we expect our techniques to allow a thorough comparison of the recently proposed techniques in CIT community \cite{3,39} for large instances, which was not feasible with the state of the art. The fact that our algorithms are grounded on guarantees that are mathematically proven makes them an attractive toolset in downstream applications.

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

## 2 NOTATIONS AND PRELIMINARIES

### 2.1 Boolean Formulas

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

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

\[
F = (x_1 \lor x_3) \land (\lnot x_1 \lor \lnot x_3) \land (x_1 \lor \lnot x_2) \land (x_3 \lor \lnot x_4).
\]

\( \text{Sol}(F) = \{0010, 0011, 1000, 1001\} \). In the literal representation the assignment 0010 is represented as \( \{\lnot x_2, x_3, \lnot x_4\} \). Let \( S = \{x_1, x_2\} \), then \( |\text{Sol}(F)|_S = |\{00, 10, 11\}| \).

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

\[
\Pr\left(1 - \epsilon \mid |\text{Sol}(F)|_S \leq c \leq (1 + \epsilon) |\text{Sol}(F)|_S\right) \geq 1 - \delta.
\]

### 2.2 t-wise Coverage

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

For example, for \( X = \{x_1, x_2, x_3\} \), \( \sigma = \{x_1, \lnot x_2, x_3\} \) is an example of a configuration.

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

**Observation 2.1.** For any \( U \neq \emptyset \) over a set of variables \( X \) and any \( 1 \leq t \leq |X| \),

\[
\frac{|X|}{t} \leq |\text{Cov}_t(U)| \leq 2^t \binom{|X|}{t}.
\]

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 \( U \subseteq \text{Sol}(F) \), the \( t \)-wise fractional coverage of a set \( U \) with respect to a formula \( F \), denoted by \( \text{FracCov}_t(U, F) \) is defined as follows:

\[
\text{FracCov}_t(U, F) = \frac{|\text{Cov}_t(U)|}{|\text{Sol}(F)|_X}
\]

**Example.** To illustrate the notions, let us consider again the CNF formula \( F \)

\[
F = (x_1 \lor x_3) \land (\lnot x_1 \lor \lnot x_3) \land (x_1 \lor \lnot x_2) \land (x_3 \lor \lnot x_4).
\]

The following table lists \( \text{Cov}_t(\text{Sol}(F)) \) for \( t = 2 \). For compactness, we use the bit representation of the assignments and coverage. For example, a set of literals \( \{\lnot x_2, \lnot x_3\} \) is listed as 00 in the column indexed \((1, 2) \). \( |\text{Cov}_t(\text{Sol}(F))|_{(1, 2)} = 17 \). For \( U = \{0010, 0011\} \) (shaded in the table), \( |\text{Cov}_t(U)|_X = 9 \), \( \text{FracCov}_t(U, F) = 9/17 \).

<table>
<thead>
<tr>
<th>2-tuples</th>
<th>( (1, 2) )</th>
<th>( (1, 3) )</th>
<th>( (1, 4) )</th>
<th>( (2, 3) )</th>
<th>( (2, 4) )</th>
<th>( (3, 4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol(F)</td>
<td>0010</td>
<td>0011</td>
<td>1000</td>
<td>1001</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>For ( U )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Coverage for example formula \( F \)
3 RELATED WORK

3.1 Combinatorial Testing

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

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

3.2 Model Counting

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

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

3.3 \(t\)-wise Coverage Estimation

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

4 ALGORITHMS

In this section we present two algorithms. The first algorithm ApproxCov (presented in Section 4.1) estimates \(\text{Cov}_t(\mathcal{U})\). More precisely ApproxCov takes as input a set \(\mathcal{U} \subset \prod_{i=1}^n \{x_i, \neg x_i\}\), error parameter \(0 < \varepsilon < 1\), and a confidence parameter \(0 < \delta < 1\), and outputs a number that, with probability at least \((1 - \delta)\), is between \((1 - \varepsilon)\text{Cov}_t(\mathcal{U})\) and \((1 + \varepsilon)\text{Cov}_t(\mathcal{U})\).

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

Using ApproxCov and ApproxMaxCov we can estimate the value of \(\text{FracCov}_t(\mathcal{U}, F)\). Given a Boolean formula \(F\) and a set \(\mathcal{U} \subset \text{Sol}(F)\), say we use \(\text{ApproxCov}(\mathcal{U}, \varepsilon_1, \delta_1)\) and \(\text{ApproxMaxCov}(F, \varepsilon_2, \delta_2)\) to obtain estimates for \(\text{Cov}_t(\mathcal{U})\) and \(\text{Cov}_t(\text{Sol}(F))\) respectively. Let \(\text{Out}_{1}\) be the output of \(\text{ApproxCov}(\mathcal{U}, \varepsilon_1, \delta_1)\) and \(\text{Out}_{2}\) be the output of \(\text{ApproxMaxCov}(F, \varepsilon_2, \delta_2)\). So we have with probability at least \((1 - \delta_1)\) the following Equation 1 and with probability \((1 - \delta_2)\) the Equation 2 holds.

\[
\begin{align*}
(1 - \varepsilon_1)\text{Cov}_t(\mathcal{U}) & \leq \text{Out}_{1} \leq (1 + \varepsilon_1)\text{Cov}_t(\mathcal{U}), \\
(1 - \varepsilon_2)|\text{Cov}_t(\text{Sol}(F))| & \leq \text{Out}_{2} \leq (1 + \varepsilon_2)|\text{Cov}_t(\text{Sol}(F))|.
\end{align*}
\]

While beta version of ApproxMC5 is released, ApproxMC4’s developer recommend usage of ApproxMC4

We use the implementation available at https://github.com/meelgroup/baital
So, by union bound, with probability at least \((1 - \delta_1 - \delta_2)\) both the above equations work. Thus with probability \((1 - \delta_1 - \delta_2)\) we have

\[
\left(1 - \epsilon_1\right)\frac{|\text{Cov}(U)|}{|\text{Cov}(\text{Sol}(F))|} \leq \frac{\text{Out}_1}{\text{Out}_2} \leq \left(1 + \epsilon_1\right)\frac{|\text{Cov}(U)|}{|\text{Cov}(\text{Sol}(F))|}.
\]

(3)

Thus given a Boolean formula \(F\), a set \(U \subseteq \text{Sol}(F)\), an error parameter \(\epsilon\) and a confidence parameter \(\delta\), if we set \(\epsilon_1, \epsilon_2, \delta_1\) and \(\delta_2\) appropriately we will be able to give an \((1 \pm \epsilon)\)-multiplicative estimate of \(\text{FracCov}(U, F)\) with probability \((1 - \delta)\). For example, let us set \(\epsilon_1 = 1/2\) and \(\epsilon_2 = \delta_1 = \delta_2 = \delta/2\). 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 \(\text{ApproxCov}(U, \epsilon_1, \delta_1)\) and \(\text{ApproxCov}(U, \epsilon_2, \delta_2)\) to estimate \(|\text{Cov}(U)|\) and \(|\text{Cov}(\text{Sol}(F))|\) respectively, from Equation 3, we have with probability at least \((1 - \delta)\)

\[
(1 - \epsilon)\text{FracCov}(U, F) \leq \frac{\text{Out}_1}{\text{Out}_2} \leq (1 + \epsilon)\text{FracCov}(U, F).
\]

4.1 Counting the Coverage of a Test Suit

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

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

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

where \(\binom{\left\{\{i\}\right\}}{t}\) is the set of all subsets of the set \(\{1, \ldots, n\}\) of size \(t\).

Samples are stored in a map \(M\), where the values indicate whether the sample is realizable by at least one test in \(U\), 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 \(M\), in other words, \(M\) is actually a multi-map. In the nested for loops between line 8 and line 14 we update values of the map \(M\) if a sampled element is a subset of any of the \(s \in U\). Finally, in line 15 we sum the variables corresponding to the elements in \(M\) and output the value multiplied with an appropriate scaling number.

**Theorem 4.1.** Let \(X = \{x_1, \ldots, x_n\}\) be the set of \(n\) variables and let \(U\) 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 \(\text{ApproxCov}\) is an \((1 \pm \epsilon)\)-multiplicative approximation of the \(|\text{Cov}(U)|\). That is, with probability at least \((1 - \delta)\),

\[
(1 - \epsilon)|\text{Cov}(U)| \leq \frac{\text{Out}_1}{\text{Out}_2} \leq (1 + \epsilon)|\text{Cov}(U)|.
\]

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

4.2 Counting the Coverage with Constraints

In this section we present the algorithm \(\text{ApproxCovMax}\) for estimating \(|\text{Cov}(\text{Sol}(F))|\). Notice that it is not straightforward to use \(\text{ApproxCov}\) to estimate this quantity as \(\text{Sol}(F)\) is not given explicitly. Hence we design a new algorithm \(\text{ApproxCovMax}\) 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 \(G^F\) on variable set \(X \cup S\), where \(S\) is an additional set of variables, such that

\[
|\text{Cov}(\text{Sol}(F))| = |\text{Sol}(G^F[S])|
\]

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

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

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

of the \(|\text{Cov}(U)|\). Hence \(|\text{Cov}(U)|\) is at least \((1 - \delta)\),

\[
(1 - \epsilon)|\text{Cov}(U)| \leq \frac{\binom{n^2}{t}}{t} \sum_{k=1}^{t} M[w_k] \leq (1 + \epsilon)|\text{Cov}(U)|.
\]

Thus the correctness of the algorithm \(\text{ApproxCovMax}\) follows once we show that the formula \(G^F(X, S)\) satisfies the condition in Equation 4.
Lemma 4.2.
\[ |\text{Cov}(\text{Sol}(F))| = |\text{Sol}(G^F_{\{S\}})| \]

We now present the main idea of the construction of $G^F_t$.

**Construction of $G^F_t$:** Intuitively, $G^F_t$ encodes $F$ and all the $t$-wise coverage tuples contained in $\text{Sol}(F)$. Given natural numbers $n$ and $t$, and a Boolean formula $F$ on $n$ variables $X = \{x_0, \ldots, x_{n-1}\}$, we will define a new Boolean formula $G^F_t$ on $n + t |\log_2 n|$ + $t$ 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 $Z = \{z_1, \ldots, z_t\}$. Intuitively, $S$ encodes a subset of indices $\{1, \ldots, n\}$ of size $t$: $Y_i$ is a bit-vector that encodes the $i^{th}$ integer in the subset and $z_i$ is its value in $X$. To ensure that only elements of $\text{Cov}(\text{Sol}(F))$ can be assigned to the set $S$, we extend the formula $F$ with additional constraints that is encoded in a formula $F'$ and the formula $G^F_t$ is constructed as a conjunction of $F$ and $F'$. For a binary string $y_t$ of length $|\log_2 n|$, let $\text{val}(y_t)$ denote the number encoded by a binary assignment to the variables in $Y$. $F'$ is a formula over variable $Y \cup Z$ that encodes the following requirements.

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

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

5 EXPERIMENTS 

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

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

As discussed in Section 4, an estimate of $\text{FracCov}(U, F)$ for a given sample set $U$ 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 $\text{FracCov}(U, F)$ is covered by the following research question:

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

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

5.1 Benchmarks & Experimental Setup 

For the experiments we selected a large number of publicly available feature models from real-world configurable systems that were used in the literature for the evaluation of sampling tools. In particular, we took 123 benchmarks appeared in [3, 32, 38, 52, 53]. The benchmarks have between 565 and 11254 variables, between 1146 and 62183 clauses, and between $9.7 \times 10^{13}$ and $7.7 \times 10^{17}$ solutions. Unfortunately, existing approaches are not capable to compute $|\text{Cov}(U)|$ and $|\text{Cov}(\text{Sol}(F))|$ for $t \geq 3$ on large benchmarks. Therefore, in order to check approximations for $t \geq 3$ we have selected smaller benchmarks from [44]. In particular we have sorted the benchmarks by the number of variables and randomly selected 1 for each value between 10 and 500 providing 111 extra benchmarks. In the remainder of the section we would reference the first 123 benchmarks as ‘large’ and the last 111 as ‘small’.

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

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

5.2 Methodology 

The evaluation of ApproxCov is focused on RQ1 and RQ3. In our experiment we used ApproxCov to approximate $|\text{Cov}(U)|$ for $t \in [2, 6]$ on 674 sample sets described above. We used $\epsilon$ and $\delta$ equal to 0.05. ApproxCov computations have been performed 10 times without fixed random seed and in the results we report the mean running time and the worst-case output; i.e. the output farthest from the $|\text{Cov}(U)|$. For comparison, we have performed the same computations with BLMCov [3]. The timeout was set to 36000 seconds and the memory limit was set to 4GB for both the tools.

The evaluation of ApproxMaxCov is focused on RQ2 and RQ4. In our experiments we approximated $|\text{Cov}(\text{Sol}(F))|$ on 234 benchmarks for $t \in [2, 6]$. We used $\delta = \epsilon = 0.05$. Similarly to the previous experiment, all computations with ApproxMaxCov have been performed 10 times without fixed random seed. For comparison, we have performed the same computations with BLMMAXCov [3].
Within this timeout for \( t \) seconds for \( \mathbf{BLCM} \) and \( \mathbf{BLCMax} \). Slower performance on smaller benchmarks was from 2 to 136 times faster. Time comparison is shown in Figure 1; among 3 points on the right border, 2 correspond to timeout of \( \mathbf{BLCM} \) and 1 corresponds to 3295 seconds for \( \mathbf{BLCMax} \).

Originally, the timeout was set to 3600 seconds for both tools. Unfortunately, \( \mathbf{BLCMax} \) is able to compute only 2 'large' benchmarks within this timeout for \( t = 2 \). Therefore, we raised the timeout for \( \mathbf{BLCMax} \) to 28800 seconds. The memory limit was set to 4Gb.

For the evaluation of \( \textbf{RQ5} \) we used the results from the two previous experiments: the estimation of \( \text{FracCov}_i(\mathbf{U}, \mathbf{F}) \) is obtained by dividing the \( i \text{th} \) result of \( \mathbf{Approx} \) by the \( i \text{th} \) result of \( \mathbf{ApproxMax} \) on the corresponding feature model. The 10 generated approximations have been compared with the correct values computed with \( \mathbf{BLCMax} \) and \( \mathbf{BLCMax} \). Confidence parameters of \( \text{FracCov}_i(\mathbf{U}, \mathbf{F}) \) are derived from the selection of \( \varepsilon \) and \( \delta \) in the computation of \( \mathbf{Approx} \) and \( \mathbf{ApproxMax} \): \( \delta = 0.0975, \varepsilon = 0.105 \).

5.3 Results for \( \mathbf{Approx} \)

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

The accuracy of \( \mathbf{Approx} \) approximation, we compared the approximation results with the correct value of \( |\text{Cov}_2(\mathbf{U})| \) computed by \( \mathbf{BLCMax} \). For each sample set we took the maximal value of \( \text{abs}(\text{result}_i - |\text{Cov}_2(\mathbf{U})|)/|\text{Cov}_2(\mathbf{U})| \) among 10 results for each sample set. Figure 2 shows the histogram of errors, the largest error was 0.0315 which is smaller than selected \( \varepsilon \).

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

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

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

To check the accuracy of ApproxMaxCov approximation, we compared it with \( \text{Cov}_2(\text{Sol}(F)) \). For each benchmark, we took the maximal value of \( \frac{\text{abs}(\text{result} - \text{Cov}_2(\text{Sol}(F)))}{\text{Cov}_2(\text{Sol}(F))} \) among 10 results of ApproxMaxCov. Results for 8 benchmarks are shown in Table 2. Figure 5 shows the histogram of errors, the largest error was 0.0058 which is smaller than selected \( \epsilon \).

For larger values of \( t \), ApproxMaxCov timed out of 3600 seconds on the largest benchmark for \( t \geq 4 \) and on one more benchmark for \( t = 6 \). Further exploration of these 2 benchmarks showed that ApproxMaxCov can generate all results within 8500 seconds. The execution time for various values of \( t \) is shown in Figure 6.
BLMMaxCov was able to compute 81 benchmarks for $t = 3$, 19 for $t = 4$, 9 for $t = 5$, and only 5 for $t = 6$. Comparison of approximation with $|\text{Cov}_t(\text{Sol}(F))|$ showed the largest error of 0.0086 which is within PAC guarantees. The error histogram is shown in Figure 7.

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

5.5 Approximation of FracCov$_t(\mathcal{U}, F)$

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

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

5.6 Extension to the General Case

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

5.6.1 Generalization of Algorithms

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

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

5.6.2 Empirical Results

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

The timeouts were 3600 seconds for BLMCov, ApproxCov, and ApproxCov, and 14400 seconds for BLMMaxCov. Memory limit was 4Gb. Both ApproxCov and ApproxCov have been run 10 times, we report mean time among 10 runs and the furthest result from the exact value computed with BLMCov and BLMMaxCov.
Figure 9: Comparison of execution time of ApproxCov and BLMCov for \( t \in [2, 6] \) on sample sets on which BLMCov have successfully terminated - 99 points in total. X axis shows BLMCov time in seconds, Y axis shows ApproxCovGeneral time in seconds. Red line indicates equal time.

Figure 10: Approximation error of ApproxCov for \( t \in [2, 6] \) on 99 sample sets on which BLMCov have successfully terminated. The error is computed as \( \max(\text{abs}(\text{result}_t - |\text{cov}(U)|)/|\text{cov}(U)|) \), where \( \text{result}_t \) is approximation returned by ApproxCov on the \( t \)th run.

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

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

5.7 Threats to Validity

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

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

6 CONCLUSION

Scalable and efficient computation of \( t \)-wise coverage is of pivotal importance for Combinatorial testing. In this work, we propose algorithms for estimating \( t \)-wise coverage for a given set of tests and also for tests sets with a given set of constraints. In particular, we present (1) a scalable Monte-Carlo based framework ApproxCov that is guaranteed to estimate the size of the coverage of for a given set of tests within \((1 \pm \varepsilon)\)-factor of the ground truth with probability at least \(1 - \delta\) for given \(\varepsilon\) and \(\delta\); (2) a scalable counting-based framework ApproxCovMaxCov that estimates maximal achievable coverage for a given formula and guarantees it to be within \((1 \pm \varepsilon)\)-factor with probability at least \(1 - \delta\) for given \(\varepsilon\) and \(\delta\). The approach have been evaluated on a large set of benchmarks involving up to 11000 variables and we have shown that both frameworks can provide highly accurate results even for estimation of 6-wise coverage of features. We also extended the frameworks to also include non-binary domains. Our work opens the possibility to compare various test set generators with the presented frameworks. An important direction for future work would be to perform an extensive evaluation of the existing test set generators by estimating the maximum achievable \( t \)-wise coverage and exploring the possibility to improve them based on the evaluation results.
ACKNOWLEDGEMENTS
This work was supported in part by EU H2020 project Serums (862678:SERUMS-H2020-SC1-FA-DTS-2018-2020), by National Science Foundation USA (NSF HDR:TRIPODS-1934884), and by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFA1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005]. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES


