# Enumerator: An Efficient Approach for Enumerating all Valid $t$-tuples

Hanefi Mercan, Kamer Kaya, and Cemal Yilmaz
*Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul*
$\{$*hanefimercan, kaya, cyilmaz*$\}$*@sabanciuniv.edu*

***Abstract*—In this paper, we present an efficient approach for enumerating all valid $t$-tuples for a given configuration space model, which is an important task in computing covering arrays. The results of our experiments suggest that the proposed approach scales better than existing approaches.**

***Keywords*-Combinatorial interaction testing; covering arrays; $t$-tuples; sketches**

## I. Introduction

Covering arrays (CAs) have been extensively used for testing [1]. Not taking constraints into account when computing CAs often results in a waste of testing resources [2]. A challenging task in the presence of constraints, which can significantly affect the performance of covering array constructors, is to enumerate/count all valid $t$-tuples, where a $t$-tuple is a combination of settings for $t$ unique configuration options. An efficient and scalable approach for enumerating/counting all valid $t$-tuples is of practical importance; to verify a configuration set being a $t$-way covering array efficiently, all valid $t$-tuples (or at least their number) need to be known. Certain covering array constructors depend on this information to operate.

In combinatorial testing, constraints are typically expressed in the form of either logical expressions or forbidden tuples – combinations of option settings that should not appear in any configuration. For the latter case, forbidden tuples can directly be used to determine the validity of a complete configuration, which is a configuration where each option assumes a value. That is, given a configuration, if any of the forbidden tuples appears in the configuration, then the configuration is invalid. However, the same approach may not be used to determine the validity of a $t$-tuple, which can indeed be considered as a partial configuration, due to the possible presence of implicit forbidden tuples. An implicit forbidden tuple is a forbidden tuple which is inferred from a given set of forbidden tuples. Consequently, *minimum forbidden tuples* [3] may need to be computed. In a nutshell, a minimum forbidden tuple is a forbidden tuple of minimum size, which can be used to determine the validity of not only complete configurations, but also partial configurations, such as $t$-tuples. However, computing minimum forbidden tuples, especially in the presence of tangled constraints can be quite costly (Section III).

When logical expressions are used as constraints, on the other hand, valid $t$-tuples can be determined by trivially expressing each possible $t$-tuple as a logical expression and solving it together with all the other inter-option constraints using an appropriate solver [4], such as a SAT or a CSP solver. However, this process needs to be repeated for every possible $t$-tuple. Since the number of $t$-tuples grows exponentially with $t$ and constraint solving is generally a costly operation, this approach can also quickly become a bottleneck (Section III).

In this paper, we present an approach for efficiently enumerating/counting all valid $t$-tuples. The approach is based on a simple observation: A number of valid configurations can often be generated randomly at low costs. When this observation is coupled with the results of many empirical studies, which strongly suggest that an “enough” number of randomly generated valid configurations typically cover a large portion of all valid $t$-tuples, we propose the following approach: 1) randomly generate a number of valid configurations; 2) since all the $t$-

**Algorithm 1** ENUMERATEVALIDTUPLES

```
Input: Configuration space model $M = (O, V, Q)$
       Coverage strength $t$
       Sketch size $s$
Output: Valid $t$-tuples $\mathcal{R}$
 1: $\mathcal{S} \leftarrow$ GENERATESKETCH($M$,$s$)
 2: $\mathcal{R} \leftarrow$ DETERMINEVALIDTUPLES($M$,$t$,$\mathcal{S}$)
```

tuples appearing in a valid configuration are valid, mark all the $t$-tuples appearing in these configurations as valid; and 3) use a constraint solver to determine the validity of the remaining $t$-tuples.

To generate a valid configuration in the first step, either all the constraints can be expressed as logical expressions (if they are not already) and solved together, such that a solution corresponds to a valid configuration; or a complete configuration can first be generated randomly and then checked to see whether it is valid. Without losing the generality of the proposed approach, we, in this work, use the latter. In either case, when forbidden tuples are used as constraints, valid configurations can be generated without requiring to compute minimum forbidden tuples. And when logical expressions are used as constraints, since the constraint solver needs to be invoked on a per configuration basis, not on a per $t$-tuple basis, the number of calls to the constraint solver is significantly reduced. On the other hand, we still invoke the constraint solver on a per $t$-tuple basis in the third step. However, if a large portion of the valid $t$-tuples are covered by the randomly generated configurations in the second step, this operation will need to be carried out only for a small fraction of all $t$-tuples.

## II. APPROACH

Given a configuration space model $M = (O, V, Q)$, where $O = \{o_1, \cdots, o_k\}$ is a set of configuration options, $V = \{V_1, \cdots, V_k\}$ is their domains, such that each option $o_i \in O$ assumes a value from the corresponding finite domain $V_i \in V$, and $Q = \{q_1, \cdots, q_m\}$ is a set of constraints specifying the valid configuration space, we refer to a set $\mathcal{S}$ of valid configurations as a *sketch*. In general, sketches store a summary of data in situations where the whole data would be prohibitively costly to store, process, unknown, etc.

**Algorithm 2** GENERATESKETCH

```
Input: Configuration space model $M = (O, V, Q)$
       Sketch size $s$
Output: A sketch $\mathcal{S}$, where $|S| = s$
 1: $S \leftarrow \emptyset$
 2: while $|\mathcal{S}| \leq s$ do
 3:     $C \leftarrow \emptyset$                         ▷ an empty configuration
 4:     for $1 \leq i \leq k$ do
 5:         $v \in_R V_i$                  ▷ randomly pick a setting for $o_i$
 6:         $C \leftarrow C \cup \{< o_i, v >\}$
 7:     if $C \notin \mathcal{S}$ and $isValid(C, M)$ then
 8:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
```

For this study, we use sketch $\mathcal{S}$ as a filter to reduce the number of times a constraint solver, such as a CSP solver, is invoked. More specifically, since all the $t$-tuples covered by $\mathcal{S}$ are valid, the constraint solver does not need to be invoked for them. In a sense, all the answers obtained from the sketch are true-positives. However, the sketch is allowed to report false-negatives, i.e., valid $t$-tuples that are missing from $\mathcal{S}$. Each false-negative is penalized by an invocation of the solver to determine the validity of the respective $t$-tuple. Consequently, the effectiveness of a sketch can be measured by the percentage of valid $t$-tuples it covers; the more $t$-tuples a sketch covers, the more effective it is.

Algorithm 1 presents the proposed approach. First, a sketch $\mathcal{S}$ of size $s$ is computed (line 1) and then the sketch is used to determine the valid $t$-tuples $\mathcal{R}$ (line 2). Although the current description stores all valid $t$-tuples in collection $\mathcal{R}$, this does not need to be the case in practice. That is, for example, instead of maintaining all the valid $t$-tuples in memory, they can be persisted as they are discovered, or only the invalid $t$-tuples can be maintained, or only the counts can be computed, which can further improve the scalability of the proposed approach.

One way to randomly generate a valid configuration is to express all the constraints as logical expressions (if they are not already), such that a solution represents a configuration. For this work, without losing the generality of the proposed approach, we experiment with an alternative approach as described in Algorithm 2. We first generate a complete configuration by randomly choosing a setting $v \in V_i$ from a uniform distribution for each

**Algorithm 3** DETERMINEVALIDTUPLES

```
Input: Configuration space model $M = (O, V, Q)$
       Coverage strength $t$
       Sketch $\mathcal{S}$
Output: Valid $t$-tuples $\mathcal{R}$
1:  $\mathcal{R} \leftarrow \emptyset$
2:  for each option comb. $Z = \{o_{i_1}, \cdots, o_{i_t}\}$ do
3:      $covered \leftarrow \emptyset$
4:      for each $C \in \mathcal{S}$ do
5:          $R \leftarrow t$-tuple consisted of options $Z$ in $C$
6:          $covered \leftarrow covered \cup \{R\}$
7:      for each possible $t$-tuple $R$ consisted of options $Z$ do
8:          if $R \in covered$ then
9:              $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
10:         else
11:             if $satisfiable(M, R)$ then
12:                 $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
```

configuration option $o_i \in O$ (lines 4-6). We then check the validity of the generated configuration (lines 7-8). This process is repeated until $s$ valid configurations have been generated (lines 2-8).

If forbidden tuples are used as constraints in the model $M$, $isValid(C, M)$ checks to see whether any of the forbidden tuples appear in the configuration $C$. Since $C$ is a complete configuration, this can be done without computing the implicit forbidden tuples and/or minimum forbidden tuples. If logical expressions are used as constraints, the configuration $C$ is first expressed as a logical expression and then checked to see whether the expression is satisfiable together with all the constraints in $M$. Since this is done on a per selected configuration basis, rather than on a per $t$-tuple basis, the number of calls to the solver is significantly reduced.

Once a sketch is created, Algorithm 3 uses it to determine all valid $t$-tuples $\mathcal{R}$. To this end, we enumerate all possible combinations of $t$ options (line 2). For each combination $Z = \{o_{i_1}, \cdots, o_{i_t}\}$, we first mark all the $t$-tuples consisted of options $Z$ appearing in the sketch $S$ as valid (lines 4-6). For each of the remaining $t$-tuples, we then check to see if the $t$-tuple is satisfiable together with all the constraints in the model $M$ by using a solver (line 11). If so, the $t$-tuple is valid (line 12); otherwise, it is invalid.

To further improve the performance of the proposed approach, we have also parallelized it in a coarse-grain manner via OpenMP. More specifically, Algorithm 2 is parallelized by having each thread execute a different iteration of the while loop in line 2, with basic synchronization mechanism for line 8 to populate the current sketch with a newly computed configuration. Furthermore, Algorithm 3 is parallelized by having each thread execute a different iteration of the for loop in line 2, which in turn requires a synchronization mechanism for lines 9 and 12 to update the set of valid $t$-tuples with a newly discovered $t$-tuple.

## III. EXPERIMENTAL RESULTS

In our experiments, we selected a number of configuration space models from two widely-used benchmarks obtained from real-world software systems, called *real-1* and *real-2* [2], [5]. The models are selected based on the complexity of the configuration spaces they specify (Table I). In all of them, the constraints were expressed as forbidden tuples. The sketch size was set to $25000$ and we experimented with three coverage strengths $t = \{2, 3, 4\}$.

We compared the performance of the proposed approach to those of two existing ones, which we respectively refer to as ALGO1 and ALGO2:

- ALGO1 first computes the minimum forbidden tuples with the algorithm given in [3]. It then determines the valid $t$-tuples by checking all possible $t$-tuples against the minimum forbidden ones. A $t$-tuple is valid, if and only if it does not contain any of the minimum forbidden tuples.
- ALGO2 enumerates all possible tuples and then checks them one-by-one by using a constraint solver. Thus, it can be considered as the proposed approach with an empty sketch.

All the experiments were carried out on an Intel Xeon E5-2680 v2 2.80 GHz machine with 256 GB of RAM, running 64-bit CentOS 6.5 as the operating system. For ALGO2 and the proposed approach, we used Gecode [6] as the constraint solver. For each experiment, a 2-hour time limit is used, i.e., we killed the processes after 2 hours.

Table I presents the results in seconds. The cases where the enumeration of valid $t$-tuples could not be completed within the time limit are marked by $\infty$. In the table, ENUM represents the proposed approach with a single thread and ENUM$^*$ does the same when 8 threads are used.

TABLE I
THE PROPERTIES OF THE BENCHMARK MODELS USED IN THE EXPERIMENTS (FIRST THREE COLUMNS) AND THE EXECUTION TIMES (SEC.) OF ALGO1, ALGO2, ENUM, AND ENUM* ON THEM FOR $t \in \{2, 3, 4\}$.

| | | #Forbidden | $t$=2 | | | | $t$=3 | | | | $t$=4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Model | Size of the space | tuples | ALGO1 | ALGO2 | ENUM | ENUM* | ALGO1 | ALGO2 | ENUM | ENUM* | ALGO1 | ALGO2 | ENUM | ENUM* |
| `Apache` | $2^{158}\ 3^{8}\ 4^{4}\ 5^{1}\ 6^{1}$ | 7 | 0.014 | 16.72 | 0.005 | 0.002 | 1.288 | 2037 | 0.991 | 0.154 | 86.77 | $\infty$ | 96.52 | 14.21 |
| `Banking2` | $2^{14}\ 4^{1}$ | 3 | 0.000 | 0.021 | 0.000 | 0.000 | 0.001 | 0.193 | 0.003 | 0.002 | 0.005 | 1.306 | 0.023 | 0.007 |
| `CommProtocol` | $2^{10}\ 7^{1}$ | 128 | 6804 | 0.452 | 0.017 | 0.007 | 6774 | 2.729 | 0.259 | 0.066 | 6768 | 12.30 | 1.775 | 0.403 |
| `Gcc` | $2^{189}\ 3^{10}$ | 40 | 0.040 | 69.46 | 0.015 | 0.004 | 3.660 | $\infty$ | 3.878 | 0.589 | 246 | $\infty$ | 561 | 88.52 |
| `Healthcare4` | $2^{13}\ 3^{12}\ 4^{6}\ 5^{2}\ 6^{1}\ 7^{1}$ | 22 | 0.009 | 1.232 | 0.004 | 0.002 | 0.050 | 43.38 | 0.291 | 0.070 | 0.864 | 1015 | 12.28 | 2.399 |
| `Insurance` | $2^{6}\ 3^{1}\ 5^{1}\ 6^{2}\ 11^{1}\ 13^{1}\ 17^{1}\ 31^{1}$ | 0 | 0.001 | 0.134 | 0.000 | 0.000 | 0.009 | 3.608 | 0.008 | 0.003 | 0.179 | 56.86 | 1.864 | 1.591 |
| `Storage5` | $2^{5}\ 3^{8}\ 5^{3}\ 6^{2}\ 8^{1}\ 9^{1}\ 10^{2}\ 11^{1}$ | 151 | $\infty$ | 6.466 | 0.085 | 0.081 | $\infty$ | 192 | 7.171 | 2.096 | $\infty$ | 4274 | 322 | 62.12 |
| `Telecom` | $2^{5}\ 3^{1}\ 4^{2}\ 5^{1}\ 6^{1}$ | 21 | 0.002 | 0.068 | 0.001 | 0.001 | 0.002 | 0.545 | 0.013 | 0.016 | 0.007 | 2.821 | 0.120 | 0.085 |

We observed that the sketches we generated cover a large portion of the valid $t$-tuples (on average 99%). Consequently, ENUM significantly performed better than ALGO2 with average running times of $16.6$ vs. $368$ seconds, respectively. In three of the models for `Apache` and `gcc`, while ALGO2 timed out, ENUM determined all valid $t$-tuples under $561$ seconds. Furthermore, parallelizing the proposed approach greatly increased the performance; ENUM's average runtime was $42.04$ seconds whereas it was $7.18$ seconds for ENUM*.

Last but not least, for smaller and less complex configuration spaces, ALGO1 performed only slightly better than ENUM. For example, when $t \in \{2, 3\}$, ALGO1 was faster by at most $0.241$ seconds, hardly having any practical significance. However, for larger and more complex configuration spaces, the proposed approach was significantly better. For example, for `Storage5`, although ALGO1 timed out even for $t = 2$, ENUM* took $0.085$, $7.171$, and $322$ seconds to enumerate all valid $t$-tuples for $t = 2$, $3$, and $4$, respectively. All these results strongly suggest that the proposed approach greatly improved the scalability of the existing approaches.

## IV. RELATED WORK

One of the closest work to ours is [4]; however, the aforementioned approach 1) operates on partial configurations, which necessitates many more invocations of a solver; 2) depends on a particular covering array constructor, thus requires an adaptation for different constructors; and 3) determines all the valid $t$-tuples only after a valid covering array is computed. In another work [7], the authors check the list of $t$-tuples one by one. While doing that, if a new forbidden tuple is discovered, the $t$-tuples containing this newly found forbidden tuple are removed from the list. Our approach, on the other hand, 1) operates on complete configurations; 2) is agnostic to constructors as all the configurations in the sketch are randomly selected; and 3) does not require to compute a CA before the valid $t$-tuples can be determined as the $t$-tuples that are not covered by the sketch are checked via a solver.

## V. CONCLUDING REMARKS

We believe that the results of our experiments are of great practical importance. Therefore, we will continue to work in this line of research. One possible avenue for future work is to extend the proposed approach to infer minimum forbidden tuples regardless of whether the constraints are given as forbidden tuples or logical expressions.

## REFERENCES

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[2] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. on Soft. Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[3] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Constraint handling in combinatorial test generation using forbidden tuples," in *8th IEEE International Conference on Software Testing Verification and Validation Workshop(ICSTW)*. IEEE, 2015, pp. 1–9.

[4] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *6th IEEE Conf. on Software Testing, Validation and Verification*. IEEE, 2013, pp. 242–251.

[5] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 254–264.

[6] Gecode Team, "Gecode: Generic constraint development environment," 2006, available from `http://www.gecode.org`.

[7] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM Int. Conf. on*. IEEE, 2016, pp. 614–624.