Combinatorial Methods for Event Sequence Testing

D. Richard Kuhn*, James M. Higdon**, James F. Lawrence***, Raghu N. Kacker*

*National Institute of Standards & Technology Gaithersburg, MD	**US Air Force 46th Test Squadron Eglin AFB, FL	***Dept. of Mathematics George Mason Univ. Fairfax, VA

Abstract

Many software testing problems involve sequences. This paper presents an application of combinatorial methods to testing problems for which it is important to test multiple configurations, but also to test the order in which events occur. For example, the methods described in this paper were motivated by testing needs for systems that may accept multiple communication or sensor inputs and generate output to several communication links and other interfaces. We use combinatorial methods to generate test sequences which ensure that any t events will be tested in every possible t-way order.

1 Sequence-Covering Arrays

In testing event-driven software, the critical condition for triggering failures often is whether or not a particular event has occurred prior to a second one, not necessarily if they are back to back. This situation reflects the fact that in many cases, a particular state must be reached before a particular failure can be triggered. For example, a failure might occur when connecting device A only if device B is already connected. The methods described in this paper were developed to address testing problems of this type, using combinatorial methods to provide efficient testing. Sequence covering arrays, as defined here, ensure that any *t* events will be tested in every possible *t*-way order.

For this problem we can define a sequence-covering array, which is a set of tests that ensure all *t-way sequence*s of events have been tested. The *t* events in the sequence may be interleaved with others, but all permutations will be tested. For example, we may have a component of a factory automation system that uses certain devices interacting with a control program. We want to test the events defined in Table 1.

There are 6! = 720 possible sequences for these six events, and the system should respond correctly and safely no matter the order in which they occur. Operators may be instructed to use a particular order, but mistakes are inevitable, and should not result in injury to users or compromise the enterprise. Because setup, connections and operation of this component are manual, each test can take a considerable amount of time. It is not uncommon for system-level tests such as this to take hours to execute, monitor, and complete. We want to test this system as thoroughly as possible, but time and budget constraints do not allow for testing all possible sequences, so we will test all 3-event sequences.

With six events, *a, b, c, d, e,* and *f*, one subset of three is {*b, d, e*}, which can be arranged in six permutations: [*b d e*], [*b e d*], [*d b e*], [*d e b*], [*e b d*], [*e d b*]. A test that covers the permutation [*d b e*] is: [*a d c f b e*]; another is [*a d c b e f*]. A larger example system may have 10 devices to connect, in which case the number of permutations is 10!, or 3,628,800 tests for exhaustive testing. In that case, a 3-way sequence covering array with 14 tests covering all $10 \cdot 9 \cdot 8 = 720$ 3-way sequences is a dramatic improvement, as is 72 tests for all 4-way sequences (see Table 4).

Event	Description
a	connect air flow meter
b	connect pressure gauge
c	connect satellite link
d	connect pressure readout
e	engage drive motor
f	engage steering control

Table 1. System events

Definition. We define a sequence covering array, SCA(*N, S, t*) as an *N* x *S* matrix where entries are from a finite set *S* of *s* symbols, such that every *t*-way permutation of symbols from *S* occurs in at

least one row and each row is a permutation of the *s* symbols. The *t* symbols in the permutation are not required to be adjacent. That is, for every *t*-way arrangement of symbols x_1, x_2, ..., x_t, the regular expression $.*x_1.*x_2.*x_t.*$ matches at least one row in the array. Sequence covering arrays, as the name implies, are analogous to standard covering arrays, which include at least one of every *t*-way combination of any *n* variables, where $t<n$. A variety of algorithms are available for constructing covering arrays, but these are not usable for generating *t*-way sequences because they are designed to cover combinations in any order.

Example 1. Consider the problem of testing four events, *a*, *b*, *c*, and *d*. For convenience, a t-way permutation of symbols is referred to as a *t-way sequence*. There are 4! = 24 possible permutations of these four events, but we can test all 3-way sequences of these events with only six tests (see Table 2).

Test				
1	*a*	*d*	*b*	*c*
2	*b*	*a*	*c*	*d*
3	*b*	*d*	*c*	*a*
4	*c*	*a*	*b*	*d*
5	*c*	*d*	*b*	*a*
6	*d*	*a*	*c*	*b*

Table 2. Tests for four events.

Example 2. A 2-way sequence covering array can be constructed by listing the events in some order for one test and in reverse order for the second test:

1	*a*	*b*	*c*	*d*
2	*d*	*c*	*b*	*a*

To see that the procedure in Example 2 generates tests that cover all 2-way sequences, note that for 2-way sequence coverage, every pair of variables *x* and *y*, *x..y* and *y..x* must both be in some test (where *a..b* means that *a* is eventually followed by *b*). All variables are included in each test, therefore any sequence *x..y* must be in either test 1 or test 2 and its reverse *y..x* in the other test.

2 Constructing Sequence Covering Arrays

For *t-way sequence* test generation, where $t > 2$, we use a greedy algorithm that generates a large number of tests, scores each by the number of previously uncovered sequences it covers, then chooses the highest scoring test. This simple approach produces surprisingly good results, and we use an additional heuristic to improve its efficiency. After each choice from candidate tests, the sequence just selected is reversed and output as the next test. The basis for this heuristic is that the test selected from a candidate pool covered the largest number of uncovered sequences, and we want to produce a new test with as many sequences as possible that do not duplicate previous ones. Creating the second test as the reversal of the first ensures that $test_2$ will have just as many new sequences covered as $test_1$, as shown below.

Algorithm *t-seq*(int *t*, int *n*)
// *t* = interaction strength; *n* = # parameters, $n > t$;
N = # candidate tests to generate
initialize test set *ts* to be an empty set;
initialize set *chk* of $n \times (n-1) \times ... \times (n-t+1)$ bits to 0;
while (all *t-way sequence*s not marked in *chk*) {
1. *tc* := set of *N* test candidates generated with random values 0..*n*-1
2. $test_1$:= test from set *tc* that covers the greatest number of sequences not marked as covered in *chk*;
3. **for each** new sequence covered in $test_1$, mark corresponding bit in set *chk* to 1;
4. *ts* := *ts* ∪ $test_1$;
5. **if** (all *t-way sequence*s not marked in *chk*) {
   $test_2$:= reverse($test_1$);
   ts := *ts* ∪ $test_2$;
   for each new sequence cover in $test_2$, mark corresponding bit in set *chk* to 1;
   }

}
return *ts*;

Figure 1. Algorithm *t-seq*

Proof of reversal step. Each $test_2$, produced at step 5 by reversing $test_1$, will cover the same number of previously uncovered sequences as $test_1$.

It will be shown that for any sequence covered prior to test1, its reverse sequence was also covered before generation of $test_1$, and for any sequence newly covered by $test_1$, its reverse has not been covered prior to generation of $test_2$. For each loop of the algorithm, two tests are produced, and any sequence in $test_1$ is accompanied by its reverse in $test_2$. The sequences covered in $test_1$ can be divided into sets *C*, sequences covered before test1 was generated, and *U*, new sequences that were not covered before $test_1$ was generated. For any sequence *s* in *C*, its reverse s^{-1} will be generated in $test_2$. Because *s* had been generated previously, its reverse also was generated by step 5 at an earlier point. For a sequence *s* in *U*, its reverse must also not have been covered prior to this point, because if s^{-1} had been generated previously, then the algorithm ensures that *s* must also have been generated, which would be a contradiction. (end of proof)

Table 3 shows the number of 3-way sequence and 4-way sequence tests generated using this algorithm. Note that the algorithm produces an even number of tests for all except n=5 for 4-way sequences, a consequence of step 5.

Events	3-seq Tests	4-seq Tests
5	8	29
6	10	38
7	12	50
8	12	56
9	14	68
10	14	72
11	14	78
12	16	86
13	16	92
14	16	100
15	18	108
16	18	112
17	20	118
18	20	122
19	22	128
20	22	134
21	22	134
22	22	140
23	24	146
24	24	146
25	24	152
26	24	158
27	26	160
28	26	162
29	26	166
30	26	166
40	32	198
50	34	214
60	38	238
70	40	250
80	42	264

Table 3. Number of tests for combinatorial 3-way and 4-way sequences.

3 Algorithm Analysis

The algorithm is dominated by the selection of a candidate test that covers the greatest number of previously uncovered sequences. An array of bits for each possible *t*-way sequence is used so that marking and testing the array for a particular sequence can be done in constant time for each of the t-way sequences This selection process checks each of the $n \times (n-1) \times ... \times (n-t+1)$ possible *t*-way sequences to determine if the sequence has previously been covered or is newly covered by the candidate test. The check is done for each of the *N* candidate tests, with constant *N*, so the time complexity of the algorithm is $O(n^t)$. Storage required for the algorithm is $O(n^t)$ also, because of the set *chk* for keeping track of which sequences have been covered at each step.

The number of tests generated grows logarithmically with *n*. We show that at each step, a greedy algorithm that selects the test which covers the largest number of previously uncovered sequences will progress at a rate of at least 1/*t*! of the remaining sequences at each iteration. Thus uncovered sequences are reduced as $U_{i+1} = U_i(1 - 1/t!)$, and after *k* iterations, remaining uncovered sequences will be $U_0(1 - 1/t!)^k$. Initially, $U_0 = n \times (n-1) \times ... \times (n-t+1)$. For small *n*, it may be possible to implement an optimal greedy algorithm that tests all *n*! possible tests. For larger values of *n*, the algorithm may be reasonably close to finding an optimal next test, with sufficient candidates.

Define a *sequence* as above, a *t*-way arrangement of *t* symbols $x_1, x_2, ..., x_t$, possibly embedded within a longer arrangement of symbols

such that the regular expression $.*x_1.*x_2.*x_t.*$ will match. A *test sequence* will be defined as a particular sequence within a particular possible test. Thus for n symbols, there are $n!$ possible tests, $n!\binom{n}{t}$ test sequences, $n(n-1)...(n-t+1) = \binom{n}{t}t!$ t-way sequences to be covered, and $\frac{n!}{t!}$ tests per sequence, i.e., each sequence occurs in $\frac{n!}{t!}$ tests.

Proof of coverage rate. At the start of iteration i, there must be at least one test that covers $\frac{1}{t!}U_i$ previously uncovered sequences.

At the start of generating test i, we have U_i uncovered sequences and $n!–i$ possible tests that have not been added to *ts*. Initially, we have $U_0 = \binom{n}{t}t!$ uncovered sequences. Any test selected for test 0 will cover $\binom{n}{t} = \frac{1}{t!}U_0$ sequences, so $U_1 = U_0(1 - 1/t!)^1$ prior to generation of test 1. For U_i remaining uncovered sequences, there are $T_u = U_i\frac{n!}{t!}$ test sequences occurring in $n!–i$ remaining possible tests, so there must be at least one test with $T_u/(n!-i)$ or more uncovered sequences: $\frac{U_i\frac{n!}{t!}}{n!–i} = \frac{n!U_i}{(n!–i)t!} \geq \frac{1}{t!}U_i$. Thus there is at least one test that will cover $1/t!$ of the remaining sequences. (end of proof)

Lower bounds on number of tests. If K(n, t) denotes the smallest number of tests in a t-way sequence covering array for n events. Clearly K(n, t) $\geq t!$, since each test covers $\binom{n}{t}$ arrangements and we need to cover a total of $\binom{n}{t}t!$. A lower bound can also be identified in relation to the size of a conventional covering array. It is shown below that K(n, 3) $\geq$ CAN(n-1, 2) for conventional covering arrays CAN(n, t). Since CAN(n-1,2) is a lower bound, the size of the sequence covering array has to grow logarithmically in n, so by this measure the algorithm performs well. For larger t, K(n, t) > K(n, 3), so this provides a lower bound for other values of t also.

Proof of lower bound relation. Suppose rows p_1, …, p_k form a 3-way sequence covering array for n events, of size K = K(n, 3). For each i = 1..k, form a 0-1 vector v_i of length n-1 by letting $v_i[j]$ = 1 if j occurs to the left of n in p_i, and 0 otherwise. Given any two j_1 and j_2, the three numbers j_1, j_2, and n must occur in all possible orders in the rows p_1, …, p_k, and clearly all four possibilities for the entries in the j_1-th and j_2-th positions must occur among the corresponding vectors v_i. Therefore v_1, …, v_k form a binary pairwise covering array. (end of proof)

Example. We will construct a binary pairwise covering array from the sequence array in Table 2, letting *a*, *b*, *c*, *d*, be represented by 1, 2, 3, 4 respectively. For the first row, 1, 4, 2, 3, note that 1 occurs before 4, so $v_1[1]$ = 1, while 2 and 3 occur after 4, so $v_1[2]$ = 0 and $v_1[3]$ = 0. Then, $v_2[2]$ = 1, $v_2[1]$ = 1, and $v_2[3]$ = 1, since 2, 1, 3 all occur to the left of 4 in test 2, and so on.

4 Using Sequence Covering Arrays

Sequence covering arrays have been incorporated into operational testing for a mission-critical system that uses multiple devices with inputs and outputs to a laptop computer. For this system, earlier attempts at covering event sequences was accomplished through the use of Latin-Square designs. A Latin-Square design (developed for testing the effect of different treatments to different plots in agriculture) has as many test-cases as treatments and each treatment appears exactly once in every row and column. Latin-Squares were proposed not for their effective sequence-coverage, but for their convenience in designing test cases where each event can only appear once per-row and each event occurs at every possible step-location.

Test				
1	*a*	*b*	*c*	*d*
2	*d*	*a*	*b*	*c*
3	*c*	*d*	*a*	*b*
4	*b*	*c*	*d*	*a*

Table 4. Latin-Square for four events.

A by-product of Latin-Squares (LS) is 2-way coverage of all pairs, but in the example in Table 4, the array only achieves 50% 3-way coverage in four cases. Compare the sequences covered per test-case (SPTC) to sequence covering array performance for a 4-event test (with 12 unique 2-way sequences and 24 unique 3-way sequences), as shown in Table 5. In short, a sequence covering approach dominated previous methods in effectiveness and efficiency.

Design	Strength	Cases	Coverage	SPTC
SCA	2	2	100%	6
	3	6	100%	4
LS	2	4	100%	3
	3	4	50%	3

Table 5. Comparison of SCA and LS designs.

As noted in Section 1, it was the case with this system that system functionality depended on the order in which events occurred, though it did not matter whether events occurred adjacent to one another (in any sub-sequence), nor did it matter which step an event fell under (without regard to the other events). The test procedure for this system has 8 steps: boot system, open application, run scan, connect peripherals P-1 through P-5. It is anticipated that because of dependencies between peripherals, the system may not function properly for some sequences. That is, correct operation requires cooperation among multiple peripherals, but experience has shown that some may fail if their partner devices were not present during startup. Thus the order of connecting peripherals is a critical aspect of testing. In addition, there are constraints on the sequence of events: can't scan until the app is open; can't open app until system is booted. There are 40,320 permutations of 8 steps, but some are redundant (e.g., changing the order of peripherals connected before boot), and some are invalid (violates a constraint). Around 7,000 are valid, and non-redundant, but this is far too many to test for a system that requires manual, physical connections of devices.

Test	Events						
1	0	1	2	3	4	5	6
2	6	5	4	3	2	1	0
3	2	1	0	6	5	4	3
4	3	4	5	6	0	1	2
5	4	1	6	0	3	2	5
6	5	2	3	0	6	1	4
7	0	6	4	5	2	1	3
8	3	1	2	5	4	6	0
9	6	2	5	0	3	4	1
10	1	4	3	0	5	2	6
11	2	0	3	4	6	1	5
12	5	1	6	4	3	0	2

Table 6. Seven-event test set.

The system was tested using a seven-step sequence covering array, removing boot-up from test sequence generation. The initial test configuration (Table 6) was generated using the algorithm given in Sect. 2. Some changes were made to the pre-computed sequences based on unique requirements of the system test. If 6='Open App' and 5='Run Scan', then cases 1, 4, 6, 8, 10, and 12 are invalid, because the scan cannot be run before the application is started. This was handled by swapping items when they are adjacent (1 and 4), and out of order. For the other cases, several were generated from each that were valid mutations of the invalid case. A test was also embedded to see whether it mattered where each of three USB connections were placed. The last test case ensures at least strength 2 (sequence of length 2) for all peripheral connections and 'Boot', i.e., that each peripheral connection occurs prior to boot. The final test array is shown in Table 7.

5 Related Work

To our knowledge, the notion of sequence covering arrays has not been discussed in the computer science or mathematics literature. Event sequence testing has a long history [2, 3, 6, 7, 10, 11], but existing work in this area has focused on coverage or program control flow graphs or

sequences derived from state transitions [3, 9, 10], syntax expressions [2, 6], or other formal descriptions of program behavior.

Of previous investigations, the most closely related to ours includes applications of covering arrays and combinatorial testing to graphical user interface (GUI) testing. Wang et al. [12, 13] describe a method of testing all 2-way sequences in the navigation graph of a web application. The method presented in this paper covers general *t*-way, rather than strictly 2-way pairwise testing, but does not deal with issues of navigation graph coverage.

Yuan, Cohen, and Memon [14] use covering arrays to improve the efficiency of GUI testing where each node in the sequence can contain one of a set of events, such as *Clear Canvas*, *Draw Circle*, or *Refresh*. That is, each test contains several steps, but events may be repeated, for example *Clear Canvas – Refresh – Refresh – Draw Circle*. Covering arrays are employed to ensure coverage of all *t*-way sequences of events in sequence, with nodes in the sequence represented as factors and possible events at each node represented as levels of the covering array. In our applications, events may not be repeated, so covering arrays do not apply in the same manner.

Apilli, Richardson, and Alexander [1] introduce a fault-based testing method that uses t-way combinations of potential faults in web services, making it possible to detect interaction faults. As in Yuan et al., combinatorial methods are applied to the configuration of inputs, rather than their sequence.

Conclusions

Sequence covering arrays can have significant practical value in testing. Because the number of tests required grows only logarithmically with the number of events, t-way sequence coverage is tractable for a wide range of testing problems. Using a sequence covering array for system testing described here made it possible to provide greater confidence that the system would function correctly regardless of possible dependencies among peripherals. Because of extensive human involvement, the time required for a single test is significant, and a small number of random tests or scenario-based ad hoc testing would be unlikely to provide *t*-way sequence coverage to a satisfactory degree.

References

1. Apilli, B. S., L. Richardson, C. Alexander, Fault-based combinatorial testing of web services. In *Proc. 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, October 25 - 29, 2009).
2. G. V. Bochmann, A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," *Softw.Eng. Notes*, ACM SIGSOFT, pp. 109-124, 1994.
3. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178-187, 1978.
4. Dalal, S.R., C.L. Mallows, Factor-covering Designs for Testing Software, Technometrics, v. 40, 1998, pp. 234-243.
5. M. Grindal, J. Offutt, S.F. Andler, Combination Testing Strategies: a Survey, Software Testing, Verification, and Reliability, v. 15, 2005, pp. 167-199.
6. K.V. Hanford, "Automatic Generation of Test Cases", IBM Systems Journal, vol. 9, no. 4, pp. 242-257, 1970.
7. W. E. Howden, G. M. Shi: Linear and Structural Event Sequence Analysis. *ISSTA 1996*: pp. 98-106, 1996.
8. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. IEEE Trans. Softw. Eng., 31(10):884–896, 2005.
9. J. Offutt, L. Shaoying, A. Abdurazik, and P. Ammann, "Generating Test Data From State-Based Specifications," *J. Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25-53, March, 2003.
10. D.L. Parnas, "On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System," *Proc. 24th ACM Nat'l Conf.*, pp. 379-385, 1969.
11. B. Sarikaya, "Conformance Testing: Architectures and Test Sequences,"

Computer Networks and ISDN Systems, vol.17, no. 2, North-Holland, pp. 111-126, 1989.

12. W. Wang Sampath, S. Yu Lei, Kacker, R., An Interaction-Based Test Sequence Generation Approach for Testing Web Applications, High Assurance Systems Engineering Symposium, 2008. HASE 2008. Nanjing, 3-5 Dec. 2008 pp. 209-218.
13. W. Wang, Y. Lei, S. Sampath, R. Kacker, D. Kuhn, J. Lawrence, "A Combinatorial Approach to Building Navigation Graphs for Dynamic Web Applications", Proceedings of 25th IEEE International Conference on Software Maintenance, pp. 211-220, 2009.
14. X. Yuan, M.B. Cohen, A. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", November 2007
15. ASE '07: Proceedings of the 22nd IEEE/ACM Intl. Conf. Automated Software Engineering, pp. 405-408.
16. X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In ICSE'07, Proceedings of the 29th International Conference on Software Engineering, pages 396–405, Minneapolis, MN, USA, May 23–25, 2007.

Original Case	Case	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8
1	1	Boot	P-1 (USB-RIGHT)	P-2 (USB-BACK)	P-3 (USB-LEFT)	P-4	P-5	Application	Scan
2	2	Boot	Application	Scan	P-5	P-4	P-3 (USB-RIGHT)	P-2 (USB-BACK)	P-1 (USB-LEFT)
3	3	Boot	P-3 (USB-RIGHT)	P-2 (USB-LEFT)	P-1 (USB-BACK)	Application	Scan	P-5	P-4
4	4	Boot	P-4	P-5	Application	Scan	P-1 (USB-RIGHT)	P-2 (USB-LEFT)	P-3 (USB-BACK)
5	5	Boot	P-5	P-2 (USB-RIGHT)	Application	P-1 (USB-BACK)	P-4	P-3 (USB-LEFT)	Scan
6A	6	Boot	Application	P-3 (USB-BACK)	P-4	P-1 (USB-LEFT)	Scan	P-2 (USB-RIGHT)	P-5
6B	7	Boot	Application	Scan	P-3 (USB-LEFT)	P-4	P-1 (USB-RIGHT)	P-2 (USB-BACK)	P-5
6C	8	Boot	P-3 (USB-RIGHT)	P-4	P-1 (USB-LEFT)	Application	Scan	P-2 (USB-BACK)	P-5
6D	9	Boot	P-3 (USB-RIGHT)	Application	P-4	Scan	P-1 (USB-BACK)	P-2 (USB-LEFT)	P-5
7	10	Boot	P-1 (USB-RIGHT)	Application	P-5	Scan	P-3 (USB-BACK)	P-2 (USB-LEFT)	P-4
8A	11	Boot	P-4	P-2 (USB-RIGHT)	P-3 (USB-LEFT)	Application	Scan	P-5	P-1 (USB-BACK)
8B	12	Boot	P-4	P-2 (USB-RIGHT)	P-3 (USB-BACK)	P-5	Application	Scan	P-1 (USB-LEFT)
9	13	Boot	Application	P-3 (USB-LEFT)	Scan	P-1 (USB-RIGHT)	P-4	P-5	P-2 (USB-BACK)
10A	14	Boot	P-2 (USB-BACK)	P-5	P-4	P-1 (USB-LEFT)	P-3 (USB-RIGHT)	Application	Scan
10B	15	Boot	P-2 (USB-LEFT)	P-5	P-4	P-1 (USB-BACK)	Application	Scan	P-3 (USB-RIGHT)
11	16	Boot	P-3 (USB-BACK)	P-1 (USB-RIGHT)	P-4	P-5	Application	P-2 (USB-LEFT)	Scan
12A	17	Boot	Application	Scan	P-2 (USB-RIGHT)	P-5	P-4	P-1 (USB-BACK)	P-3 (USB-LEFT)