Ordered t-way Combinations for Testing State-based Systems

D. Richard Kuhn
M S Raunak
Computer Security Division
Information Technology Laboratory

Raghu N. Kacker
Applied and Computational Mathematics Division
Information Technology Laboratory

June 13, 2022

This publication is available free of charge from:
https://doi.org/10.6028/NIST.CSWP.26

National Institute of Standards and Technology
U.S. Department of Commerce

Abstract

Fault detection often depends on the specific order of inputs that establish states which eventually lead to a failure. However, beyond basic structural coverage metrics, it is often difficult to determine if code has been exercised sufficiently to ensure confidence in its functions. Measures are needed to ensure that relevant combinations of input values have been tested with adequate diversity of ordering to ensure correct operation. Combinatorial testing and combinatorial coverage measures have been applied to many types of applications but have some deficiencies for verifying and testing state-based systems where the response depends on both input values and the current system state. In such systems, internal states change as input values are processed. Examples include network protocols, which may be in listening, partial connection, full connection, disconnected, and many other states depending on the values of packet fields and the order of packets received. Similarly, merchant account balances in credit card systems change continuously as transactions are processed. This publication introduces a notion of ordered t-way combinations, proves a result regarding the construction of adequate blocks of test inputs, and discusses the application of the results to verify and test state-based systems.

Keywords

combinatorial coverage; combinatorial methods; combinatorial testing; software testing; structural coverage; test coverage.

Disclaimer

Additional Information

For additional information on NIST's Cybersecurity programs, projects and publications, visit the Computer Security Resource Center. Information on other efforts at NIST and in the Information Technology Laboratory (ITL) is also available.

Submit comments on this publication to: cfdwp@nist.gov

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

1 Introduction

Vulnerability and fault detection often depend on the specific order of inputs that establish states which eventually lead to a failure. That is, many software processes are not deterministic functions where an input produces the same output whenever the process is invoked irrespective of previous invocations. This is particularly true of real-time systems, which are designed to run continuously, maintain states, and respond to a changing series of inputs. Such systems are typically driven by a loop function that accepts input values, processes and responds to those inputs, and updates its current state. Examples include network protocols and data servers. The system state may change, depending on input, and the system may subsequently respond differently to the same input. That is, the response of the process to a particular set of input values may depend on its current state, such as whether a communication protocol is in a listen or connection-open state. The current state depends on the order of input values that were contained in previously received packets. The same sort of state-dependent behavior occurs in many other types of systems as well.

Ensuring that inputs and system states in testing are sufficiently representative of what will be encountered in practice is critical to any form of effective software testing. The common practice for evaluating how thoroughly software has been tested is by using some sort of structural coverage metrics, such as statement coverage, branch coverage, or MC/DC coverage [13] . Test cases selected using only structural coverage criteria are often not very effective as they are not designed to include corner cases with specific combinations of input values that may cause a failure. Looking beyond these commonly used metrics, it is often difficult to determine if the code has been adequately tested and even more difficult to ensure that a sufficient diversity of inputs has been achieved. This is particularly true of assertion-based testing or runtime verification, where program states and properties are monitored to verify correct processing. For runtime assertions to discover bugs, the software needs to be exercised with a set of values in a particular order that leads to the failure. Consequently, for strong software assurance, measures are needed to verify that *combinations of input values* and *combinations of input orders* in a test suite are sufficient.

Combinatorial coverage measures provide an effective method for quantifying the thoroughness of test input values [26] . A number of measures have been defined for the coverage of (static) input value combinations. For example, with four binary variables, there are a total of $2^2 * C(4,2) = 24$ possible settings of the four variables taken two at a time. If a test set includes tests that cover 19 of the 24, the simple combinatorial coverage is 19/24 = 0.79. These measures quantify the degree to which input values cover the potential space of parameter value combinations without regard for the *order* in which these inputs occur in a test set or in normal operations. However, if a system state is affected or determined by the order of inputs, even thorough coverage of the input space may not detect some failure conditions. Thus, it is desirable to supplement measures of input space coverage with measures of the input value combination ordering.

2 Related Work

Combinatorial aspects of input ordering have been studied in the context of *event sequences*. Sequence covering arrays (SCAs) were introduced in [1] and [2] and further developed in [3] , [4] , [5] , [7] , [8] [9] , [10] , and [11] .

Definition. A *sequence covering array* [2] , SCA(*N, S, t*) is 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. The *t* symbols in the permutation are not required to be adjacent.

For example, Fig. 1 shows an *event* sequence $a *\rightarrow b *\rightarrow c$ in test 1 and an event sequence of $d* \rightarrow c* \rightarrow a$ in test 3, where $x *\rightarrow y$ denotes *x* is eventually followed by *y*, with possible interleaving. Note that the event sequence array has *sequences of events* in each *row*. Event sequences are made up of a value in a column followed by values in columns to the right.

Test	*p0*	*p1*	*p2*	*p3*
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*

Fig. 1. Event sequence

Combinatorial testing with constraints on the order in which values and combinations are applied in tests was analyzed extensively in [5] . Extended covering arrays that consider the sequence of values in each test were defined in [6] . Combination sequences were studied in [20] , combining configuration and the order of combinations while also considering constraints. Another structure defined as a sequence covering array of t-way combinations has been termed a multi-valued sequence covering array, introduced in [25] .

3 Ordered Combination Covering

A combination order is different from an event sequence. As noted in the previous section, an event sequence is a possibly interleaved sequence of symbols in a single row of a test array. A combination order, as defined below, is across multiple rows, given in the order in which tests will be executed. A t-way permutation of symbols is referred to as a *t-way order*, which will be called a t-order for brevity. The t events in the order may be interleaved with others (i.e., the order $a \rightarrow b \rightarrow c$ covers three 2-event orders: a followed by b and b followed by c, and a followed by c). Denoting event a eventually followed by event b, possibly with other events interleaved, is written as $a^* \rightarrow b$.

Consider the notion of an s-order of t-way combinations of the input parameters as a series of rows of test data that contain a particular set of t-way combinations in a specified order, with possibly interleaved rows.

Definition 1. The notation $a^* \rightarrow b$ denotes the presence of combination a eventually followed by combination b, possibly with other rows interleaved. A combination order c_1 $^* \rightarrow$ c_2 $^* \rightarrow$ $\ldots ^* \rightarrow c_s$ of s sequence of t-way combinations, abbreviated s-order, is a set of t-way combinations in s rows. Each c_i is a t-way combination of parameter values.

Example. Fig. 2 shows *combination* order $p_0p_1 = ad$ $^* \rightarrow p_0p_3 = ba$ $^* \rightarrow p_1p_3 = ab$, which is a 3-order of 2-way combinations. Thus, the term *ordered combinations* refers to combinations in a row followed by combinations in rows below.

Test	p0	p1	p2	p3
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***

Fig. 2. Ordered combinations

When all s-orders of t-way combinations of the input parameters have been covered, it is referred to as an *ordered combination cover* (OCC). For the OCCs, the combination orders are treated across rows (i.e., a combination in a row followed by combinations in rows below). A t-way combination occurs in some row and is eventually followed by other t-way combinations in other rows. For three Boolean parameters a, b, c in Fig. 3, ab=00 is followed by ab=10 ab=00 ac=11 ac=01 bc=01 (ac=01 and bc=01 are also followed by this group).

a	*b*	*c*
0	0	1
1	0	1
0	0	1

Fig 3. Ordered combinations of parameter values

Combinations of parameter values such as this can be significant in protocol verification and testing, such as with combinations of values in each packet affecting the state where later packets produce different responses depending on the state that resulted from previous packet orders.

Definition 2. An *ordered combination cover*, OCC(N,s,t,p,v), covers all s-orders of t-way combinations of the v values of p parameters, where t is the number of parameters in combinations and s is the number of combinations in an ordered series. Permutations of parameter value combinations may appear multiple times in a combination order. For example, a particular 2-order of 2-way combinations may be $(p_1p_2 = 01)\ ^* \rightarrow (p_2p_4 = 11)$.

The utility of combination order covering can be illustrated with an example. Consider the covering array in Fig. 4, which includes all 2-way combinations of four Boolean variables.

Test	p_1	p_2	p_3	p_4
1	0	0	1	0
2	0	1	0	1
3	1	0	0	1
4	1	1	1	0
5	0	1	0	0
6	0	0	1	1

Fig. 4. 2-way covering array of Boolean variables.

Suppose these six tests are applied to the system modeled with a finite state machine diagram in Fig. 5, and tests are run in the order 1..6. If condition A = "$p_1 \wedge p_2$" and condition B = "$p_1 \wedge \sim p_2$", then the error in state 2 will not be discovered. The system returns to state 0 for tests 1 through 3, then enters state 1 with test 4 and moves to state 3 with test 5. Because the test array does not include the ordered combinations $p_1, p_2 = 11 \ast\!\!\rightarrow p_1, p_2 = 10$, the error is not exposed. However, if the tests are run in the order [1, 2, 4, 3, 5, 6], then the error in state 2 will be discovered because the third test (row 4 in Fig. 4) leads to state 1, the fourth test (row 3) causes condition B to evaluate to true, and the system enters state 2, exposing the error.

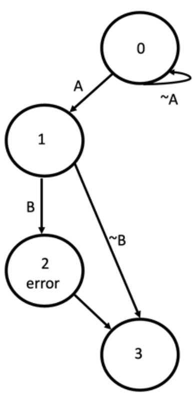

Fig. 5. Example Finite State Machine.

Note that the definition includes the repetition of a value combination in the OCC. The possibility of repeated occurrences of a value combination in an order is allowed based on the assumption that a particular combination may occur in multiple tests, and this sequence may be relevant to the system or software under test (SUT). For example, a function call to create a new file '`f1`' followed by a duplicate call to create '`f1`' may trigger some behavior other than an expected error message. So the 3-way combination `(create, f1, 256)` may be desirable to include more than once in a series of test inputs.

Returning to the example in Fig. 4 and Fig. 5, suppose that a file system is being tested, where p_1 is *function* with values {0 = read, 1 = write}, and p_2 is *rewind*, which indicates if the pointer to the starting block is reset to 0 {0 = start from last position, 1 = start from block 0}. A read or write test processes from the starting block indicated by p_2 and continues to the end of the file. So tests 1 to 4 represent 'read from last read position,' 'read from start,' 'write from last write position,' and 'write from start.' State 1 is entered when the file is filled by writing to the end after rewinding to start. The failure represented by state 2 is only exposed when a write is attempted on a file starting from the end. Then, as noted previously, running tests in the order 1..6 will not detect the error. However, when test 4 is run before test 3, the error will be detected because a write is attempted from the last position (end of file), as indicated by the value of p_2.

When tests are executed in sequence with each individual *t-way* combination considered an event, an order of *t*-way combinations containing *s* combinations input in sequence with possible interleaving is an *s*-sequence of *t*-way combinations. For example, a 2-sequence of 3-way combinations could be

$$abd = 001 \ast\!\!\rightarrow bcd = 100,$$

and a 3-sequence of 2-way combinations could be

$$bc = 01 \ast\!\!\rightarrow ad = 11 \ast\!\!\rightarrow bc = 10.$$

An OCC covers all *s*-sequences of *t*-way combinations of the *v* values of the *p* parameters. Because a *t*-tuple is included *s* times in an *s*-sequence, and the number of *t*-way combinations of *p* parameters is $C(p,t)$, for v^t settings of each combination, the total number of combination sequence tuples to be covered is

$$(v^t C(p,t))^s. \qquad (1)$$

The number of combination sequences to be covered grows rapidly with *s* and *t*, so methods for the efficient construction of OCCs are of interest.

Example. Fig. 6 shows a test array that covers all 2-way combinations of values for four parameters, as well as all 2-sequences of 2-way parameter combinations.

	p_1	p_2	p_3	p_4
1	1	1	1	0
2	0	0	0	0
3	1	1	0	1
4	0	0	1	1
5	1	1	0	0
6	1	1	1	1
7	0	1	1	0
8	0	1	1	0
9	0	1	0	1
10	1	1	1	1
11	1	0	1	0
12	1	1	0	1
13	0	1	1	1
14	0	0	0	0
15	0	0	0	1
16	0	0	1	0
17	1	1	1	0
18	0	1	0	1
19	1	1	1	0
20	1	0	1	1

Fig.6. Tests for four parameters, OCC(20,2,4,2)

That is, the test array includes every solution of ($p_w p_x = v_1 v_2$) $*\!\!\rightarrow$ ($p_y p_z = v_1 v_2$), of which there are $(C(4,2)x\ 2^2)^2$= 576 instances. For example, each of the four possible settings of $p_1 p_2$ is followed by each of the four possible settings of $p_3 p_4$ somewhere in the table (distinguished by color). That is, ($p_1 p_2 = 11$) in line 1 is followed by ($p_3 p_4 = 01$), ($p_3 p_4 = 01$), ($p_3 p_4 = 11$), ($p_3 p_4 = 00$), ($p_3 p_4 = 10$) in lines 3, 4, 5, and 7, respectively, highlighted in yellow (also for ($p_1 p_2 = 00$)). Sequences for ($p_1 p_2 = 01$) are shown in green, plus line 14, which provides a ($p_3 p_4 = 00$) for both ($p_1 p_2 = 01$) and ($p_1 p_2 = 10$). Sequences for ($p_1 p_2 = 10$) are highlighted in blue.

3.1 Combination Sequence Covering Arrays in Testing

As seen in Fig. 6, the numbers of combination sequences to be covered will become very large with realistic testing problems as a result of the exponents in expression (1). Consequently, measuring combination sequence coverage can be inefficient and resource intensive. Fortunately, the problem of ensuring combination sequence coverage can be reduced to ensuring coverage of t-way covering arrays, as shown in the following proof. Checking that a test array is a covering array can be done efficiently, making it practical to ensure s-sequences of t-way combinations in large-scale testing.

Theorem: A test set covers s-orders of t-way combinations if and only if it includes an ordered series containing a total of s covering arrays, each of strength t.

Proof: From Definition 2, a sufficient process for generating an s-order t-way OCC is to concatenate s covering arrays of strength t, as shown below in Fig. 7. Because a covering array includes every t-way combination, any order of at least s combinations will occur by taking the rows of s covering arrays from CA_1, CA_2, …, CA_s., where CA_i are t-way covering arrays. Clearly, for any s-order of s combinations, $c_1 *\!\!\rightarrow c_2 *\!\!\rightarrow \ldots *\!\!\rightarrow c_s$, c_1 must be present in CA_1, c_2 in CA_2, etc. because they cover all t-way value combinations by definition, giving the required order.

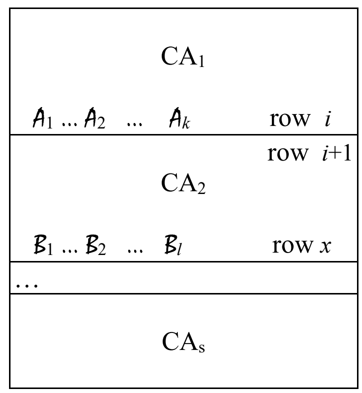

Fig.7. OCC constructed from covering arrays.

To show necessity, consider a series of rows in a test array. There must be at least one combination order that can only exist if the test array can be divided into subarrays, each of which is a covering array. Each row covers some number of t-way combinations. For each row, add the combinations covered to a set, and continue adding non-covered combinations from each successive row. Eventually, a row will be reached that covers the last remaining previously uncovered combinations. Label these previously uncovered combinations $A_1 \ldots A_2 \ \ldots \ A_k$ and the row containing these combinations as row i. $A_1 \ldots A_2 \ \ldots \ A_k$ do not occur in any row prior to row i. The subarray of rows from the first row to and including row i forms a covering array that will be labeled CA_1. With the inclusion of row i, CA_1 includes all t-way combinations, so it is a t-way covering array.

At row i+1, start a new set of combinations covered in rows i+1 and following rows. Continue adding combinations covered in each successive row until a row is reached that covers the last remaining previously uncovered combinations after row i+1. Label these previously uncovered combinations $\mathcal{B}_1$ ... $\mathcal{B}_2$ ... $\mathcal{B}_l$ and the row containing these combinations as row x. $\mathcal{B}_1$ ... $\mathcal{B}_2$ ... $\mathcal{B}_l$ do not occur in any row after row i and prior to row x. The subarray beginning with row i+1 and ending with row x forms a covering array that will be labeled CA_2. CA_2 includes all t-way combinations, with the inclusion of row x, so it is a t-way covering array. Any combination in $\mathcal{A}_1$ ... $\mathcal{A}_2$ ... $\mathcal{A}_k$ must be followed by any t-way combination in some row of CA_2.

Note that any 2-order c_1 *→ c_2 where c_1 is one of $\mathcal{A}_i$ and c_2 is one of $\mathcal{B}_i$ could not have been covered until row x of CA_2 because the $\mathcal{B}_i$ tuples are those that had not been covered in CA_2 before row x (and after row i). Assume that c_1 *→ c_2 is covered before row x in the combined array CA_1 || CA_2. Since c_2 is not in subarray CA_2 before row x, it must be in subarray CA_1. However, c_1 is in the last row of CA_1, so c_2 must be in a row of CA_1 following the last row of CA_1, which is a contradiction.

Therefore c_1 *→ c_2 can be covered only if CA_1 and CA_2 are covering arrays. Continuing in this manner shows that orders of s combinations of strength t can be covered only if the subarrays of the set of test rows form s covering arrays (*end proof*).

Example. Fig. 8 shows the concatenation of two 2-way covering arrays for four binary parameters. Any 2-order of 2-way combinations occurs somewhere in the rows of Fig. 6. For example, (p_1p_2 = 10) (row 3) is followed by p_1p_3 = 00, 01, 10, 11 in rows 5, 6, 9, and 4, respectively. If row 12 is removed, there must be at least one combination order c_1 *→ c_2 where c_2 = (p_3p_4 = 11) that is not covered because (p_3p_4 = 11) is covered only in the last row of CA_1 and CA_2 (row 12). Removing row 12 would result in losing (p_3p_4 = 11) *→ (p_3p_4 = 11). Similarly, (p_1p_4 = 10) is covered only in the third-to-last row (row 10) of CA_1 and CA_2, so there must be at least one combination order c_1 *→ c_2 where c_2= (p_1p_4 = 10) that is not covered if row 10 is removed. Removing row 10 would result in losing (p_1p_3 = 11) *→ (p_1p_4 = 10), (p_3p_4 = 00) *→ (p_1p_4 = 10), and others.

The practical utility of this result is that it shows one can efficiently produce tests that cover all orders of t-way combinations up to any necessary order length by concatenating t-way covering arrays. It also shows that the minimum size of the OCC is determined by the minimum size of the relevant t-way covering arrays. From a testing perspective, producing full coverage of t-way combinations in s length orders makes it possible to detect faults that are only detectable when a system is in a particular state that can only be reached by an order of input combinations.

	p1	p2	p3	p4
1	0	0	1	0
2	0	1	0	1
3	1	0	0	1
4	1	1	1	0
5	0	1	0	0
6	0	0	1	1
7	0	0	1	0
8	0	1	0	1
9	1	0	0	1
10	1	1	1	0
11	0	1	0	0
12	0	0	1	1

Fig. 8. Two concatenated covering arrays.

This result can also be useful for runtime verification, assertion monitoring, and other methods that rely on checking program properties and states as code is executed. If inputs are monitored and recorded, then it is possible to verify whether a covering array series of desired length has been applied in testing. The system should run long enough to enter all major states and allow detection of errors that occur only in particular states. The use of covering arrays gives stronger assurance that relevant states have been reached, as program states depend on the order of inputs, and the coverage of input value combinations can be measured.

3.2 Combination Order Coverage Measurement

A combination order tool for OCCs, Corder, has been developed, allowing for the order coverage of any test set to be measured. It may also be used in generating OCCs using random test generation, measuring coverage, and extending the test array until sufficient coverage is achieved.

In its current form, the tool assumes that all single values of input variables have been included in the input test array and computes *t*-way coverage for *t* = 2..4 in the same manner as the CCM tool for combination coverage [26] . This is referred to in the output report as *static* coverage and measures the coverage of combinations in each row of the array where any t-way covering array will have 100 % coverage of t-way combinations. A second output provides coverage, referred to as *dynamic*, of *s*-orders of *t*-way combinations in the test array.

For example, the test array in Fig. 9 (a) shows 12 tests with four binary parameters or variables. If these are executed in order, the first test includes C(4,2) = 6 events defined as 2-way combinations: *ab* = 00, *ac* = 01, *ad* = 00, *bc* = 01, *bd* = 00, and *cd* = 10. For 2-orders containing ab, there are four possible settings of *ab*. Each of these may followed by value combinations of *ab*, *ac*, *ad*, *bc*, *bd*, and *cd*. Completely covering all 2-way 2-orders (i.e., orders of length 2 of 2-way combinations) would produce 4 * C(4,2) * 4 * C(4,2) orders. One can measure the degree to which these orders are covered and output any missing orders, as shown in Fig. 4(b). Note that *ab* = 11 is followed by *cd* = 01 and *cd* = 10, but *cd* = 00 and *cd* = 11 do not follow *ab* = 11 in the test series, as shown in Fig 5(c), which shows <parameter numbers> : <order> → <parameter numbers> : <order> for 2-way orders.

(a)

a	b	c	d
0	0	1	0
0	1	0	1
1	0	0	1
0	0	1	1
1	1	0	0
1	1	1	0
1	0	0	1
0	1	0	1
0	0	1	0
1	0	0	1
0	1	0	1
0	0	1	0

(b)

a	b	c	d
0	0	1	0
0	1	0	1
1	0	0	1
0	0	1	1
1	1	0	0
1	1	1	0
1	0	0	1
0	1	0	1
0	0	1	0
1	0	0	1
0	1	0	1
0	0	1	0

(c)

```
0,1: ('1', '1') -> 2,3: ('1', '1')
0,1: ('1', '1') -> 2,3: ('0', '0')
0,2: ('1', '1') -> 0,1: ('1', '1')
0,2: ('1', '1') -> 0,2: ('1', '1')
0,2: ('1', '1') -> 0,3: ('1', '0')
0,2: ('1', '1') -> 1,2: ('1', '1')
0,2: ('1', '1') -> 1,3: ('1', '0')
0,2: ('1', '1') -> 2,3: ('1', '1')
0,2: ('1', '1') -> 2,3: ('0', '0')
0,3: ('1', '0') -> 2,3: ('1', '1')
```

Fig. 9 Missing combination orders

Fig. 8 illustrates the output of the Corder tool for the matrix shown in Fig. 4. Basic static coverage measures are shown in the top half of the results to provide an overview of input space coverage. For more detailed data on input space coverage, the CCM tool measuring combination coverage can be used [26] . An example is shown in Fig. 11 for a larger, more realistic array of 10 parameters with 6 values each in a file of 263 tests.

```
file = t9.csv      Nvars:  4   Nrows:  12
_______________________________________________________________
Static - input space coverage
              t|       covered |    max possible |coverage|
              1|             8|                8|  1.0000|
              2|            24|               24|  1.0000|
              3|            22|               32|  0.6875|
              4|             6|               16|  0.3750|
_______________________________________________________________
Dynamic - order coverage
                |       covered  |    max possible |coverage|
 1-way | 2-seq |            64  |             64 |  1.0000|
 1-way | 3-seq |           512  |            512 |  1.0000|
 2-way | 2-seq |           553  |            576 |  0.9601|
 2-way | 3-seq |        11,069  |         13,824 |  0.8007|
```

Fig. 9. Output of Corder tool for matrix shown in Fig. 4

The Corder tool provides the following output:

- *file* = input file name containing the test vectors to be analyzed
- *Nvars* = number of variables; each column of the input .csv file corresponds to a single variable
- *Nrows* = number of rows of input file
- *Static-input space coverage*: coverage statistics for t-way combination coverage of the input file for levels of t specified in first column
- *Dynamic-order coverage*: coverage statistics for orders of combinations as described in this section

```
file = g10.6.263.csv    Nvars:  10   Nrows:  263
___________________________________________________
Static - input space coverage
      t|     covered |    max possible |coverage|
      1|          60|             60|  1.0000|
      2|       1,618|          1,620|  0.9988|
      3|      18,268|         25,920|  0.7048|
      4|      50,044|        272,160|  0.1839|
___________________________________________________
Dynamic - order coverage
        |     covered  |    max possible |coverage|
 1-way | 2-seq |         3,600 |          3,600 |  1.0000|
 1-way | 3-seq |       216,000 |        216,000 |  1.0000|
 2-way | 2-seq |     2,606,616 |      2,617,924 |  0.9957|
 2-way | 3-seq | 4,143,537,228 |  4,235,801,032 |  0.9782|
___________________________________________________
```

Fig. 11. 10-parameter combination order cover measurement

It is important to understand the difference between static and dynamic coverage as defined here. Essentially, static coverage is based on the presence or absence of *t*-way settings of the input variables, and dynamic coverage refers to the coverage of possible orders of these combinations.

Static coverage corresponds to combination coverage measures defined in [26] and other publications. In this case, there are four variables with two values each, so for single-variable coverage (t=1), there are 4x2 = 8 possible settings. The tool assumes that all single values of variables are included in the input file. For 2-way combinations, there are C(4,2) = 6 possible combinations, each of which has 2x2 = 4 possible settings, so the number of combinations to be covered is 24, all of which are covered at least once. Similarly, for 3-way combinations there are C(4,3) = 4 combinations, which may have 2x2x2 = 8 settings for a total of 32 possible settings to be covered. Of the possible settings, there are 22 covered (for example, *abc* = 000 is missing, as is *acd* = 000) for a coverage figure of .6875.

For dynamic coverage, the interaction strength (level of *t*) for the combinations included in orders and the number of combinations in an order need to be specified.

4 Conclusions

This report considers practical methods for testing complex orders of all t-way combinations up to some specified level of t. It is shown that the test set covers *s*-orders of *t*-way combinations if and only if it includes an ordered series of *s* covering arrays of strength *t*. This result can efficiently produce tests that cover all orders of *t*-way combinations up to any necessary order length by concatenating *t*-way covering arrays. The notion of ordered combination covers may be applied in runtime verification, assertion monitoring, and other verification and test methods that rely on checking program properties and states as code is executed.

5 References

[1] Kuhn DR, Higdon JM. Combinatorial Testing for Event Sequences. http://csrc.nist.gov/groups/SNS/acts/sequence_covar rays.html, Feb 1, 2010.

[2] Kuhn DR, Higdon JM, Lawrence JF, Kacker RN, Lei Y. Combinatorial methods for event sequence testing. 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation 2012 Apr 17 (pp. 601-609). IEEE.

[3] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2013). *Introduction to combinatorial testing*. CRC press.

[4] Chee YM, Colbourn CJ, Horsley D, Zhou J. Sequence covering arrays. SIAM Journal on Discrete Mathematics. 2013;27(4):1844-61.

[5] Farchi E, Segall I, Tzoref-Brill R, Zlotnick A. Combinatorial testing with order requirements. In2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops 2014 Mar 31 (pp. 118-127). IEEE.

[6] Sheng, Y., Sun, C., Jiang, S., & Wei, C. A. (2018). Extended covering arrays for sequence coverage. *Symmetry*, *10*(5), 146.

[7] Margalit O. Better bounds for event sequencing testing. In2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops 2013 Mar 18 (pp. 281-284). IEEE.

[8] Murray PC, Colbourn CJ. Sequence covering arrays and linear extensions. International Workshop on Combinatorial Algorithms 2014 Oct 15 (pp. 274-285). Springer, Cham.

[9] Yang CP, Dhadyalla G, Marco J, Jennings P. The effect of time-between-events for sequence interaction testing of a real-time system. In2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2018 Apr 9 (pp. 332-340). IEEE.

[10] Binder RV. Optimal scheduling for combinatorial software testing and design of experiments. In2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2018 Apr 9 (pp. 295-301). IEEE.

[11] Z. Ratliff, CSCM Manual. NIST. (*Note: this coverage measurement tool was originally named CSCM. It has been renamed Eseq to more easily distinguish between the two forms of sequence coverage.*)

[12] Kuhn, D.R., Wallace, D.R. and Gallo Jr, A.M., 2004. Software fault interactions and implications for software testing. *IEEE Trans Soft Eng,30*(6), pp.418-421.

[13] Chilenski, J. J., & Miller, S. P. (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, *9*(5), 193-200.

[14] Rahman M, Sultana D, Khatun S, Jusof MF, Shaharum SM, Yusof NA, Qaiduzzaman KM, Hasan MH, Rahman MM, Hossen MA, Begum A. T-Way Strategy for Sequence Input Interaction Test Case Generation Adopting Fish Swarm Algorithm. InInECCE2019 2020 (pp. 87-99). Springer, Singapore.

[15] Erdem E, Inoue K, Oetsch J, Pührer J, Tompits H, Yılmaz C. Answer-set programming as a new approach to event-sequence testing.

[16] Brain M, Erdem E, Inoue K, Oetsch J, Pührer J, Tompits H, Yilmaz C. Event-sequence testing using answer-set programming. International Journal on Advances in Software Volume. 2012;5.

[17] Rahman M, Othman RR, Ahmad RB, Rahman MM. Event driven input sequence t-way test strategy using simulated annealing. In2014 5th International Conference on Intelligent Systems, Modelling and Simulation 2014 Jan 27 (pp. 663-667). IEEE.

[18] Nasser AB, Zamli KZ, Alsewari AA, Ahmed BS. An elitist-flower pollination-based strategy for constructing sequence and sequence-less t-way test suite. International Journal of Bio-Inspired Computation. 2018;12(2):115-27.

[19] Rabbi K, Mamun Q, Islam MR. A Novel Swarm Intelligence Based Sequence Generator. Intl Conference on

Applications and Techniques in Cyber Security and Intelligence 2017 Jun 16 (pp. 238-246). Edizioni della Normale, Cham.

Extensions

[20] James M. Higdon, Gerald Oveson, Efficient Test of Configurations and Orders of Complex Operating Procedures Tech. Rpt., COLSA North Florida Ops, 96th Cyberspace Test Group, AFMC, Eglin AFB, FL, 4 Apr 2018.

[21] Duan F, Lei Y, Kacker RN, Kuhn DR. An Approach to T-Way Test Sequence Generation With Constraints. 2019 IEEE Intl Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2019 Apr 22 (pp. 241-250). IEEE.

[22] Garn B, Simos DE, Duan F, Lei Y, Bozic J, Wotawa F. Weighted Combinatorial Sequence Testing for the TLS Protocol. In2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2019 Apr 22 (pp. 46-51). IEEE.

[23] Oetsch J, Nieves JC. Stable-Ordered Models for Propositional Theories with Order Operators. InEuropean Conference on Logics in Artificial Intelligence 2019 May 7 (pp. 794-802). Springer, Cham.

[24] Yuster R. Perfect sequence covering arrays. Designs, Codes and Cryptography. 2020 Mar;88(3):585-93.

[25] Younis MI. MVSCA: Multi-Valued Sequence Covering Array. Journal of Engineering. 2019 Oct 29;25(11):82-91.

[26] D.R. Kuhn, I. Dominguez, R.N. Kacker and Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications", *2nd Intl Workshop on Combinatorial Testing*, Luxembourg, IWCT2013, IEEE, Mar. 2013.

[27] Zamli KZ, Othman RR, Zabil MH. On sequence based interaction testing. In2011 IEEE Symposium on Computers & Informatics 2011 Mar 20 (pp. 662-667). IEEE.

[28] Kuhn DR, Higdon JM, Lawrence JF, Kacker RN, Lei Y. Efficient methods for interoperability testing using event sequences. FLIGHT TEST SQUADRON (46TH) EGLIN AFB FL; 2012 Aug 1.

[29] Yilmaz C, Fouche S, Cohen MB, Porter A, Demiroz G, Koc U. Moving forward with combinatorial interaction testing. Computer. 2013 Dec 11;47(2):37-45.

[30] Garn B, Simos DE, Zauner S, Kuhn R, Kacker R. Browser fingerprinting using combinatorial sequence testing. In Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security 2019 Apr 1 (pp. 1-9).

[31] Ozcan M. An Industrial Study on Applications of Combinatorial Testing in Modern Web Development. In2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2019 Apr 22 (pp. 210-213).

[32] Ratliff, Z. B. (2018). *Black-box Testing Mobile Applications Using Sequence Covering Arrays.* undergraduate thesis, Texas A&M University.

[33] Ratliff ZB, Kuhn DR, Ragsdale DJ. Detecting Vulnerabilities in Android Applications using Event Sequences. In2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS) 2019 Jul 22 (pp. 159-166). IEEE.

[34] Elks DC, Deloglos C, Jayakumar A, Tantawy DA, Hite R, Gautham S. Realization Of An Automated T-Way Combinatorial Testing Approach For A Software Based Embedded Digital Device. Idaho National Lab.(INL), Idaho Falls, ID (United States); 2019 Jun 17.

[35] Becci G, Dhadyalla G, Mouzakitis A, Marco J, Moore AD. Robustness testing of real-time automotive systems using sequence covering arrays. SAE International Journal of Passenger Cars-Electronic and Electrical Systems. 2013 Apr 8;6(2013-01-1228):287-93.