Implementing arrays for fault detection in R software

Ulrike Grömping

14 December 2024,

CMStatistics London

BHT
Berliner Hochschule
für Technik

Contents

Upcoming Sabbatical

Goal: implement an R package that provides covering arrays (CAs) with the most important tools around them

I am new to that field.

I will share my fresh insights on CAs:

- their usage,
- algorithms for creating them,
- quality criteria,
- practical aspects,

and my thoughts regarding implementation.

Test situation

Assumption: a system under test (SUT) whose behavior is driven by m factors $F_1, \dots, F_m$, where F_i has s_i levels $v_{i1}, \dots, v_{is_i}$.

The test suite D consists of N test runs, i.e., N particular level combinations of the m factors.

For each test, the outcome is a “pass” or a “fail”.

We will assume that - in the absence of mistakes - the same test run would yield the same outcome again and again (e.g., in software testing), i.e., testing itself is deterministic.

Fault detection as a screening task

Faults can be caused by pairs, triples, quadruples, ... of factors.

> ***Interaction Rule***: *Most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors. Kuhn et al. (2010)*

Different questions:

- **Is the product fault-free?**
  sufficient to pass all runs of certain coverage quality
  (e.g., all value pairs or all value triples) → Covering Arrays
- **What conditions lead to faults?**
  more difficult, distinguishing different potential candidates required
  - either engineering judgment combined with subsequent confirmation experiments permitted
  - or experiment that directly leads to identification of failure root causes (further types of arrays)

ℓway Interactions

An ℓway interaction is an ℓ-dimensional value combination for ℓ factors, i.e., e.g.,

A=0, B=0
A=1, B=0
A=0, B=1
A=1, B=1

are four 2way-interactions (2IAs).

Covering arrays (CAs)

A **C**overing **A**rray (CA) of strength *t* is a test set (=experimental design) *D* for which each *t*IA appears at least once.

Notation:

- $\text{CA}\left(N, t, m, s_1^{m1} \ldots s_k^{mk}\right)$ with $m_1 + \cdots + m_k = m$,
- $\text{CA}\left(N, t, m, \left(s_1, \ldots, s_m\right)\right)$,
- $\text{CA}\left(N, t, m, s\right)$ (uniform arrays)

Orthogonal array (OA) vs. CA of strength *t*:

OA requires each *t*IA for a given *t*-tuple of factors to appear **the same number** of times;
"**at least once**" (CA) is much less demanding.

Impression of run savings

Example
six factors with 2,2,3,5,7,8 levels
strength $t = 2$:

Total number of 2IAs to be covered

$$\sum_{i=1}^{5} \sum_{j=i+1}^{6} s_i s_j = 287$$

3360 runs – Full factorial (reference)
1680 runs – OA
56 runs – CA ($7 \cdot 8 = 56$)

We can save even more runs by using many more 2-level factors in those 56 runs (at least 600 2-level factors are possible, see also below).

An example CA D (a CA$\left(56, 2, 6, \left(2, 2, 3, 5, 7, 8\right)\right)$)

Run	A	B	C	D	E	F	Run	A	B	C	D	E	F
1	0	0	0	0	0	0	29	1	1	1	3	3	4
2	1	0	1	1	0	1	30	0	1	2	4	3	5
3	0	1	2	2	0	2	31	1	0	0	0	3	6
4	1	1	0	3	0	3	32	0	0	1	1	3	7
5	0	0	1	4	0	4	33	1	1	2	2	4	0
6	1	0	2	0	0	5	34	0	1	0	3	4	1
7	0	1	0	1	0	6	35	1	0	1	4	4	2
8	1	1	1	2	0	7	36	0	0	2	0	4	3
9	0	0	2	3	1	0	37	1	1	0	1	4	4
10	1	0	0	4	1	1	38	0	1	1	2	4	5
11	0	1	1	0	1	2	39	1	0	2	3	4	6
12	1	1	2	1	1	3	40	0	0	0	4	4	7
13	0	0	0	2	1	4	41	1	1	1	0	5	0
14	1	0	1	3	1	5	42	0	1	2	1	5	1
15	0	1	2	4	1	6	43	1	0	0	2	5	2
16	1	1	0	0	1	7	44	0	0	1	3	5	3
17	0	0	1	1	2	0	45	1	1	2	4	5	4
18	1	0	2	2	2	1	46	0	1	0	0	5	5
19	0	1	0	3	2	2	47	1	0	1	1	5	6
20	1	1	1	4	2	3	48	0	0	2	2	5	7
21	0	0	2	0	2	4	49	1	1	0	3	6	0
22	1	0	0	1	2	5	50	0	1	1	4	6	1
23	0	1	1	2	2	6	51	1	0	2	0	6	2
24	1	1	2	3	2	7	52	0	0	0	1	6	3
25	0	0	0	4	3	0	53	1	1	1	2	6	4
26	1	0	1	0	3	1	54	0	1	2	3	6	5
27	0	1	2	1	3	2	55	1	0	0	4	6	6
28	1	1	0	2	3	3	56	0	0	1	0	6	7

Usage principle

- Sets $\mathcal{F}$ and $\mathcal{P}$ hold *t*IAs that occur in failed runs or passed runs, respectively.
- Potential Failure Generating 2IAs are those in $\mathcal{F} - \mathcal{P}$.

Examples with design *D* ($\text{CA}\left(56, 2, 6, \left(2, 2, 3, 5, 7, 8\right)\right)$, $t = 2$):

- all runs passed: no failures from 2IAs
- Run 10 failed, all others passed: There is only a single 2IA in $\mathcal{F} - \mathcal{P}$, namely levels of factors 5 and 6 both equal to 1. Thus, the 2IA is uniquely identified.
- Run 36 failed, all others passed: There are three 2IAs in $\mathcal{F} - \mathcal{P}$, namely
  factor 4 level 0 with factor 5 level 4,
  factor 4 level 0 with factor 6 level 3,
  factor 5 level 4 with factor 6 level 3

For non-unique answers, engineering judgment or further experimentation can help.

Quality criteria: Coverage behavior for the two 56 run CAs

Table 2: Coverage behavior of the two 56 run CAs (in pct))

two 2-level columns						600 2-level columns	
		Coverage				Coverage (based on JMP)	
ell	n.IAs	total	ave.	simple	n.projs	total (approx.)	ave.
2	287	100.00	100.00	100	15	100.00	100.00
3	1529	55.66	72.62	30	20	99.51	99.85
4	4296	19.04	29.87	0	15	95.61	96.89
5	6052	5.55	7.50	0	6	79.88	82.21
6	3360	1.67	1.67	0	1	55.11	57.53

More interesting quality criteria?

- irritating that CA$(56, 2, 604, 2^{600}3^{1}5^{1}7^{1}8^{1})$ looks better on the quality criteria than CA$(56, 2, 6, 2^{2}3^{1}5^{1}7^{1}8^{1})$
- usability for identifying fault-generating IAs would be of interest (LAs, DAs)

Outline of remaining talk

- approaches for creating CAs
- practical aspects
- thoughts on the planned project

Contents

Strategies

- mathematical methods for uniform CAs (e.g., Torres-Jimenez et al. (2019)) and (fewer) for mixed level CAs (e.g., Akhtar et al. (2024))
- search algorithms, as, e.g., described in Leithner et al. (2024)
  - IPO = in parameter order:
    - starts with an array in the first *t* columns,
    - extends it horizontally with further columns,
    - and - where necessary - vertically with further rows
    - various variants ((F)IPOG, (F)IPOG-F, (F)IPOG-F2)
  - OTAT = one test at a time:
    - add tests one at a time,
    - maximize the number of additionally covered *t*-way interactions with each additional test (greedy)
  - metaheuristics and postoptimization
    - tabu search
    - simulated annealing
    - optimizing (=reducing the number of tests) an existing array with one of these methods

Software

Specific search tools, e.g.

- **CAgen** (by an Austrian team around Dimitris Simos, Wagner et al. (2020), https: //srd.sba-research.org/tools/cagen/#/help; FIPOG, FIPOG-F, FIPOG-F2),
- **CTwedge** (by an Italian team from Bergamo, Gargantini and Radavelli (2018))

typically handle constraints,
these two offer free web versions.

Further software, e.g.

- allpairspy (Python package) by Hombashi (2023)
- JMPPro implementation (JMP Statistical Discovery LLC (2024)), commercial, free access for students and academics

Catalogues of arrays

The catalogues cover uniform arrays only:

- NIST catalogue of CAs:
  21 964 **actual arrays** for strengths 2 to 6, 2 to 6 levels, and up to 2 000 columns, size up to 125 683 runs ("NIST Covering Array Tables" (n.d.); arrays are downloadable as zip-files)
- Table of smallest possible sizes (Colbourn (n.d.)), **without the actual arrays** but with, somewhat cryptic, references;
  13 641 entries, strength 2 to 6, for 2 to 25 levels, and up to 10 000 columns; still 2 884 entries for up to 6 levels, with size up to 13 759 798)

Contents

Numbers of levels

Uniform arrays will rarely suffice!

Constraints

It will often not be possible or desirable to accommodate all level combinations.

- a certain level combination is technically incompatible
- a certain level combination is not sold for other reasons
- ...

It can also happen that there is not a single valid level for a factor, which is called **unsatisfiable constraint** (e.g., in JMP, see next slide).

Unsatisfiable constraints (created with JMPPro)

Run	A	B	C	D	E	F	Run	A	B	C	D	E	F
1	0	0	2	1	5	0	29	0	0	2	4	1	6
2	1	1	2	4	2	1	30	1	0	2	3	6	5
3	1	0	1	1	0	1	31	0	1	2	2	0	2
4	1	0	2	3	3	0	32	1	0	2	2	4	7
5	1	0	1	1	3	4	33	0	1	1	2	3	3
6	0	1	2	3	1	2	34	1	1	1	3	4	1
7	1	1	2	3	6	1	35	0	1	1	3	4	6
8	0	0	1	4	6	7	36	1	1	1	4	0	0
9	0	0	1	4	5	3	37	1	0	2	0	3	5
10	1	1	1	2	1	0	38	1	1	1	0	6	0
11	0	0	1	0	0	5	39	1	0	2	3	3	7
12	0	1	2	3	2	4	40	1	0	1	4	4	5
13	1	0	2	1	1	7	41	1	0	2	4	3	2
14	0	1	1	2	4	4	42	1	0	1	1	4	3
15	1	0	1	2	2	5	43	1	1	2	0	1	4
16	1	1	2	0	2	2	44	0	1	1	3	6	4
17	1	1	2	1	5	5	45	1	0	2	4	0	4
18	0	1	1	0	5	6	46	0	1	2	4	1	3
19	1	1	2	0	2	3	47	1	0	1	0	3	6
20	1	0	2	1	2	6	48	1	1	1	1	2	7
21	0	0	1	3	0	3	49	0	0	0	.	2	0
22	0	1	2	2	6	6	50	1	1	0	.	0	7
23	1	0	1	1	6	2	51	0	1	0	.	1	5
24	0	1	1	0	1	1	52	0	0	0	.	0	6
25	0	1	1	3	5	2	53	1	1	0	.	5	4
26	1	1	2	0	5	7	54	0	0	0	.	4	2
27	0	0	1	2	5	1	55	1	0	0	.	6	3
28	0	0	2	0	4	0	56	0	0	0	.	3	1

Don’t care values (created with CAgen)

Run	A	B	C	D	E	F	Run	A	B	C	D	E	F
1	0	0	0	0	0	0	29	*	0	1	4	0	4
2	1	1	1	1	1	0	30	*	*	*	0	1	4
3	*	*	2	2	2	0	31	*	*	*	1	2	4
4	*	*	*	3	3	0	32	0	1	2	2	3	4
5	*	*	2	4	4	0	33	1	*	0	3	4	4
6	*	*	*	0	5	0	34	*	*	*	*	5	4
7	*	*	*	1	6	0	35	*	*	*	*	6	4
8	*	*	2	1	0	1	36	*	*	0	1	0	5
9	*	*	0	2	1	1	37	*	*	*	2	1	5
10	0	0	1	3	2	1	38	*	*	*	3	2	5
11	*	*	*	4	3	1	39	*	*	*	*	3	5
12	*	1	1	0	4	1	40	*	*	*	*	4	5
13	1	0	*	2	5	1	41	0	1	1	4	5	5
14	*	*	*	3	6	1	42	1	0	2	0	6	5
15	*	*	1	2	0	2	43	*	*	2	4	0	6
16	*	*	2	3	1	2	44	*	*	*	0	1	6
17	1	1	0	4	2	2	45	*	*	*	3	2	6
18	*	*	*	0	3	2	46	1	0	1	*	3	6
19	0	0	*	1	4	2	47	*	*	*	*	4	6
20	*	*	*	*	5	2	48	*	*	0	1	5	6
21	*	*	*	*	6	2	49	0	1	*	2	6	6
22	1	1	*	3	0	3	50	1	1	1	0	0	7
23	0	0	*	4	1	3	51	0	0	2	1	1	7
24	*	*	*	0	2	3	52	*	*	*	2	2	7
25	*	*	0	1	3	3	53	*	*	*	*	3	7
26	*	*	*	2	4	3	54	*	*	*	*	4	7
27	*	*	2	*	5	3	55	*	*	*	3	5	7
28	*	*	1	*	6	3	56	*	*	0	4	6	7

Don't care values

are useful during design creation

- for high expandability
- for potential optimization of quality metrics
- ...

Varying strength CAs

It might be of interest to have varying strength requirements for different subsets of factors in the SUT.

e.g. uniform array of relatively high strength for many factors with few levels, combined with a few factors at more levels (added with a lower strength requirement).

Contents

Starting points for own implementations

- The catalogue “NIST Covering Array Tables” (n.d.) of Tables of reasonably small (IPOG-F generated) uniform CAs ($s = 2, \dots, 6, t = 2, \dots, 6$ without $s = 6$ with $t = 6$);
- the Colbourn tables in conjunction with review papers on mathematical algorithms, e.g. Torres-Jimenez et al. (2019)
- papers on mathematical algorithms for mixed CAs (e.g., Akhtar et al. (2024))
- open source Python code for - possibly - transferring

Helpful for comparisons:
free CA generation tools

- CAgen web, CAgen CLI (free academic use on request)
- CTwedge web (free),
- JMPPro (free academic use)

Example for using the Colbourn tables

- Tables provide combinations of k, t, s, N plus some reference, e.g. “Cyclotomy (Colbourn)”
- **Cyclotomy** construction (Colbourn or otherwise) is behind 183 of the 2884 entries for up to 6 levels (6.35%)
- Colbourn (2010) provides several related constructions (1, 2, 3, 3a, 3b, 4, 4a, 4b) around cyclotomic start vectors (based on Galois field logarithms).

Current state:

- figured out which construction for which entry
- created several arrays and confirmed their coverage properties

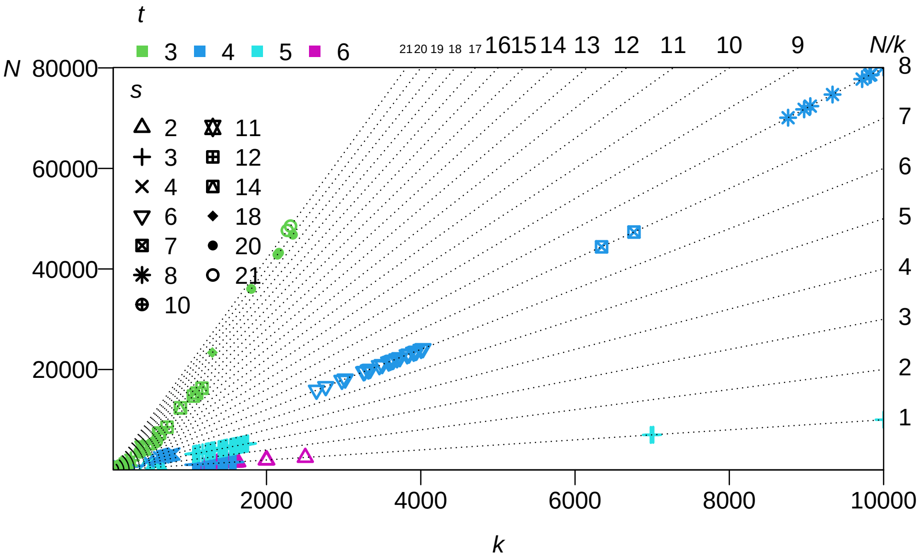
Expectation:

- most of these arrays can be created
- creation is fast, even for large arrays

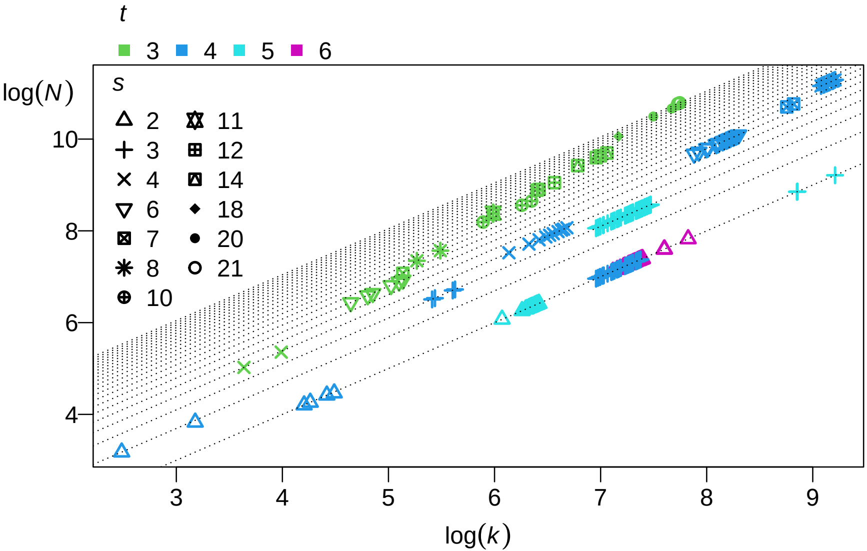
Example continued: Cyclotomy constructions

- whether or not a construction yields a CA for a given prime or prime power can only be decided by **combinatorial checks** of relatively complicated conditions (mathematically proven general bounds way too large)
- these are still much **much less demanding than checking coverage brute force**;
  e.g., a brute force full check of strength 4 coverage a CA$\left(1051, 4, 3^{1051}\right)$ took 3.5 hours on a powerful machine with 30 parallel threads.
- checks are needed, if only for programming mistakes;
  current strategy for routine checks: sampling subgroups of columns

Example continued: Cyclotomy constructions



Example continued: Cyclotomy constructions



Thoughts on planned project

- Implement *mathematical* algorithms for good uniform CAs, based on Colbourn tables (sounds easier than it is)
- These can be building blocks for more practically relevant arrays.
- Implement mathematical algorithms for mixed level CAs
- It might be worth while to try and supplement the Colbourn tables with actual arrays and/or pseudo codes of algorithms.
- Identify and implement useful algorithms for extending uniform arrays with a few different columns?
- Is it worth while for applications to implement LAs and DAs?
- Resource-intensive search tools can possibly be accessed via an API from R (e.g., the command line interface of CAgen, which is, however, not freely available for everybody)
- Are there real-world good practice examples for my benefit - focusing on usage of CAs (LAs, DAs) rather than the entire software testing cycle ?

Many questions

???

Any advice is highly appreciated.

Contents

References

Akhtar, Y., Colbourn, C. J., and Syrotiuk, V. R. (2024), “Mixed-level covering, locating, and detecting arrays via cyclotomy,” in *Combinatorics, graph theory and computing*, eds. F. Hoffman, S. Holliday, Z. Rosen, F. Shahrokhi, and J. Wierman, Cham: Springer International Publishing, pp. 37–50.

Colbourn, C. J. (2010), “Covering arrays from cyclotomy,” *Designs, Codes and Cryptography*, 55, 201–219. https://doi.org/10.1007/s10623-009-9333-8.

Colbourn, C. J. (n.d.). “Covering array tables: 2 ≤v ≤25, 2 ≤t≤6, t≤k ≤10000, 2005–23,” Available at https://www.public.asu .edu /~ccolbou /src /tabby.

Gargantini, A., and Radavelli, M. (2018), “Migrating Combinatorial Interaction Test Modeling and Generation to the Web,” in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Vasteras: IEEE, pp. 308–317. https://doi.org/10.1109/ICSTW.2018.00066.

Hombashi, T. (2023), “allpairspy.”

JMP Statistical Discovery LLC (2024), “Design of experiments guide,” Section “Covering Arrays.”

Kuhn, D. R., Kacker, R. N., and Lei, Y. (2010), *Practical combinatorial testing*, Gaithersburg, MD: National Institute of Standards and Technology, pp. NIST SP 800–142. https://doi.org/10.6028/NIST.SP.800-142.

Leithner, M., Bombarda, A., Wagner, M., Gargantini, A., and Simos, D. E. (2024), “State of the CArt: evaluating covering array generators at scale,” *International Journal on Software Tools for Technology Transfer*, 26, 301–326. https://doi.org/10.1007/s10009-024-00745-2.

“NIST Covering Array Tables” (n.d.). Available at https://math.nist.gov/coveringarrays/.

Torres-Jimenez, J., Izquierdo-Marquez, I., and Avila-George, H. (2019), “Methods to construct uniform covering arrays,” *IEEE access : practical innovations, open solutions*, 7, 42774–42797. https://doi.org/10.1109/ACCESS.2019.2907057.

Wagner, M., Kleine, K., Simos, D. E., Kuhn, R., and Kacker, R. (2020), “CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays,” in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 191–200. https://doi.org/10.1109/ICSTW50294.2020.00041.