Efficient Methods for Interoperability Testing Using Event Sequences

D. Richard Kuhn, NIST
James M. Higdon, Eglin AFB
James F. Lawrence, NIST
Raghu N. Kacker, NIST
Yu Lei, University of Texas at Arlington

Abstract. Many software testing problems involve sequences of events. The methods described in this paper were motivated by testing needs of mission critical systems that may accept multiple communication or sensor inputs and generate output to several communication links and other interfaces, where it is important to test the order in which events occur. Using combinatorial methods makes it possible to test sequences of events using significantly fewer tests than previous procedures.

Introduction

For many types of software, the sequence of events is an important consideration [1, 2]. For example, graphical user interfaces may present the user with a large number of options that include both order-independent (e.g., choosing items) and order-dependent selections (such as final selection of items, quantity, and payment information). The software should work correctly, or issue an appropriate error message, regardless of the order of events selected by the user. A number of test approaches have been devised for these problems, including graph-covering, syntax-based, and finite-state machine methods [3, 4, 5].

In testing such 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, or only if devices B and C were both already connected. The methods described in this paper were developed to address testing problems of this nature, using combinatorial methods to provide efficient testing. Sequence covering arrays, as defined here, ensure that every t events from a set of n ($n > t$) will be tested in every possible t-way order, possibly with interleaving events among each subset of t 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 [6]. The t symbols in the permutation are not required to be adjacent. That is, for every t-way arrangement of symbols $x1$, $x2$, ..., xt, the regular expression *.*x1.*x2.*xt.** matches at least one row in the array.

Example 1

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 operation. 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.

Table 1. Example system events.

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

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: [***bde***], [***bed***], [***dbe***], [***deb***], [***ebd***], [***edb***]. A test that covers the permutation [***dbe***] is: [***adcfbe***]; another is [***adcbef***]. With only 10 tests, we can test all 3-event sequences, shown in Table 2. In other words, any sequence of three events taken from ***a..f*** arranged in any order can be found in at least one test in Table 2 (possibly with interleaved events).

Table 2. All 3-event sequences of six events.

Test	Sequence
1	*a b c d e f*
2	*f e d c b a*
3	*d e f a b c*
4	*c b a f e d*
5	*b f a d c e*
6	*e c d a f b*
7	*a e f c b d*
8	*d b c f e a*
9	*c e a d b f*
10	*f b d a e c*

Returning to the example set of events {*b, d, e*}, with six permutations: [*bde*] is in Test 5, [*bed*] is in Test 4, [*dbe*] is in Test 8, [*deb*] is in Test 3, [*ebd*] is in Test 7, and [*edb*] is in Test 2.

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 3-way sequences is a dramatic improvement, as is 72 tests for all 4-way sequences (see Table 4).

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, as shown in Table 3:

Table 3. 2-way sequence covering array.

Test	Sequence
1	a b c d
2	d c b a

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

Events	3-seq Tests	4-seq Tests
5	8	26
6	10	36
7	12	46
8	12	50
9	14	58
10	14	66
11	14	70
12	16	78
13	16	86
14	16	90
15	18	96
16	18	100
17	20	108
18	20	112
19	22	114
20	22	120
21	22	126
22	22	128
23	24	134
24	24	136
25	24	140
26	24	142
27	26	148
28	26	150
29	26	154
30	26	156
40	32	182
50	34	204
60	38	222
70	40	238
80	42	250

Generating Sequence Covering Arrays

Sequence covering arrays, as the name implies, are analogous to standard covering arrays [7], which include at least one of every *t*-way combination of any *n* variables, where $t < n$. We have developed several methods of generating SCAs, but the most efficient approach is a simple greedy algorithm that iteratively generates multiple candidate tests, then selects the one that covers the largest number of previously uncovered sequences, repeating until all sequences have been covered. This algorithm produces more compact arrays than others developed so far.

Table 4 shows the number of 3-way and 4-way sequence tests for event sets of varying sizes generated using the algorithm. In another paper [6], we have shown the number of tests generated is proportional to log *n*, for *n* events, making it practical to test complex systems with a large number of events using a reasonable number of tests. Logarithmic growth in number of tests can also be seen in Table 4.

Using Sequence Covering Arrays

The motivation for this work was a USAF mission-critical system that uses multiple devices with inputs and outputs to a laptop computer. (Confidentiality rules do not permit a detailed description of this system.) System functionality depends on the order in which events occur, though it does not matter whether events are adjacent to one another (in any sub-sequence), nor which step an event falls under, without regard to the other events. The test procedure for this system has eight steps: boot system, open application, run scan, and 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 critical. In addition, there are constraints on the sequence of events: cannot scan until the app is open; cannot open app until the system is booted. There are 40,320 permutations of eight 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.

The system was tested using a seven-step sequence covering array, removing boot-up from test sequence generation. The initial test configuration for 3-way sequences was generated using the algorithm given in Sect. 2. Covering all 3-way sequences allowed testing a much larger set of states than using 2-way sequences, but could be accomplished at a reasonable cost. 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 permutations of the invalid case. A test was also embedded to see whether it mattered where each of three USB connec-

tions 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 5. Errors detected in testing included several that could not be attributed to 2-way sub-sequences. These errors would not have been detected using a simple 2-way sequence covering array (which could consist of only two tests, as in Example 2), and may not have been caught with more conventional tests.

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.

Acknowledgments:

We are very grateful to Tim Grance for support of this work within the NIST Cybersecurity program, and to Paul E. Black for suggestions that helped clarify and strengthen the paper.

Disclaimer:

We identify certain software products in this document, but such identification does not imply recommendation by NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

Table 5. Final test array.

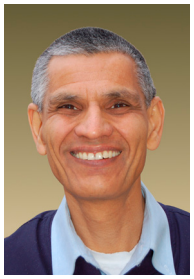
Original Case	Case	Step1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
1	1	Boot	P-1 (USB-RIGHT)	P-2 (USB-BACK)	P-3 (USB-LEFT)	P-4	P-5	Application	Scan
2	2	Boot	Application	Scan	P-5	P-4	P-3 (USB-RIGHT)	P-2 (USB-BACK)	P-1 (USB-LEFT)
3	3	Boot	P-3 (USB-RIGHT)	P-2 (USB-LEFT)	P-1 (USB-BACK)	Application	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)
12B	18	Boot	P 2 (USB RIGHT)	Appli ti	Sc	P 5	P 4	P 1 (USB LEFT)	P 3 (USB BACK)

ABOUT THE AUTHORS

Richard Kuhn is a computer scientist in the Computer Security Division of NIST. His current interests are in information security, empirical studies of software failure, and software assurance, focusing on combinatorial testing. He received an MS in computer science from the University of Maryland College Park.

James Higdon is a senior analyst in Technical Engineering and Acquisition Support with Jacobs Engineering, at the 46th Test Squadron, Eglin Air Force Base, Florida. His current interests are in experimental design and combinatorial testing of hardware/software systems. He received an MS from the Air Force Institute of Technology.

James Lawrence is a Professor in the Department of Mathematics at George Mason University, Fairfax, VA, and a faculty associate at NIST. His current interests are in convexity and combinatorics, including applications in software testing. He received a Ph.D. from the University of Washington.



Raghu Kacker is a researcher in the Applied and Computational Mathematics Division of NIST. His current interests include software testing and evaluation of the uncertainty in outputs of computational models and physical measurements. He has a Ph.D. in statistics and is a Fellow of the American Statistical Association, and American Society for Quality.

Yu Lei is an Associate Professor in Department of Computer Science and Engineering at the University of Texas, Arlington. His current research interests include automated software analysis and testing, with a special focus on combinatorial testing, concurrency testing, and security testing. He received his Ph.D. from North Carolina State University.

REFERENCES

1. 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.
2. W. E. Howden, G. M. Shi: Linear and Structural Event Sequence Analysis. ISSTA 1996: pp. 98-106, 1996.
3. S. Chow, "Testing Software Design Modeled by Finite-State Machines," IEEE Trans. Softw. Eng., vol. 4, no. 3, pp. 178187, 1978.
4. 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.
5. B. Sarikaya, "Conformance Testing: Architectures and Test Sequences," Computer Networks and ISDN Systems, vol.17, no. 2, North-Holland, pp. 111-126, 1989.
6. D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R.N. Kacker, Y. Lei, "Combinatorial Methods for Event Sequence Testing", 8 Oct 2010 (submitted for publication). <http://csrc.nist.gov/groups/SNS/acts/documents/event-seq101008.pdf>
7. X. Yuan, M.B. Cohen, A. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", November 2007 ASE '07: Proc. 22nd IEEE/ACM Intl. Conf. Automated Software Engineering, pp. 405-408.