Generating Covering Arrays with Pseudo-Boolean Constraint Solving and Balancing Heuristic

Hai Liu[2], Feifei Ma[1(✉)], and Jian Zhang[1]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{maff,zj}@ios.ac.cn
[2] Beijing Information Science and Technology University, Beijing, China
stuliuhai@gmail.com

Abstract. Covering arrays (CAs) are interesting objects in combinatorics and they also play an important role in software testing. It is a challenging task to generate small CAs automatically and efficiently. In this paper, we propose a new approach which generates a CA column by column. A kind of balancing heuristic is adopted to guide the searching procedure. At each step (column extension), some pseudo Boolean constraints are generated and solved by a PBO solver. A prototype tool is implemented, which turns out to be able to find smaller CAs than other tools, for some cases.

1 Introduction

In complex software systems, faults usually arise from the interaction of a few components/factors. It is reported that up to 90 % of the faults are caused by interactions of at most 3 factors, among which 70 % are caused by pairwise interactions [6]. Combinatorial testing is a useful black-box testing technique to reveal such faults. It usually uses the concept of Covering Arrays (CAs) [13]. Such arrays are important objects in combinatorics. A covering array of strength t is an array with property that each ordered combination of t values from different columns appears at least once in the rows. Each row corresponds to a test case.

Given parameters and coverage criteria, we would like to obtain CAs with the least number of rows, which correspond to the smallest test suites for the System Under Test (SUT). However, generating such CAs is a challenging task, which has been proved to be NP-complete. Therefore, most works on CA generation are based on heuristic search, so as to obtain a solution in reasonable time. The heuristics adopted in the search algorithms are vital to the optimality of the solutions and the efficiency of the algorithms.

This work has been supported in part by the National 973 Program under grant No. 2014CB340701, and by the Youth Innovation Promotion Association, CAS (grant No. 2016104).

R. Booth and M.-L. Zhang (Eds.): PRICAI 2016, LNAI 9810, pp. 262–270, 2016.
DOI: 10.1007/978-3-319-42911-3_22

In general, search algorithms for CAs using problem-specific heuristics fall into two categories: the one-test-at-a-time strategy and its variants, and the In-Parameter-Order (IPO) family [7,8]. The one-test-at-a-time strategy is widely employed in many CA generation algorithms [1]. The basic idea is to generate test cases one by one in a greedy manner until the coverage requirement is met. During the process, each test case covers as many uncovered target combinations as possible, so as to minimize the number of test cases in the test suite. The IPO algorithm expands a CA both horizontally and vertically. It firstly initializes a small sub-array by enumerating all value combinations of the first t parameters, then adds an additional column so that as many target combinations are covered as possible, and then adds rows to cover the remaining uncovered combinations. The horizontal extension stage and the vertical extension stage is repeated alternatively until a CA is completed.

Both of the above strategies are greedy approaches, trying to cover as many target combinations as possible when expanding the array. In this paper, we propose a different heuristic called the balancing heuristic, which only imposes constraints upon each individual column. With this heuristic, our main algorithm generates CAs in the horizontal way: Each time it derives pseudo-Boolean constraints for the next column, and then employs a pseudo-Boolean constraint solver to obtain the solution of the new column. Simple as it is, our algorithm demonstrates much advantage over the state-of-the-art CA generators for instances of strength 2. It can generate smaller CAs, while the execution time is comparable to that of other solvers.

2 Preliminaries

Definition 1. *A* ***covering array*** $CA(N, d_1 d_2 \cdots d_k, t)$ *of strength* t *is an* $N \times k$ *array having the following two properties:*
1. There are exactly d_i *symbols in each column* i *(*$1 \leq i \leq k$*);*
2. In every $N \times t$ *sub-array, each ordered combination of symbols from the* t *columns appears at least once.*

Each column of the CA corresponds to a factor or parameter p_i ($1 \leq i \leq k$), and d_i is called the *level* of p_i.

The parameters in a CA can be combined when their levels are the same. If every parameter has the same level, the array can be denoted by $CA(N, d^k, t)$.

0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

Fig. 1. CA$(5, 2^4, 2)$

0	1	1
0	0	0
1	1	0
1	0	1
2	1	1
2	0	0

Fig. 2. MCA$(6, 3^1.2^2, 2)$

Otherwise, it is called a *mixed level covering array* (MCA) [2]. For a quick example, Fig. 1 shows an instance of $CA(5, 2^4, 2)$. In any two columns, each ordered pair of symbols occurs at least once. Similarly, an instance of $CA(6, 3^1 2^2, 2)$ is given in Fig. 2.

3 Encoding Column Restrictions as Pseudo-Boolean Constraints

As mentioned before, our algorithm constructs a CA column by column. As an initial step, the algorithm randomly generates $t-1$ columns. Assume that we have constructed m columns ($m \geq t-1$), we now discuss how to generate pseudo-Boolean constraints for column $m+1$.

A pseudo-Boolean (PB) constraint is an equation or inequality between polynomials in 0-1 variables. A linear PB clause has the form: $\sum c_i \cdot L_i \sim d$, where $c_i, d \in \mathcal{Z}$, $\sim \in \{=, <, \leq, >, \geq\}$, and L_is are literals.

Let us denote the variable at the ith entry of column $m+1$ by V_i. For each entry i and each value v ($0 \leq v < d_{m+1}$), we introduce a Boolean variable $P_{i,v}$ such that $P_{i,v} \equiv (V_i = v)$.

The first class of constraints guarantees that each entry can only take one value. For the ith entry of column $m+1$, we have:

$$\sum_{0 \leq v < d_{m+1}} P_{i,v} = 1$$

Now consider the covering property of the array. Given an array A, suppose we extract $t-1$ columns (denoted by $C_{i_1}, C_{i_2}, \ldots, C_{i_{t-1}}$) from A and denote the sub-array by A_s. The p-set corresponding to a row vector $\boldsymbol{v}$ is the set of row indices i such that i_{th} row of A_s is $\boldsymbol{v}$. Apparently, there are $d_{i_1} \times \cdots \times d_{i_{t-1}}$ mutually exclusive p-sets induced by the sub-array A_s.

Example 1. Figure 3 illustrates the p-sets from the $CA(5, 2^4, 2)$ in Fig. 1. Each column induces 2 p-sets. More specifically, the p-set {1, 2} is induced by the sub-array formed by column 1 because row 1 and row 2 in the sub-array share the same row vector $\langle 0 \rangle$.

Column	*P-set*
1	{1,2} {3,4,5}
2	{1,3} {2,4,5}
3	{1,4} {2,3,5}
4	{1,5} {2,3,4}

Fig. 3. p-sets from CA(5,2^4,2)

0	0	0	1	0
0	0	0	0	1
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
1	1	0	1	1
1	0	1	1	1
0	1	1	1	1
1	1	1	1	0
1	1	1	0	1

Fig. 4. Balanced $CA(10, 2^5, 3)$

Theorem 1. *An $N \times (m+1)$ array is a $CA(N, d_1 \ldots d_{m+1}, t)$* ***iff*** *the array formed by its first m columns is a $CA(N, d_1 \ldots d_m, t)$, and for each $N \times (t-1)$ sub-array of the first m columns, the d_{m+1} symbols in column $m+1$ all appear within the rows indexed by each p-set induced by the sub-array.*

Proof. For the **if** way of the implication: Suppose an $N \times (m+1)$ array M satisfies the latter condition and denote the array formed by the first m columns of M by M'. For any t columns extracted from M, we are to show all combinations of symbols from these columns are covered in rows. If column $m+1$ isn't chosen, the conclusion obviously holds since all the t columns are contained in M' and M' is a $\mathrm{CA}(N, d_1 \ldots d_m, t)$. Otherwise, denote the other $t-1$ columns from M' by $C_{i_1}, \ldots, C_{i_{t-1}}$, and label an arbitrary combination of symbols from these columns and column $m+1$ as $\langle v_{i_1}, \ldots, v_{i_{t-1}}, v_{m+1}\rangle$. Since M' is also a $\mathrm{CA}(N, d_1 \ldots d_m, t-1)$, it covers $\langle v_{i_1}, \ldots, v_{i_{t-1}}\rangle$. Among all the p-sets induced by the sub-array formed by the columns $C_{i_1}, \ldots, C_{i_{t-1}}$, we denote the one corresponding to the vector $\langle v_{i_1}, \ldots, v_{i_{t-1}}\rangle$ by T. From the presumption we know that v_{m+1} of column $m+1$ appears within the rows indexed by T. Hence the vector $\langle v_{i_1}, \ldots, v_{i_{t-1}}, v_{m+1}\rangle$ is covered in M. By definition, M is a $\mathrm{CA}(N, d_1 \ldots d_{m+1}, t)$.

For the **only if** way of implication: Suppose the array M is a CA $(N, d_1 \ldots d_{m+1}, t)$. Let the array formed by the first m columns be M'. Obviously M' is a CA $(N, d_1 \ldots d_m, t)$. Now suppose there exists a p-set T in the rows of which the symbol from column $m+1$, namely v_{m+1} does not appear. We denote the symbol combination corresponding to T by $\langle v_{i_1}, \ldots, v_{i_{t-1}}\rangle$. Then the symbol combination $\langle v_{i_1}, \ldots, v_{i_{t-1}}, v_{m+1}\rangle$ is not covered in M, contradicting the presumption that M is a CA of strength t. □

According to Theorem 1, firstly we should calculate all p-sets induced by all $N \times (t-1)$ sub-arrays from the first m columns, then the constraints for the covering property of column $m+1$ can be obtained directly. In practice, we may add the p-sets incrementally to a stack as the columns expand, so that re-computation can be avoided.

Once all the p-sets are computed, it's easy to translate the constraints to pseudo-Boolean constraints. For an arbitrary p-set T, each of the d_{m+1} symbols from column $m+1$ should appear at least once in the rows indexed by T. The constraint of a p-set is naturally represented by the following pseudo-Boolean clauses:

$$\bigwedge_{0 \le v < d_{m+1}} \sum_{i \in T} P_{i,v} \ge 1$$

4 The Balancing Heuristic

By observing many CA instances, we find that the CAs of the optimal sizes are likely to have the following property:

Definition 2. *A CA is* ***balanced*** *if in each column, all symbols occur nearly equally often. Formally, for a $CA(N, d_1 d_2 \cdots d_k, t)$, denote the number of occurrences of symbol v in column j by $O_j(v)$. If for any pair $\langle v, j\rangle$, $\left\lfloor \frac{N}{d_j} \right\rfloor \le O_j(v) \le \left\lceil \frac{N}{d_j} \right\rceil$, then the $CA(N, d_1 d_2 \cdots d_k, t)$ is balanced.*

The balancing property indicates that the difference of the numbers of occurrences of any two symbols within the same column is no larger than 1. For example, the CA$(10, 2^5, 3)$ in Fig. 4 is balanced.

Our algorithm employs the balancing heuristic, searching for balanced CAs so as to enhance the probability of finding optimal CAs. Hence in addition to the constraints encoding the CA properties, there are constraints encoding the balancing property. For each value v ($1 \leq v < d_{m+1}$), we have:

$$\left\lfloor \frac{N}{d_{m+1}} \right\rfloor \leq \sum_{1 \leq i \leq N} P_{i,v} \leq \left\lceil \frac{N}{d_{m+1}} \right\rceil$$

Currently we are unable to prove the rationality of the balancing heuristic, nevertheless we can provide some explanation which may shed light on this issue. Unlike the greedy strategies, which aim to locally optimize the current CA solution, the balancing heuristic is concerned with the expansibility of the current solution. The symbol distribution in column $m+1$ will influence the column that follows (if any). Take a $CA(N, d^k, 2)$ for example. According to Theorem 1, the d symbols in column $m+2$ must all appear within each p-set induced by column $m+1$. Intuitively the smallest p-set is the most restrictive one. Since the sum of the sizes of all these p-sets equals to N, it is better if all these p-sets are nearly of the same size, which means column $m+1$ is balanced.

5 Implementation and Experimental Evaluation

The column generation approach requires that the size of a CA is specified beforehand. To overcome this limitation, we employ the binary search strategy to determine the optimal size of the CA. Our tool `CAB` (CA searcher with Balancing heuristic) was implemented in the C++ programming language and integrated with the pseudo-Boolean constraint Solver `clasp` v3.1.1 [16].

For comparison, we selected three state-of-the-art CA generators, `PICT` [3], `ACTS` [9] and `CASA` [4]. `PICT` is a widely used test case generator developed by Jacek Czerwonka at Microsoft Corporation. The core generation algorithm of `PICT` adopts the one-test-at-a-time strategy. `ACTS` is a powerful test generation tool which implements the IPO algorithms. `CASA` is a CA generator based on Simulated Annealing. We compared `CAB` with these tools on various benchmarks. The experiments were conducted on an Intel 1.7 GHZ Core Duo i5-4210U PC with virtual Linux2.6 OS. Timeout (TO) means more than one hour.

The experimental results on pure-level CAs of strength 2 with level d ranging from 2 to 4 are illustrated in Tables 1, 2 and 3. Given the number of parameters (denoted by k), the number of rows (denoted by N) generated by each tool and the running time (denoted by T, measured in milliseconds) are listed. It can be seen that in most cases, `CAB` produces the best results, covering the pairwise interactions of k parameters with the least number of rows. The running times of `CAB` are also reasonable. In particular, for strength $t = 2$ and level $d = 2$, `CAB` is able to obtain the best results in dramatically shorter time. The results

Table 1. CA(N,2^k,2)

K	CAB		CASA		PICT		ACTS	
	N	T	N	T	N	T	N	T
3	**4**	2	**4**	80	**4**	84	**4**	72
4	**5**	2	**5**	50	**5**	25	6	0
10	**6**	2	**6**	140	9	20	10	4
15	**7**	2	**7**	420	10	37	10	4
35	**8**	2	9	1010	12	29	14	8
56	**9**	2	10	2720	14	28	14	8
126	**10**	2	11	19130	16	185	16	28
210	**11**	2	12	50670	18	376	18	8
462	**12**	2	14	303030	20	872	20	624
792	**13**	4	15	1785650	22	2816	22	2965
1716	**14**	5	-	TO	24	16221	24	39564
3003	**15**	9	-	TO	26	59832	26	190143
6435	**16**	15	-	TO	28	436264	28	1391365
11440	**17**	37	-	TO	-	crash	-	crach
24310	**18**	71	-	TO	-	crash	-	crash
43758	**19**	124	-	TO	-	crash	-	crash
92378	**20**	251	-	TO	-	crash	-	crash
167960	**21**	474	-	TO	-	crash	-	crash
352716	**22**	1036	-	TO	-	crash	-	crash
646646	**23**	2248	-	TO	-	crash	-	crash
1352078	**24**	4457	-	TO	-	crash	-	crash
2496144	**25**	9293	-	TO	-	crash	-	crash
5200300	**26**	23993	-	TO	-	crash	-	crash
9657700	**27**	45162	-	TO	-	crash	-	crash

Table 2. CA(N,3^k,2)

K	CAB		CASA		PICT		ACTS	
	N	T	N	T	N	T	N	T
4	**9**	140	**9**	29	13	16	**9**	0
5	12	90	**11**	100	12	21	15	4
6	13	70	**12**	360	14	16	15	0
9	**14**	120	15	340	17	16	15	4
12	**15**	150	16	1070	19	16	19	0
18	**17**	230	**17**	12350	22	29	21	0
24	**18**	350	19	5080	23	17	24	0
30	**19**	1190	21	3600	25	21	25	4
39	**20**	1560	21	38540	27	23	26	4
52	**21**	2830	22	111140	29	36	28	4
64	**22**	4600	23	148610	30	40	29	12
83	**23**	10620	25	108110	33	109	31	16
117	**24**	20330	26	1376720	34	112	33	32
137	**25**	39280	27	855600	35	152	34	48
170	**26**	84420	28	2733730	37	268	36	84
248	**27**	167000	-	TO	39	537	37	184
289	**28**	513110	-	TO	39	800	39	264
361	**29**	996630	-	TO	40	1201	40	456
476	**30**	919390	-	TO	42	2336	42	888

Table 3. CA(N,4^k,2)

K	CAB		CASA		PICT		ACTS	
	N	T	N	T	N	T	N	T
3	**16**	160	**16**	140	17	17	**16**	0
5	19	80	**16**	190	22	41	24	0
6	21	160	**19**	1980	25	36	24	0
7	23	260	**22**	1660	27	24	32	0
8	**24**	380	**24**	4330	28	27	32	0
9	**25**	760	27	820	30	23	32	0
10	**26**	6350	**26**	5290	31	28	33	0
12	**27**	5360	28	2720	34	25	33	0
15	**28**	13830	29	41270	35	28	36	0
16	**29**	10630	**29**	90910	36	35	38	0
18	**30**	151950	**30**	105930	36	48	41	4
21	**31**	169020	**31**	457320	39	57	41	4
26	**32**	313420	33	492560	42	68	43	4

Table 4. CA(N,2^k,3)

K	CAB		CASA		PICT		ACTS	
	N	T	N	T	N	T	N	T
4	**8**	40	**8**	140	**8**	20	**8**	0
5	**10**	50	**10**	90	13	28	12	0
8	**12**	90	**12**	450	16	24	18	0
10	16	120	**12**	1700	18	28	20	0
12	18	310	**16**	2410	19	28	22	0
14	22	600	**18**	6890	22	36	25	4
15	22	540	**19**	1890	23	29	26	4
17	26	940	**20**	5640	23	20	26	0
20	26	1210	**21**	24040	26	32	27	4
24	28	1860	**23**	150800	29	44	29	8

obtained by `CASA` are quite close to that of `CAB`, but it often takes much more running times. `ACTS` and `PICT` are the fastest solvers in general. However, the CAs found by them are usually larger.

Table 5. Mixed Covering Array

Model	Description	CAB		CASA		PICT		ACTS	
		N	T	N	T	N	T	N	T
Apache	CA($2^{158}3^{8}4^{4}5^{1}6^{1}$;2)	**30**	3490	33	44230	32	151	33	172
Bugzilla	CA($2^{49}3^{1}4^{2}$;2)	**16**	360	**16**	2660	17	24	18	4
gcc	CA($2^{189}3^{10}$;2)	**15**	9780	17	71710	20	133	20	76
SpinS	CA($2^{13}4^{5}$;2)	19	320	**16**	2480	23	13	24	0
SpinV	CA($2^{42}3^{2}4^{11}$;2)	**27**	6460	28	7010	32	28	33	8
Banking1	CA($3^{4}4^{1}$;2)	13	70	**12**	130	16	17	15	0
Banking2	CA($2^{14}4^{1}$;2)	**10**	180	**10**	210	12	12	**10**	0
CommProtocol	CA($2^{10}7^{1}$;2)	**14**	40	**14**	320	16	15	**14**	0
Concurrency	CA(2^{5};2)	**6**	0	**6**	60	7	11	**6**	0
Healthcare1	CA($2^{6}3^{2}5^{1}6^{1}$;2)	**30**	40	**30**	220	**30**	16	**30**	0
Healthcare2	CA($2^{5}3^{6}4^{1}$;2)	**15**	140	**15**	210	18	11	**15**	0
Healthcare3	CA($2^{16}3^{6}4^{5}5^{1}6^{1}$;2)	**30**	250	32	1590	35	19	32	4
Healthcare4	CA($2^{13}3^{12}4^{6}5^{2}6^{1}7^{1}$;2)	**42**	320	**42**	9150	47	27	44	8
Insurance	CA($2^{6}3^{1}5^{1}6^{2}11^{1}13^{1}17^{1}31^{1}$;2)	**527**	26420	**527**	215930	**527**	92	**527**	4
NetworkMgmt	CA($2^{2}4^{1}5^{3}10^{2}11^{1}$;2)	**110**	28610	**110**	98160	118	28	**110**	0
ProcessorComm1	CA($2^{3}3^{6}4^{6}$;2)	**21**	270	22	5390	26	16	26	4
ProcessorComm2	CA($2^{3}3^{12}4^{8}5^{2}$;2)	**28**	2690	29	32810	36	21	37	4
Services	CA($2^{3}3^{4}5^{2}8^{2}10^{2}$;2)	**100**	350	102	5810	101	31	102	4
Storage1	CA($2^{1}3^{1}4^{1}5^{1}$;2)	**20**	10	**20**	100	**20**	28	**20**	71
Storage2	CA($3^{4}6^{1}$;2)	**18**	20	**18**	170	19	32	**18**	0
Storage3	CA($2^{9}3^{1}5^{3}6^{1}8^{1}$;2)	**48**	80	**48**	610	52	53	51	0
Storage4	CA($2^{5}3^{7}4^{1}5^{2}6^{2}7^{1}10^{1}13^{1}$;2)	**130**	2330	132	7770	**130**	68	134	4
Storage5	CA($2^{5}3^{8}5^{3}6^{2}8^{1}9^{1}10^{2}11^{1}$;2)	**113**	521870	**113**	243140	123	84	119	4
SystemMgmt	CA($2^{5}3^{4}5^{1}$;2)	**15**	50	**15**	310	19	2	16	0
Telecom	CA($2^{5}3^{1}4^{2}5^{1}6^{1}$;2)	**30**	40	**30**	4550	**30**	23	**30**	4

Table 4 demonstrates the results on CAs of strength 3 and level 2. `CASA` produces the smallest CAs in all cases, although its execution times are significantly longer than those of the other tools. Compared with `PICT` and `ACTS`, `CAB` usually obtains smaller CAs in longer times.

We also performed experiments on a number of MCAs, as listed in Table 5. These MCAs are derived from some benchmark SUT models in previous papers on combinatorial testing, see [12] for example. `CAB` outperforms `CASA` in both the quality of results and the execution times. `PICT` and `ACTS` are very fast, and both fail to produce the smallest CAs in most occasions. Interestingly, for many cases in Table 5, the smallest size (in bold type) happens to be the lower bound of the optimal size in theory (the product of t largest levels).

6 Related Work

The computational methods for covering array generation have been extensively studied in literature. Besides the aforementioned one-test-a-time strategy and IPO strategy, there are also some metaheuristic search and evolutionary algorithms applied to the automatic generation of CAs, including simulated annealing, which is the core algorithm of CASA, genetic algorithms, and particle swarm optimization. For a detailed review, one can refer to [11]. Recently, an efficient local search algorithm has been proposed for generating CAs with constraints [14].

Constraint Solving techniques are also used for automatic generation of CAs. Hinch et al. [5] developed constraint programming models which exploited global constraints and symmetry breaking constraints. They also studied the local search algorithm for a SAT-encoding of the model. Yan and Zhang developed another backtrack search tool for finding CAs [10]. A kind of balancing heuristic is applied to value-tuples in the CA, so as to prune the search space. In particular, pseudo-Boolean constraint solving has also been employed to generate orthogonal arrays (OA), which can be viewed as a special class of CA [15].

7 Conclusion

In this paper, we propose a non-backtracking algorithm which generates covering arrays column by column. It integrates a pseudo-Boolean constraint solver for column generation, and adopts a new heuristic named the balancing heuristic to guide the searching procedure. The principle of balancing heuristic is very different from the greedy strategies such as one-test-at-a-time and IPO, which have been widely employed by CA generators. The balancing heuristic suggests that, rather than locally optimizing the current CA solution, the algorithm should generate balanced columns so that the current solution is more likely to be horizontally expanded. Simple as it is, our algorithm demonstrates advantage over the state-of-the-art CA generators for instances of strength 2. It can generate smaller CAs in reasonable time. In the future, we will study how to improve the performance of our tool CAB on CAs with higher strengths.

References

1. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Software Eng. **23**(7), 437–444 (1997)
2. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: Proceedings of ICSE 2003, pp. 38–48 (2003)
3. Czerwonka, J.: Pairwise testing in the real world. In: Proceedings of the 24th Pacific Northwest Software Quality Conference (PNSQC 2006), pp. 419–430 (2006)
4. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: An improved meta-heuristic search for constrained interaction testing. In: Proceedings of the 1st International Symposium on Search Based Software Engineering (SBSE 2009), pp. 13–22 (2009)
5. Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints **11**(2–3), 199–219 (2006)
6. Kuhn, D.R., Michael, J.R.: An investigation of the applicability of design of experiments to software testing. In: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop (2002)

7. Lei, Y., Tai, K.C.: In-parameter-order: a test generation strategy for pair-wise testing. In: Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE 1998), pp. 254–261. IEEE Computer Society (1998)
8. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG: a general strategy for T-way software testing. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2007), pp. 549–556. IEEE (2007)
9. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test. Verification Reliab. **18**(3), 125–148 (2008)
10. Yan, J., Zhang, J.: A backtracking search tool for constructing combinatorial test suites. J. Syst. Softw. **81**(10), 1681–1693 (2008)
11. Zhang, J., Zhang, Z., Ma, F.: Automatic Generation of Combinatorial Test Data. Springer Briefs in Computer Science. Springer, Heidelberg (2014). ISBN 978-3-662-43428-4
12. Zhang, Z., Yan, J., Zhao, Y., Zhang, J.: Generating combinatorial test suite using combinatorial optimization. J. Syst. Softw. **98**, 191–207 (2014)
13. Hartman, A., Raskin, L.: Problems and algorithms for covering arrays. Discrete Math. **284**, 149–156 (2004)
14. Lin, J., Luo, C., Cai, S., Su, K., Hao, D., Zhang, L.: TCA: an efficient two-mode meta-heuristic algorithm for combinatorial test generation. In: Proceedings of ASE 2015, pp. 494–505 (2015)
15. Ma, F., Zhang, J.: Finding orthogonal arrays using satisfiability checkers and symmetry breaking constraints. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 247–259. Springer, Heidelberg (2008)
16. The clasp webpage. http://potassco.sourceforge.net/