An Algorithm for Generating Very Large Covering Arrays

D. Richard Kuhn

One problem that must be solved for automated testing of large systems is producing tests for a large number of variables with a high degree of interaction. The problem of covering arrays is known to be NP-hard, so deterministic algorithms do not perform well for large numbers of variables. FireEye, which implements the In Parameter Order (IPO) algorithm, can handle a large number of variables, but its run time increases exponentially as the number of variables and interaction levels increase, and it is not clear if IPO can be parallelized. This note describes a algorithm – called Paintball – that can be parallelized, unlike other covering array algorithms, making it possible to handle a much larger number of variables than other known algorithms. The algorithm trades test case optimization for speed – it produces roughly 3% to 15% more tests (depending on run time allowed) than FireEye for 10 or more variables, but this ratio improves as the number of variables increases (see examples in Table 1). It thus complements but may not replace FireEye and other algorithms, which produce somewhat fewer tests. The significant advantage of the algorithm is that it can be distributed across any number of processors. For automated test generation this means that both test data generation and test oracle production can be run in parallel.

The algorithm is called Paintball because it is conceptually similar to firing paintballs at a wall, using a few tunable parameters, until it is completely covered. Although this would be absurdly inefficient for a small number of variables, it becomes effective for the large number of variables that must be handled for software testing. For k-way interaction and variables with v values each, the number of tests is ideally v^k, if there were no duplication of combinations among tests, a condition that is not possible to obtain for realistic examples. Optimal algorithms produce some multiple of this number. Because the number of k-way combinations among n variables, $\binom{n}{k}$, is very large for large n, randomly generated test cases can produce a high degree of coverage. The algorithm works as follows:

1. Construct an $\binom{n}{k}$ row by v^k column matrix M. Set all $M_{i,j} = 0$.
2. Matrix M is indexed by the k-way combinations of n variables; i.e., M is an associative array of the possible combinations of n variables; each combination is associated with a row of v^k cells. M will be used as a checkbox to check whether combinations have been covered.
3. Repeat d times, where d is a pre-specified duplication factor: generate v^k tests with n random values per test, resulting in a dv^k by n test case matrix T. Each test generated (each row of T), up to a configurable parameter t, is checked to determine the number of new combinations covered. The test with the highest number of new combinations covered is used. The objective of this step is to minimize the duplication of combinations covered by tests.
4. Mark combinations covered by each generated test:
   for each possible combination of variables, set $M_{i,j} = 1$, where $i =$ variable combination $<v_1 v_2 \ldots v_m>$ and $j = <v_m \ldots v_2 v_1>_v$. For example, suppose the values of the current combination of variables are 2,0,3, and 0. Then $j = 0302_4 = 50_{10}$. For variables with differing numbers of values, a similar procedure is used, with variables ordered by number of values.

5. Scan M and set c = number of combinations not covered. If c is less than an experience-based parameter for similar sized problems, then stop, else increase duplication factor: $d := d+1$ and repeat the algorithm until number of tests is minimum. This procedure generally covers 99.9% or more of the possible combinations.
6. Uncovered combinations may number from a few to 10,000 or more. These are written out as tests with one combination per test, with other variable values given as "don't care" conditions. For example, for 4-way interactions, an uncovered test may be "- 3 - - 2 5 - - 3 - -" ("-" represents don't care value).
7. Randomize the lines produced in step 6; i.e., for lines L_i in the output file, shuffle index i.
8. Reduce the tests for combinations not covered by combining them into test cases containing multiple combinations. Test cases are combined when their values for each variable are the same (i.e., $T_{1i} = T_{2i}$, all i) or when one is a "don't care" condition. For each line i in the file of uncovered combinations, compare line i with following lines $j = i+p..(i+p < n ? i+p : n)$, where p is a lookahead parameter that is some fraction of the total lines in the file. If a line is found that can be combined with line i, merge the two and resume scanning with line i+1. If no line in i+1 .. p matches line i, write out line i and resume scanning with line i+1. This procedure can normally reduce 10,000 to 100,000 tests to a several hundred, with the final reduced tests containing less than 1% "don't care" values.

Discussion: This procedure works because with many variables, each test case contains thousands of k-way combinations of variables. Another way of viewing the covering procedure is to think of each of the $C = \binom{n}{k} \times v^k$ combinations that must be covered as a separate bin. If we throw C marbles into random bins, most bins will have at least one marble. Another set of C marbles covers more bins, and so on repeated d times, until nearly all contain one or more marbles. For example, with C=20, d=8, all C combinations are covered with 99.5% probability The number of combinations produced by the algorithm is $d \times C$, and each random combination is assigned to one of the C bins, with the total number of randomly generated combinations proportional to $C \ln C$. With suitable values of d, at least one combination is assigned to each bin with a very high probability. The efficiency of this approach is increased by minimizing the number of duplicate combinations in each new test in step 3

Implementation notes: The algorithm can be distributed across any number of machines, but requires a common "checkbox" to record the combinations covered. For a large number of variables, a very large amount of disk storage would be required for the blackboard, roughly $\frac{\left(\binom{n}{k} \times v^k\right)}{8}$ bytes. For 6-way combinations of 50 variables with 6 values each, this is approximately 9 gigabytes; large but fairly easy to implement on a cluster system. With lower interaction levels or fewer values per variable, a desktop PC is adequate for 100 variables or more. Alternatively, a disk based version of the program can be used, although this increases run time from a few minutes to hours.

Calculation of d in step 3. The occupancy problem in probability addresses the question of throwing m balls into n bins at random, or in this case, the number of combinations generated randomly to cover all possible combinations. For large m and n, the proportion of n bins covered is

$p = e^{-m/n}$. The algorithm is designed to cover a high percentage of bins, $\frac{C-R}{C}$, where $C = \binom{n}{k} v^k$ is the number of combinations to be covered, and R is the remaining set that is not covered, usually a few hundred to several thousand (see step 6). From this we can determine the duplication factor d (step 3) and an upper bound on the number of tests produced:

$d \cong \ln C - \ln R$ and

$\#T = generated\ tests \leq (\ln C - \ln R) v^k$

In practice the best starting point for the duplication factor d seems to be $\ln C - \ln R - 1$.

Note from Table 1 that the difference between the number of tests produced by Paintball and FireEye declines as the number of variables increases, but this has not been tested above 50 variables because run times for FireEye become prohibitive (exceeding 24 hours) above this number. There may be relatively little difference for significantly higher numbers of variables.

4-way Interactions, 4 to 8 value variables, single processor

Num. of variables	PB2 tests	PB2 time	FE tests	FE time	PB2/FE tests
4 value					
10	771	3	684	1	112.7%
20	1367	31	1123	62	121.7%
30	1766	68	1486	570	118.8%
40	2051	365	1718	2704	119.4%
50	2260	300	1897	12958	119.1%
60					
70	2504	4989	2144	42383	116.8%
5 value					
10	1906	6	1861		102.4%
20	3493	27	3019		115.7%
25	3987	158	3185		125.2%
30	4340	385	3749		115.8%
40	5026	699	4275	7041	117.6%
50	5512	2084	4691	24352	117.5%
6 value					
10	3967	53	3858	57	102.8%
7 value					
10	7394	152	7155	134	103.3%
8 value					
10	12602	320	12173	288	103.5%

Table 1. Comparison of test generation for 4-way interactions
PB = paintball algorithm, #tests and time in seconds
FE = FireEye algorithm, #tests and time in seconds