A New Algorithm for Post-Processing Covering Arrays

Carlos Lara-Alvarez
CONACYT Research Fellow.
HCI-Lab, Center for Research in Mathematics (CIMAT).
Av. Universidad 222, 98068 Zacatecas, Mexico

Himer Avila-George
CONACYT Research Fellow - HARAMARA TIC-LAB,
CICESE Unidad Tepic (CICESE-UT^3),
Andador 10 #109, 63173 Tepic, Nayarit, Mexico

Abstract—**Software testing is a critical component of modern software development. For this reason, it has been one of the most active research topics for several years, resulting in many different algorithms, methodologies and tools. Combinatorial testing is one of the most important testing strategies. The test generation problem for combinatorial testing can be modeled as constructing a matrix which has certain properties, typically this matrix is a covering array. The construction of covering arrays with the fewest rows remains a challenging problem. This paper proposes a post-processing technique that repeatedly adjusts the covering array in an attempt to reduce its number of rows. In the experiment, 85 covering arrays, created by a state-of-the-art algorithm, were subject to the reduction process. The results report a reduction in the size of 28 covering arrays (∼33%).**

Keywords—Software testing; Combinatorial testing; Covering arrays; Post-Processing

I. Introduction

The ever increasing complexity, ubiquity, and dynamism of modern software systems demands new approaches to quality assurance. Extensive testing is required to assure that software works correctly, however, in many practical applications the number of configurable parameters may be large, and testing all possible configurations is not possible due to limited testing resources. Combinatorial testing enables the tester to execute a small set of test cases on the system, while achieving very high fault coverage. The pairwise test is one of main approaches in black-box testing. Several studies have demonstrated the effectiveness of *pairwise* testing [1], [2]. By examining fault reports for several systems [3] shown that ∼100% of faults can be discovered with *4-wise* to *6-wise* interactions.

The first step to apply combinatorial testing is to construct a parametrized model of the *System Under Test* (SUT). The tester should first identify the input parameters related to the test goal, i.e. parameters affecting the system behavior; they may include but not limited to the following: (a) parameters of method calls; (b) parameters in system settings; and (c) a selection of replaceable system components installed in a test environment, such as hardware devices, system libraries and applications [4].

The key idea of combinatorial testing is that most of the SUT faults can be detected by combinations of a small number of factors. In combinatorial testing, a covering array (CA) is usually used as test suite, which covers parameter combinations involving t factors.

Covering Arrays (CA) are one of the most popular methods for representing pseudo-exhaustive test suites, they are small in comparison with an exhaustive approach but guarantee a level of interaction coverage among the parameters involved. They focus on having minimum cardinality (i.e. minimize the number of test cases), and maximum coverage (i.e. they guarantee to cover all combinations of certain size between the input parameters). To address this problem it has been proposed several methods (e.g., algebraic, exact, greedy and metaheuristic); however, usually they produce quasi-optimal covering arrays that contain combinations of symbols which are covered more than once (redundant). Redundancy opens the possibility for designing post-processing algorithms that eliminate the redundant information in the existing covering arrays with the aim of improve them.

This paper presents a new algorithm called *Post-Procesing Covering Arrays* (PPCA) for eliminating redundant tests; it receives a covering array as input, then it tries to reduce the number of tests (rows).

The remainder of this paper is organized in four more sections. Section II, presents a brief overview of the principal techniques and tools for constructing covering arrays; Section III presents the new algorithm for post-processing covering arrays by deleting unnecessary tests. Section IV, shows the complete results for post-processing a benchmark composed by 85 covering arrays. Final remarks are presented in the section V.

II. Related work

There are several methods for constructing covering arrays; according to the strategy for generating covering arrays, they can be classified into algebraic, exact, greedy and metaheuristic approaches. Additionally, there are some useful operations that can be applied to a covering array previously constructed.

Algebraic approaches use formulas or operations with mathematical objects such as cyclic vectors [5], permutation vectors (Zero-sum method [6]), groups [7], cover starter [8] or covering arrays with small values of t, k, v (doubling [9] and v-plication [10] Algebraic constructions often provide a better bound in less computational time, but impose serious restrictions on the system configurations to which they can be applied. For example, many approaches for constructing covering arrays require that the domain size be a prime number or a power of a prime number; this significantly limits the applicability of algebraic approaches for testing.

Greedy approaches are more flexible than algebraic constructions. These methods can generate any covering array using as input t, k, and v. The majority of commercial and open source test data generating tools use greedy approaches for covering arrays construction (TVG [11], ACTS [12], Jenny [13] and $T\ tuples$ tool [14]). The problem with these approaches are the quality of results –greedy methods rarely obtain optimal covering arrays–.

The *exact approaches* are exhaustive methods for the construction of optimal covering arrays. Despite of the fact that some approaches have techniques for accelerating the search process, in general they require an exponential time for completing the task, making them only practical for constructing small covering arrays. Some examples of this type of construction were reported in [15], [16], [17].

Metaheuristic approaches do not guarantee the construction of the optimal covering array but in practice they give good results in a reasonable amount of time. Among the most used metaheuristics are simulated annealing [18], tabu search [19], [20] and genetic algorithms [21].

III. Methodology

This section presents an algorithm for post-processing covering arrays; it starts with some basic definitions that introduce the problem, and then the proposed algorithm is described.

A. Definitions and Preliminaries

Definition 1: Let N, t, k, and v be positive integers where $t \leq k$. A covering array $\mathrm{CA}(N; t, k, v)$ is a matrix of size $N \times k$ and strength t where each column has entries from alphabet Σ of size v. In every $N \times t$ subarray, all possible v^t t-tuples of symbols occurs at least once. Then N is the number of rows, t is the strength of the coverage of interactions, k is the number of factors (also called the degree), and v is the number of symbols for each factor (also called the order).

Definition 2: A t-way interaction is the assignment of specific values to each factor from set of t factors. The array is 'covering' in the sense that every t-way interaction is represented by at least one experimental run. In any covering array, the number of $N \times t$ subarrays is $M = \binom{k}{t}$, and the number of t-way interactions to be covered is $\binom{k}{t}\ v^t$.

Definition 3: The covering array number $\mathrm{CAN}(t, k, v)$ is the smallest N for which a $\mathrm{CA}(N; t, k, v)$ exists. The $\mathrm{CA}N$ is defined according to

$$\mathrm{CAN}(t, k, v) = \min\{N : \exists\ \mathrm{CA}(N; t, k, v)\};$$

evidently $\mathrm{CA}N(t, k, v) \geq v^t$.

When a covering array is used as test suite:

- Each column represents a parameter of the software under testing (SUT).
- The symbols in the column specify the values for such parameter.
- Each row represents a test case to be performed.

$$\text{(a)}\quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{pmatrix} \qquad \text{(b)}\quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 0 \\ * & * & * & * \end{pmatrix}$$

Fig. 1: Detection of a redundant row (a) a covering array $\mathrm{CA}(10; 2, 4, 3)$; (b) the last row (maked by asterisks) is not required in $\mathrm{CA}(9; 2, 4, 3)$.

- The fundamental problem is to determine $\mathrm{CAN}(t, k, v)$.

When a covering array is constructed (see section II), it can contain t-way interactions which are covered more than once (in the definition of a covering array, the indication at *least once* means that a combination of symbols can be covered more than once). This fact opens the possibility that some symbols in certain positions are redundant and can be changed for any value without affecting the coverage of a CA, these symbols are referred to as *redundant*. To illustrate the existence of redundant rows, consider the example provided in Fig. 1. If the last row is deleted from the $\mathrm{CA}(10; 2, 4, 3)$ shown in Fig. 1(a) then the matrix shown in Fig. 1(b) is obtained which is still a covering array because all 2–combinations of symbols are present. Hence, the last row is *redundant* and can be deleted from the original matrix; then $\mathrm{CA}(9; 2, 4, 3)$ is better than the original one.

B. Proposed approach

Let $\mathcal{R}$ be the set of possible realizations (t–tuples of Σ), and $\mathcal{I} = (\mathcal{I}_j)_{j=1}^{M}$ be the vector of interactions (t–tuples of columns). The i–th row test r_i can be represented by a vector S_i of the form

$$S_i = (s_{ij})_{j=1}^{M}, \tag{1}$$

where the t–way interaction $s_{ij} = (\mathcal{I}_j, v_{ij})$ associates the interaction $\mathcal{I}_j$ to its realization $v_{ij} \in \mathcal{R}$ in the i–th test.

In the example shown in Fig. 2, the set of possible realizations are $\mathcal{R} = \{00, 01, 10, 11\}$ and the interactions are $(\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2) = (c_0c_1, c_0c_2, c_1c_2)$ and the row test r_0 is represented by $S_0 = ((c_0c_1, 10), (c_0c_2, 10), (c_1c_2, 00))$, the row test r_1 by $S_1 = ((c_0c_1, 10), (c_0c_2, 11), (c_1c_2, 01))$, and so on.

Elements of S_i can be used for building an index $\mathcal{M}$ that maps t–way interactions to lists of row tests that cover each interaction. That is,

$$\mathcal{M} = ((e_o, \mathcal{L}(e_o))_{o=1}^{N} \tag{2}$$

where $e_o \in \mathcal{I} \times \mathcal{R}$, and $\mathcal{L}(e_o)$ is the list of rows that test e_0.

For obtaining the reduced covering array, the lists of rows in $\mathcal{M}$ are iteratively modified by removing elements. Given a map $\mathcal{M}$, the vector of cardinality $S_i^{\#}$ of row test r_i is defined as

$$S_i^{\#} = (f(s_{ij}))_{j=1}^{M}, \tag{3}$$

where

$$f(s_{ij}) = \begin{cases} \#\mathcal{L}(s_{ij}) & \text{if } s_{ij} \in \text{keys}(\mathcal{M}) \\ N+1 & \text{otherwise,} \end{cases} \tag{4}$$

and $\#\mathcal{L}(s_{ij})$ is the size of the list $\mathcal{L}(s_{ij})$.

For deciding which rows are included in the reduced covering array, the vector $S_i^{\#}$ is sorted in ascending order

$$S_i^s = \text{sort}(S_i^{\#}). \tag{5}$$

This array is an indicator of how a row test is required; when the first element of S_i^s is one it means that the row is strictly required. If all elements are set to $N+1$, then the i–th row test is unnecessary. Hence, the order of elements in S_i^s is important; then, for obtaining a reduced covering array vectors S_i^s of all the rows are compared; the first row in the lexicographic order –i.e., first unequal elements determine the order– is selected as the best row test in each iteration; the selected row test is included in the reduced covering array

C. Algorithm

Procedure PPCA(C) shown in the algorithm 1 illustrates the proposed approach. The input C is a covering array of size $N \times k$ and the algorithm produces a reduced covering array C'. The set L is used to store the row tests of C that are included in C', initially L is set to empty. At line 3, the algorithm creates a map $\mathcal{M}$ by analyzing each row test as stated in (2). After that, the algorithm iterates the following steps while the map $\mathcal{M}$ has entries, i.e. keys$(\mathcal{M}) \neq \emptyset$: (a) Select the index i_m such that $S_{i_m}^{\#}$ is the smaller according to the lexicographic order (step 5), (b) Remove entries of $\mathcal{M}$ that include i_m (step 6), and (c) Add i_m to the set L (step 7). Finally, C' is obtained by selecting rows L from C (step 9). Hence, the number of rows of the resulting covering array, $N' \leq N$, is equal to the size of L.

Algorithm 1 A Post-Processing covering array algorithm (PPCA).

```
Require: A covering array, $C$, of size $N \times k$
Ensure: A reduced covering array, $C'$, of size $N' \times k$ with $N' \leq N$
 1: procedure PPCA($C$)
 2:     $L \leftarrow \emptyset$
 3:     $\mathcal{M} \leftarrow$ CREATEMAP($C$)
 4:     while keys($\mathcal{M}$) $\neq \emptyset$ do
 5:         $i_m \leftarrow \underset{i \in 1 \ldots N}{\text{argmin}}\, S_i^s$
 6:         keys($\mathcal{M}$) $\leftarrow$ keys($\mathcal{M}$) $\setminus S_{i_m}$
 7:         $L \leftarrow L \cup \{i_m\}$
 8:     end while
 9:     $C' \leftarrow$ Select rows $L$ from $C$
10:     return $C'$
11: end procedure
```

D. Example

The toy example shown in Fig. 2 is used for clarifying algorithm 1, the matrix C of size 9×3 illustrates the test cases; but some of them are redundant. For obtaining the reduced covering array C' the PPCA algorithm proceeds as is illustrated in Figure 3 and described in the following:

INITIALIZATION:
After creating the initial map $\mathcal{M}$ from C, and obtaining $S_i^s | i = 0, \ldots, 8$; the row $i = 2$ is selected for the first iteration because it is the smaller according to the lexicographic order; i.e. $S_2^s[1, 2, 2]$ is selected because row 2 is strictly required for covering $(c_0c_2, 01)$.

ITERATION 1:
After inserting row 2 into the reduced covering array, and updating the vectors S_i^s for $i = \{1, 6\}$; the row $i = 4$ is selected for the next iteration.

ITERATIONS 2,3:
Row s 3 and 0 were included in C'. Note that if one of the rows $1, 6$ or 8 were selected in iteration 3 (by a function other than the proposed), the covering array C' must include one or more additional rows for the complete covering. But, by using (5) the optimal solution can be found because row 0 completes the covering array.

ITERATION 4:
The algorithm finishes because keys$(\mathcal{M}) = \emptyset$ and the resulting selection $L = \{2, 4, 3, 0\}$ are the row tests included in the reduced covering array.

It is easy to show that all combinations of $t = 2$ are included in the matrix C' that only includes rows $\{2, 4, 3, 0\}$ of C.

IV. Results and Discussion

This section presents an experimental design and results derived from the methodology described in the previous section. An experiment consisting of 85 covering arrays was designed, each covering array was built using a tool called IPOG (one

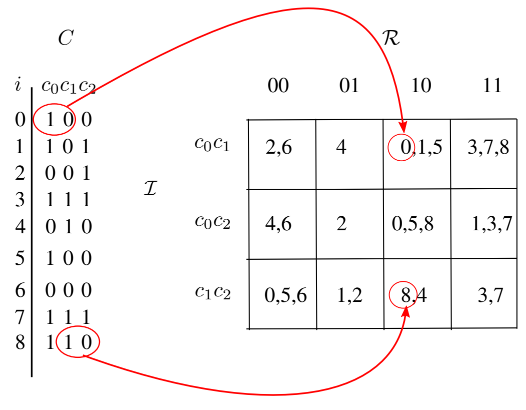

Fig. 2: *Left*: A covering array with $k = 3$, $t = 2$, and $\Sigma = \{0, 1\}$. *Right*: an illustration of the t–way interactions of row tests used for generating the map $\mathcal{M}$ that relates realizations $\mathcal{R}$ to interactions $\mathcal{I}$.

INITIALIZATION

$\mathcal{M}$		
e_O	$\mathcal{L}(e_O)$	$\#\mathcal{L}(e_O)$
$(\mathcal{I}_0, 00)$	[2,6]	2
$(\mathcal{I}_0, 01)$	[4]	1
$(\mathcal{I}_0, 10)$	[0,1,5]	3
$(\mathcal{I}_0, 11)$	[3,7,8]	3
$(\mathcal{I}_1, 00)$	[4,6]	2
$(\mathcal{I}_1, 01)$	[2]	1
$(\mathcal{I}_1, 10)$	[0,5,8]	3
$(\mathcal{I}_1, 11)$	[1,3,7]	3
$(\mathcal{I}_2, 00)$	[0,5,6]	3
$(\mathcal{I}_2, 01)$	[1,2]	2
$(\mathcal{I}_2, 10)$	[8,4]	2
$(\mathcal{I}_2, 11)$	[3,7]	2

i	S_i^s	
0	[3,3,3]	
1	[2,3,3]	
2	[1,2,2]	$\Leftarrow$
3	[2,3,3]	
4	[1,2,2]	
5	[3,3,3]	
6	[2,2,3]	
7	[2,3,3]	
8	[2,3,3]	

$L = \emptyset$

ITERATION 1

$\mathcal{M}$		
e_O	$\mathcal{L}(e_O)$	$\#\mathcal{L}(e_O)$
$(\mathcal{I}_0, 01)$	[4]	1
$(\mathcal{I}_0, 10)$	[0,1,5]	3
$(\mathcal{I}_0, 11)$	[3,7,8]	3
$(\mathcal{I}_1, 00)$	[4,6]	2
$(\mathcal{I}_1, 10)$	[0,5,8]	3
$(\mathcal{I}_1, 11)$	[1,3,7]	3
$(\mathcal{I}_2, 00)$	[0,5,6]	3
$(\mathcal{I}_2, 10)$	[8,4]	2
$(\mathcal{I}_2, 11)$	[3,7]	2

i	S_i^s	
0	[3,3,3]	
1	[3,3,9]	
3	[2,3,3]	
4	[1,2,2]	$\Leftarrow$
5	[3,3,3]	
6	[2,3,9]	
7	[2,3,3]	
8	[2,3,3]	

$L = \{2\}$

ITERATION 2

$\mathcal{M}$		
e_O	$\mathcal{L}(e_O)$	$\#\mathcal{L}(e_O)$
$(\mathcal{I}_0, 10)$	[0,1,5]	3
$(\mathcal{I}_0, 11)$	[3,7,8]	3
$(\mathcal{I}_1, 10)$	[0,5,8]	3
$(\mathcal{I}_1, 11)$	[1,3,7]	3
$(\mathcal{I}_2, 00)$	[0,5,6]	3
$(\mathcal{I}_2, 11)$	[3,7]	2

i	S_i^s	
0	[3,3,3]	
1	[3,3,9]	
3	[2,3,3]	$\Leftarrow$
5	[3,3,3]	
6	[3,9,9]	
7	[2,3,3]	
8	[3,3,9]	

$L = \{2, 4\}$

ITERATION 3

$\mathcal{M}$		
e_O	$\mathcal{L}(e_O)$	$\#\mathcal{L}(e_O)$
$(\mathcal{I}_0, 10)$	[0,1,5]	3
$(\mathcal{I}_1, 10)$	[0,5,8]	3
$(\mathcal{I}_2, 00)$	[0,5,6]	3

i	S_i^s	
0	[3,3,3]	$\Leftarrow$
1	[3,9,9]	
5	[3,3,3]	
6	[3,9,9]	
8	[3,9,9]	

$L = \{2, 4, 3\}$

ITERATION 4

$\mathcal{M}$

$L = \{2, 4, 3, 0\}$

Fig. 3: PPCA for reducing the covering array instance shown in Fig. 2, the reduced covering array C' is obtained by selecting the rows $\{2, 4, 3, 0\}$ from C.

TABLE I: Results of post-processing binary covering arrays, with $2 \leq t \leq 6$ and $k \leq 50$. The number in each entry is the value $N - N'$ for the instance with values k, t.

k	t					
	2	3	4	5	6	
2	0	-	-	-	-	
3	1	0	-	-	-	
4	1	0	0	-	-	
5	1	1	1	0	-	
6	1	1	0	9	0	
7	0	0	3	3	4	
8	0	0	1	3	4	
9	1	0	0	2	3	
10	1	1	0	2	3	
11	0	1	0	0	1	
12	0	0	0	0	0	
13	0	0	0	0	0	
14	1	0	1	0	0	
15	0	0	0	0	0	
16	0	0	0	0	0	
17	1	0	0	0	0	
18	0	0	0	0	1	
19	0	0	0	0	0	
20	1	0	0	0	0	total
# tested instances	19	18	17	16	15	85
# reduced instances	9	4	4	5	6	**28**

of the most popular tools in the state-of-the-art of covering arrays construction).

The results derived from our experiment are shown in table I. In this analysis, binary covering arrays are grouped by the number of their columns and their strength. Every group of t contains the different values of the alphabet for each covering array. Every cell of the this table shows the number of rows reduced in the corresponding binary covering array. As seen in the last row, the results reported a reduction in the size of 28 covering arrays (∼33%).

Section II summarizes the techniques for constructing covering arrays, they can be grouped into: algebraic, greedy, exact and metaheuristics techniques. The best known solutions for CA with $t = 2, 3, \ldots, 6$ are publicly available [8]. By analyzing that results, one can see that metaheuristics techniques produce better bounds but they are computationally expensive. For this reason, these techniques have concentrated on the construction of CA with $k < 100$. Algebraic and greedy techniques are better suited for large covering arrays, i.e. $v > 3$, $k > 100$ and $t > 3$; therefore, PPCA algorithm can be used for post-processing solutions constructed by these heuristics.

V. CONCLUDING REMARKS AND FUTURE WORK

This paper presents a post-processing strategy, called PPCA, for reducing the size of a covering array. The post-processing reduces the number of rows of a covering array through iteratively including the best row in the reduced covering array –the row that is most important for guaranteeing covering–. In some cases, the reduced covering array could be optimized but here we are interested just in reducing the size of a previously constructed CA, not in building a new one.

A dataset of 85 covering arrays constructed by the state-of-the-art algorithm IPOG was used to test the PPCA algorithm. The results show a reduction in ∼33% of the instances.

In conclusion, PPCA has already proved being effective for reducing a wide variety of covering arrays.

We are designing a parallel version of the PPCA algorithm, in order to address problems with high strength, many factors or rows.

ACKNOWLEDGMENT

The following projects have funded the research reported in this paper: 148784 - Fondo Mixto CONACyT y Gobierno del Estado de Nayarit; 2143 - Cátedras CONACyT (H. Avila), 3163 - Cátedras CONACyT (C. Lara).

REFERENCES

[1] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[2] B. Pérez Lamancha, M. Polo, and M. Piattini, "PROW: A pairwise algorithm with constraints, order and weight," *Journal of Systems and Software*, vol. 99, pp. 1 – 19, 2015.

[3] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[4] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *Journal of Systems and Software*, vol. 98, pp. 191 – 207, 2014.

[5] C. J. Colbourn, "Covering arrays from cyclotomy," *Designs, Codes and Cryptography*, vol. 55, no. 2-3, pp. 201–219, 2010.

[6] G. Sherwood, "On the construction of orthogonal arrays and covering arrays using permutation groups," 2015, available online at http://testcover.com/pub/background/cover.htm. Accessed October 20, 2015.

[7] K. Meagher and B. Stevens, "Group construction of covering arrays," *Journal of Combinatorial Designs*, vol. 13, no. 1, pp. 70–77, 2005.

[8] C. J. Colbourn, "Covering array tables," 2015, available online at http://www.public.asu.edu/~ccolbou/src/tabby/3-3-ca.html. Accessed on October 11, 2015.

[9] M. Chateauneuf and D. L. Kreher, "On the state of strength-three covering arrays," *Journal of Combinatorial Designs*, vol. 10, no. 4, pp. 217–238, 2002.

[10] N. J. A. Sloane, "Covering arrays and intersecting codes," *Journal of Combinatorial Designs*, vol. 1, no. 1, pp. 51–63, 1993.

[11] P. J. Schroeder, E. Kim, J. Arshem, and P. Bolaki, "Combining behavior and data modeling in automated test case generation," in *Proceedings of the Third International Conference on Quality Software*, 2003, pp. 247–254.

[12] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, pp. 287–297, 2008.

[13] B. Jenkins, "Jenny: a pairwise testing tool," 2015, available online at http://burtleburtle.net/bob/math/jenny.html. Accessed on October 11, 2015.

[14] A. Calvagna and A. Gargantini, "T-wise combinatorial interaction test suites construction based on coverage inheritance," *Software Testing, Verification and Reliability*, vol. 22, pp. 507–526, 2012.

[15] J. Martinez-Pena and J. Torres-Jimenez, "A branch and bound algorithm for ternary covering arrays construction using trinomial coefficients," *Research in Computing Science*, vol. 49, pp. 61–71, 2010.

[16] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers," in *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, 2010, pp. 112–126.

[17] C. Ansótegui, I. Izquierdo, F. Manya, and J. Torres-Jimenez, "A Max-SAT-Based approach to constructing optimal covering arrays," in *Artificial Intelligence Research and Development*, 2013, pp. 51–59.

[18] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Constructing strength three covering arrays with augmented annealing," *Discrete Mathematics*, vol. 308, no. 13, pp. 2709 – 2722, 2008.

[19] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, pp. 143–152, 2004.

[20] R. A. Walker II and C. J. Colbourn, "Tabu search for covering arrays using permutation vectors," *Journal of Statistical Planning and Inference*, vol. 139, no. 1, pp. 69–80, 2009.

[21] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proc. of the 28th Annual Intl. Computer Software and Applications Conf.* IEEE Computer Society, 2004, pp. 72–77.