Preprint: IEEE 16th Intl Conf on Software Testing, Verification and Validation Workshops, 2023

Ordered t-way Combinations for Testing State-based Systems

D. Richard Kuhn, M S Raunak, Raghu N. Kacker
National Institute of Standards & Technology
Gaithersburg, MD, USA
{kuhn, raunak, raghu.kacker}@nist.gov

***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 the 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 other states depending on the values of packet fields and the order of packets received. This paper discusses definitions for ordered *t*-way combination arrays and proves results regarding the construction of adequate blocks of test inputs, including consecutive ordering of combinations and with interleaving allowed. The application of the results to verify and test state- based systems is also illustrated.**

***Index Terms*—combinatorial coverage; combinatorial testing; software testing; structural coverage; test coverage**

I. 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, data servers, and any system in which user interaction sequence determines the states, and in turn the outcome. 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. Structural coverage metrics, such as statement coverage or branch coverage, are common practice for evaluating software test thoroughness. 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 [1]. 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 $2^2 \times C(4,2) = 24$ 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. In this paper, we discuss constructing and using tests that cover all t-way combinations in series of a given length, with possibly interleaved rows.

II. Related Work

Combinatorial aspects of input ordering have been studied in the context of event sequences. Sequence covering arrays (SCAs) were introduced in [2], [3] and further developed in [4], [5], [6], [7], [8], [9], and [10]. A sequence covering array [3], $SCA(N,S,t)$ is an $N \times S$ 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. Such sequence covers can be constructed with a simple greedy algorithm, although more compact results can be achieved with a variety of search algorithms, including answer set programming [11], [12], simulated annealing [13], and machine learning oriented algorithms [14], [15], [16].

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 array

Combinatorial testing with constraints on the order in which values and combinations are applied in tests was analyzed in [5] and [17]. Extended covering arrays that consider the sequence of values in each test were defined in [6]. The notion of a perfect sequence covering array was introduced in [18]. Another structure defined as a sequence covering array of t-way combinations has been termed a multi-valued sequence covering array [19], which extended the notion of event sequence cover to possibly interleaved sequences of combinations, without repetition. A method for including constraints in event sequences was introduced in [20], using an FSM to ensure coverage of all valid event sequences, while also allowing for repetition. Another automata-based approach to generating sequences was developed by [21], providing reduced test set size (compared with conventional SCAs) while also incorporating information from previous tests for optimal error detection. Properties of combination sequences were studied in [22], combining configuration and the order of combinations while also considering constraints. In this paper we consider repeated occurrences of value combinations, and properties of ordered combinations with or without interleaving, along with methods of constructing ordered combination arrays.

Sequence covering arrays have found extensive use in practical applications [23], [24], [25], [26], [27], [28], [29], [30], [31], [20]. These applications typically involve cases where an error is triggered when two or more combinations occur in series within inputs, but other combinations may occur between those that are significant to triggering the error. Such situations may occur in protocol testing, graphical user interfaces, and others. However, better test methods are also needed for errors that are only revealed when two or more combinations occur consecutively in input, without interleaving of other combinations. The methods described in this paper address this consecutive ordered combination problem as well.

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

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 Combination array

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).

Definition 2. An ordered combination cover, designated $OCC(N, s, t, p, v)$, 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

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

Fig. 3: Ordered Combinations of Parameter Values

combinations may appear multiple times in a combination order.

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 4 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 = "$p_1 \bigwedge p_2$" and condition B = "$p_1 \bigwedge \neg p_2$", 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 $p_1p_2 = 11$ $*\!\!\rightarrow$ $p_1p_2 = 10$, 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.

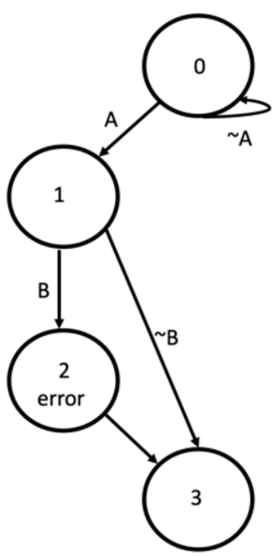

Fig. 5: Example Finite State Machine

Note that the definition allows 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 read/write 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, a sequence of t-way combinations containing s combinations input in sequence with possible interleaving is an s-order of t-way combinations. For example, a 2-order of 3-way combinations could be

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

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

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

An OCC covers all s-orders of t-way combinations of the v values of the p parameters. Because a t-tuple is included s times in an s-order, 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 order tuples to be covered is

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

The number of combination orders 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-orders of 2-way parameter combinations.

That is, the test array includes every solution of $(p_wp_x = v_1v_2)$ $*\!\!\rightarrow$ $(p_yp_z = v_3v_4)$, of which there are $(C(4, 2)\times 2^2)^2 = 576$ instances. For example, each of the four possible settings of p_1p_2 is followed by each of the four possible settings of p_3p_4 somewhere in the table (distinguished by color). That is, (p_1p_2 = 11) in line 1 is followed by ($p_3p_4 = 01$), ($p_3p_4 = 11$), ($p_3p_4 = 00$), ($p_3p_4 = 10$) in lines 3, 4, 5, and 7, respectively, highlighted in yellow (also for (p_1p_2 = 00)). Sequences for (p_1p_2 = 01) are shown in green, plus line 14, which provides a ($p_3p_4 = 00$) for both (p_1p_2 = 01) and (p_1p_2 = 10). Sequences for (p_1p_2 = 10) are highlighted in blue.

	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)

A. *Constructing Ordered Combination Covers*

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 could 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 result. Checking that a test array is a covering array can be done efficiently, making it practical to ensure s-orders of t-way combinations in large-scale testing. Note that constraints across rows are not considered, but there may be constraints among value combinations in the covering arrays from which the OCC is constructed.

Theorem 1 (OCC Coverage)**.** *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 $CA_1, CA_2, \ldots, CA_s$., 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.

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 $\mathcal{A}_1 \ldots \mathcal{A}_2 \ldots \mathcal{A}_k$ and the row containing these combinations as row i. $\mathcal{A}_1 \ldots \mathcal{A}_2 \ldots \mathcal{A}_k$ 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.

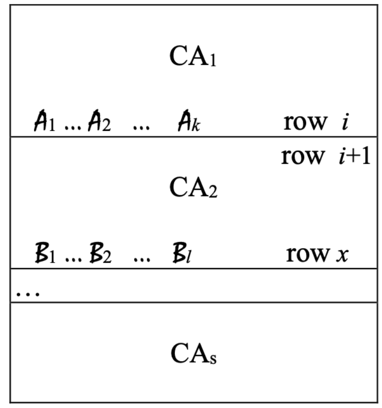

Fig. 7: OCC constructed from covering arrays

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 $\mathcal{B}_1 \ldots \mathcal{B}_2 \ldots \mathcal{B}_l$ and the row containing these combinations as row x. $\mathcal{B}_1 \ldots \mathcal{B}_2 \ldots \mathcal{B}_l$ 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 $\mathcal{A}_1 \ldots \mathcal{A}_2 \ldots \mathcal{A}_k$ must be followed by any t-way combination in some row of CA_2.

Note that any 2-order $c_1 \;{*}{\rightarrow}\; c_2$ where c_1 is one of A_i and c_2 is one of B_i could not have been covered until row x of CA_2 because the B_i tuples are those that had not been covered in CA_2 before row x (and after row i). Assume that $c_1 \;{*}{\rightarrow}\; c_2$ 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 $c_1 \;{*}{\rightarrow}\; c_2$ 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. □

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. 8. 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 state that can only be reached by a particular sequence 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.

B. 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 [32]. 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 $4 \times C(4, 2) \times 4 \times C(4, 2)$ orders. One can measure the degree to which these orders are covered and output any missing orders, as shown in Fig. 9(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 9(c), which shows the missing ⟨parameter indices⟩:⟨value combination⟩ → ⟨parameter indices⟩:⟨value combination⟩ for 2-way orders.

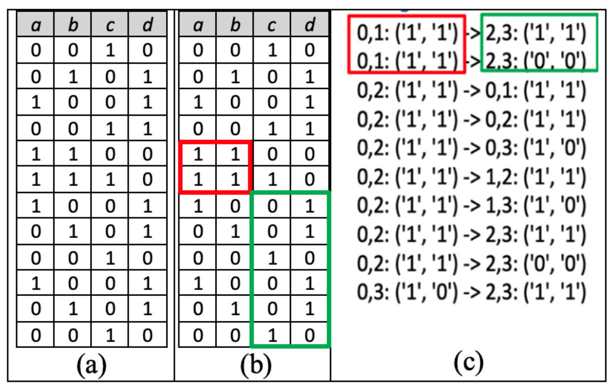

Fig. 9: Missing combination orders

Fig. 10 illustrates the output of the Corder tool for a simple example. Further explanation and additional examples may be found in [33]. 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 [32].

```
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. 10: Example output of Corder tool

The Corder tool provides the following output:

• file = input file name for 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: 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

Static coverage refers to 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. 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.

IV. Ordered Coverage of Adjacent Combinations

In some testing problems, errors may be revealed only when a series of particular inputs appear in sequence consecutively. That is, we remove the possibility of interleaving with other combinations not in the specified order, in contrast with the combination order of Definition 1, so that the ordering refers to combinations in rows that are adjacent in the test array.

Definition 3. An adjacent combination order $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_s$ of t-way combinations, abbreviated s-order, is a set of t-way combinations in s consecutive rows. Each c_i is a t-way combination of parameter values. The notation $a \rightarrow b$ denotes the presence of combination a immediately followed by combination b, where a and b are in adjacent rows, i.e., combination a is in row i followed by combination b in row $i+1$.

Example. For Boolean parameters a, b, c in Fig. 11, $ab = 00$ is followed consecutively by $ab = 10$ $ac = 11$ and $bc = 01$.

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

Fig. 11: Ordered combinations of parameter values

When all s-orders of t-way combinations of the input parameters have been covered in this manner, where only adjacent combinations are considered, it is referred to as an adjacent ordered combination cover (OCCa). For the OCCa, 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 followed consecutively by other t-way combinations in other rows.

Definition 4. An adjacent ordered combination cover, $OCCa(N, s, t, p, v)$, 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 a consecutive set of rows. Permutations of parameter value combinations may appear multiple times in a combination order. For example, a particular adjacent 2-order of 2-way combinations may be $(p_1p_2 = 01) \rightarrow (p_2p_4 = 11)$.

A. Constructing Adjacent Ordered Combination Covering Arrays

To use adjacent ordered combination covers in testing, it is necessary to efficiently construct arrays of test inputs. There is a straightforward construction for such arrays, as shown in this section.

For a given set S of k symbols, a deBruijn sequence $D(k, n)$ includes every n-length permutation of the symbols in S, and practical algorithms for constructing such sequences are available. The length of a deBruijn sequence is k^n, and no shorter length sequence covering all the n-length permutations is possible.

OCCa Construction. We can construct an adjacent ordered combination covering array $OCCa(N, s, t, p, v)$ with the following steps:

1) Generate a covering array of desired strength for the input model of the SUT.
2) Number the rows of the covering array sequentially, from 1 to k, for a covering array with k rows.
3) Generate a deBruijn sequence $D(k, s)$ of the k row indices.
4) For each row index i in the sequence, write row i from the covering array. After the last row, append the initial $s-1$ rows of the covering array, resulting in $N = k^s + s - 1$ rows.

It is easy to show that any s-order of adjacent combinations will be generated by the OCCa construction procedure above.

Theorem 2 (OCCa Construction)**.** *Any adjacent combination order $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_s$ of t-way combinations will be found in the array generated by the OCCa construction procedure.*

Proof. By definition each combination c_i must occur somewhere in the covering array produced in step 1. Two or more of the c_i combinations may occur in the same row. Label the row number where each c_i appears in the covering array as R_i. By definition, the deBruijn sequence generated in step 3 must contain R_i $R_{i+1} \ldots R_s$, so the rows associated with each of these indices written in step 4 must appear as consecutive rows in the completed OCCa. Appending the initial $s-1$ rows of the covering array allows orders of combinations in the tail of the deBruijn sequence followed by those in the start of the sequence.

□

Example: Fig. 12 shows a covering array, $CA(2, 9, 2)$ of t = 2-way combinations of 9 variables of 2 values each.

1	0	0	1	0	0	0	1	1	1
2	0	1	0	1	1	0	0	0	1
3	1	0	0	1	0	1	0	1	0
4	1	1	1	0	1	1	0	1	1
5	1	1	0	0	0	0	1	0	0
6	0	0	1	1	1	1	1	0	0

Fig. 12: 2-way covering array of 9 binary variables

Next we produce a deBruijn sequence of length 36 with the indices of the covering array rows 1...6:

$$112131415162232425263343536445465566$$

Listing the row of variable values from the array in Fig. 12 indexed by each number in the deBruijn sequence produces the adjacent ordered combination covering array shown in Fig. 13. The OCCa contains 6^2 = 36 rows corresponding to the 36 indices in the deBruijn sequence.

B. Using Adjacent Ordered Combination Covering Arrays

An important problem in testing is the discovery of unintended functions or unspecified paths in a program. To do so, it is useful to generate tests that thoroughly cover the specification, such as in protocol testing. Similarly ordered coverage can be useful in catching potential race condition situations in complex software. When a state machine is specified, a number of methods are available to generate tests that will cover the paths specified [34], [35], typically by processing the FSM definition and producing conditions in tests based on the specification.

Example: Using the covering array shown in Fig. 4, we construct the OCCa for this configuration, shown in Fig. 14. Comparing with the (non-adjacent) OCC of Fig. 8, it is easy to recognize many adjacent combination orders that occur in Fig. 14 that are not present in the OCC of Fig. 8. For example, $(p_1p_2 = 00) \rightarrow (p_3p_4 = 11)$ on adjacent rows. It can also be verified that any adjacent order of two 2-way combinations occurs somewhere in Fig. 14.

0	0	1	0	0	0	1	1	1
0	0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0	1
0	0	1	0	0	0	1	1	1
1	0	0	1	0	1	0	1	0
0	0	1	0	0	0	1	1	1
1	1	1	0	1	1	0	1	1
0	0	1	0	0	0	1	1	1
1	1	0	0	0	0	1	0	0
0	0	1	0	0	0	1	1	1
0	0	1	1	1	1	1	0	0
0	1	0	1	1	0	0	0	1
0	1	0	1	1	0	0	0	1
1	0	0	1	0	1	0	1	0
0	1	0	1	1	0	0	0	1
1	1	1	0	1	1	0	1	1
0	1	0	1	1	0	0	0	1
1	1	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0	1
0	0	1	1	1	1	1	0	0
1	0	0	1	0	1	0	1	0
1	0	0	1	0	1	0	1	0
1	1	1	0	1	1	0	1	1
1	0	0	1	0	1	0	1	0
1	1	0	0	0	0	1	0	0
1	0	0	1	0	1	0	1	0
0	0	1	1	1	1	1	0	0
1	1	1	0	1	1	0	1	1
1	1	1	0	1	1	0	1	1
1	1	0	0	0	0	1	0	0
1	1	1	0	1	1	0	1	1
0	0	1	1	1	1	1	0	0
1	1	0	0	0	0	1	0	0
1	1	0	0	0	0	1	0	0
0	0	1	1	1	1	1	0	0
0	0	1	1	1	1	1	0	0
0	0	1	0	0	0	1	1	1

Fig. 13: OCCa generated from covering array in Fig. 12

0	0	1	0
0	0	1	0
0	1	0	1
0	0	1	0
1	0	0	1
0	0	1	0
1	1	1	0
0	0	1	0
0	1	0	0
0	0	1	0
0	0	1	1
0	1	0	1
0	1	0	1
1	0	0	1
0	1	0	1
1	1	1	0
0	1	0	1
0	1	0	0
0	1	0	1
0	0	1	1
1	0	0	1
1	0	0	1
1	1	1	0
1	0	0	1
0	1	0	0
1	0	0	1
0	0	1	1
1	1	1	0
1	1	1	0
0	1	0	0
1	1	1	0
0	0	1	1
0	1	0	0
0	1	0	0
0	0	1	1
0	0	1	1
0	0	1	0

Fig. 14: OCCa for covering array in Fig. 4

V. THERAC 25: A POIGNANT EXAMPLE

To illustrate the importance and usefulness of designing test cases based on OCC and OCCa, let us examine one of the well known fatal software failures in history, the Therac-25. [36]. Therac-25 was an expensive radiation treatment device designed to treat cancer by administering energy beams to destroy tumors. It operated in three modes: *field light* mode to adjust the position of the beam using light simulation, an *electron mode* to administer low energy electron radiation, and *x-ray mode* to beam high energy photons flattened over a target area. The turntable adjustments and other safety features were all provided with computer control. After putting the patient on the treatment table, the operator went to the user interface outside the treatment room, choose the beam type, set all other parameters on the console, and administered the radiation beam with the command “B”. Figure 15 shows the operator user interface.

Between 1985 and 1987, the machine administered massive overdose of radiation to six patients leading to three serious injuries and three deaths. The extensive testing and safety analysis of the system failed to discover the corner-case errors before deployment, which led to a false sense of confidence about the reliability of the device. Even when early incidents were reported, the manufacturer’s engineers could not reproduce the error and kept claiming that the device was incapable of administering overdose.

In reality, there were multiple software bugs and other safety failures associated with Therac-25. One of the primary errors was due to an underlying race condition of a shared variable which could be manifested through a very specific order of interaction in the user interface.

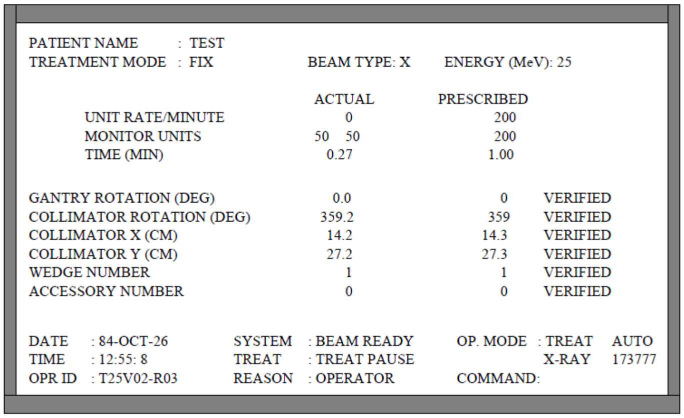

Fig. 15: Therac 25 Operator Interface

If the operator chose Beam Type to be “X” for x-ray by mistake, set all the parameters, and then went back to change the Beam Type to be “E” for electron, stepped through the parameters quickly with a series of enters and then hit the “B” command for beam, the machine would administer a very high energy x-ray instead of low energy electron. The device did recognize that an overdose was administered and paused. But due to the cryptic error message, which did not mention the overdose and since the machine used to frequently go into the “Paused” state with benign error messages, it led operators to simply press “P” for proceed to continue with the beam administration resulting in repeated massive overdose on most of those fatal cases.

Let us illustrate this with a very similar test scenario illustrated above in Fig. 4. Suppose p_1, p_2, p_3, and p_4 are four boolean variables representing the states of some user interactions controlling the device. Let p_1 represents whether sufficient time has passed since last test interaction at the console. Let p_2 and p_3 stand for selection of x-ray beam (“X”) and electron beam (“E”) respectively, and let p_4 represent the administration of the beam (“B”). There is a constraint in this example where p_2 and p_3 can not both be 1, i.e., both x-ray and electron can not be selected simultaneously.

The fatal error is manifested for an ordered sequence of tests where $p_1p_2p_3p_4 = 1100$ immediately followed by the test $p_1p_2p_3p_4 = 0011$. Let us revisit the FSM that is shown in Fig. 5. Let the conditions A and B the following:

$$A = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4$$

$$B = \neg p_1 \wedge \neg p_2 \wedge p_3 \wedge p_4$$

Only A followed by B will lead to the error state. Any other path will not discover the error. Fig 16 shows the partial OCCa for Therac-25 that would have enumerated this scenario. Here test case t_m represents the selection of “X” for x-ray, but not administering the beam, and then a selection of “E” followed by the command “B” for administering the beam.

Test	p_1	p_2	p_3	p_4
t_1	1	1	0	0
t_2	1	0	1	0
t_3	1	0	0	1
t_4	0	0	1	0
t_5	0	1	0	1
...				
t_m	1	1	0	0
t_n	0	0	1	1
...				

Fig. 16: Therac-25 partial OCCa

The two test cases are also exercised without sufficient delay. In case of Therac-25, this delay was 8 seconds. This sequence of test interactions would be guaranteed to be exercised if and only if an adjacent ordered combinatorial coverage, or OCCa was generated and used for the Therac-25 finite state machine. Although a potential threat to validity can arise from the fact that this example is very specific and designed with the knowledge of the flaw, it is not unreasonable that for such safety critical system, an extensive set of tests incorporating the time delay and enumerating all relevant combinations of user interactions would indeed be systematically created and exercised. In fact, we argue to ensure the complete safety of such complex systems, this level of extensive test case and user interaction generation and execution is necessary. OCC and OCCa allow us to do that.

VI. Discussion and Conclusions

This paper considers 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 for 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. Additional applications may include checking digital designs, where sequential circuits in particular present challenges in verification and testing. Automatic test pattern generation (ATPG) for sequential circuit designs requires a number of heuristics, as there is no universally applicable most effective method. Ordered combination covers may have potential for improving the efficiency and effectiveness of ATPG methods. Another important application area can be autonomous systems, especially self driving vehicles. A sequence of states and events in a specific order may lead to unexpected behavior from autonomous systems. We plan to study these application areas in large-scale experiments, along with the inclusion of constraints on combination orders.

Note: Sections I to III of this paper were originally published as a NIST technical report [33].

References

[1] D. R. Kuhn, I. Dominguez Mendoza, R. N. Kacker, and Y Lei. Combinatorial coverage measurement concepts and applications. In *2013 IEEE Sixth Intl Conf on Software Testing, Verification and Validation Workshops*, pages 352–361, 2013.

[2] D. R. Kuhn and J. M Higdon. Combinatorial methods for event sequence testing, 2010.

[3] D. R. Kuhn, J. M Higdon, J. F Lawrence, R. N. Kacker, and Y. Lei. Combinatorial methods for event sequence testing. In *2012 IEEE Fifth Intl Conf on Software Testing, Verification and Validation*, pages 601–609, 2012.

[4] Yeow Meng Chee, Charles J Colbourn, Daniel Horsley, and Junling Zhou. Sequence covering arrays. *SIAM Journal on Discrete Mathematics*, 27(4):1844–1861, 2013.

[5] Eitan Farchi, Itai Segall, Rachel Tzoref-Brill, and Aviad Zlotnick. Combinatorial testing with order requirements. In *2014 IEEE Seventh Intl Conf on Software Testing, Verification and Validation Workshops*, pages 118–127, 2014.

[6] Yunlong Sheng, Chao Sun, Shouda Jiang, and Chang'an Wei. Extended covering arrays for sequence coverage. *Symmetry*, 10(5):146, 2018.

[7] Oded Margalit. Better bounds for event sequencing testing. In *2013 IEEE Sixth Intl Conf on Software Testing, Verification and Validation Workshops*, pages 281–284, 2013.

[8] Patrick C Murray and Charles J Colbourn. Sequence covering arrays and linear extensions. In *Intl Workshop on Combinatorial Algorithms*, pages 274–285. Springer, 2014.

[9] Chek Pin Yang, Gunwant Dhadyalla, J. Marco, and Paul Jennings. The effect of time-between-events for sequence interaction testing of a real-time system. In *2018 IEEE Intl Conf on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 332–340, 2018.

[10] Robert V Binder. Optimal scheduling for combinatorial software testing and design of experiments. In *2018 IEEE Intl Conf on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 295–301, 2018.

[11] Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yılmaz. Answer-set programming as a new approach to event-sequence testing. 2011.

[12] Martin Brain, Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yilmaz. Event-sequence testing using answer-set programming. *Intl Journal on Advances in Software Volume*, 5, 2012.

[13] Mostafijur Rahman, Rozmie Razif Othman, R Badlishah Ahmad, and Md Mijanur Rahman. Event driven input sequence t-way test strategy using simulated annealing. In *2014 5th Intl Conf on Intelligent Systems, Modelling and Simulation*, pages 663–667, 2014.

[14] Abdullah B Nasser, Kamal Z Zamli, AbdulRahman A Alsewari, and Bestoun S Ahmed. An elitist-flower pollination-based strategy for constructing sequence and sequence-less t-way test suite. *Intl Journal of Bio-Inspired Computation*, 12(2):115–127, 2018.

[15] Khandakar Rabbi, Quazi Mamun, and Md Rafiqul Islam. A novel swarm intelligence based strategy to generate optimum test data in t-way testing. In *Intl Conf on Applications and Techniques in Cyber Security and Intelligence*, pages 247–255. Springer, 2017.

[16] Kamal Z. Zamli and Md Abdul Kader. Sequence t-way test generation using the barnacles mating optimizer algorithm. In *2021 10th International Conference on Software and Computer Applications*, pages 88–93, 2021.

[17] Feng Duan, Y. Lei, R. N. Kacker, and D R Kuhn. An approach to t-way test sequence generation with constraints. In *2019 IEEE Intl Conf on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 241–250, 2019.

[18] Raphael Yuster. Perfect sequence covering arrays. *Designs, Codes and Cryptography*, 88(3):585–593, 2020.

[19] Mohamed Issam Younis. Mvsca: multi-valued sequence covering array. *Journal of Engineering*, 25(11):82–91, 2019.

[20] David Adamo, Dmitry Nurmuradov, Shraddha Piparia, and Renée Bryce. Combinatorial-based event sequence testing of android applications. *Information and Software Technology*, 99:98–117, 2018.

[21] Achiya Elyasaf, Eitan Farchi, Oded Margalit, Gera Weiss, and Yeshayahu Weiss. Combinatorial sequence testing using behavioral programming and generalized coverage criteria. *arXiv preprint arXiv:2201.00522*, 2022.

[22] J. M Higdon and G. Oveson. Efficient test of configurations and orders of complex operating procedures. Technical report, COLSA North Florida Ops, 96th Cyberspace Test Group, AFMC, Eglin AFB, FL, 2018.

[23] Kamal Z Zamli, Rozmie R Othman, and Mohd H Mohamed Zabil. On sequence based interaction testing. In *2011 IEEE Sym. Computers & Informatics*, pages 662–667, 2011.

[24] D. R. Kuhn, J. M Higdon, J. F Lawrence, R. N. Kacker, and Y. Lei. Efficient methods for interoperability testing using event sequences. Technical report, FLIGHT TEST SQUADRON (46TH) EGLIN AFB FL, 2012.

[25] Cemal Yilmaz, Sandro Fouché, Myra B Cohen, Adam Porter, Gulsen Demiroz, and Ugur Koc. Moving forward with combinatorial interaction testing. *Computer*, 47(2):37–45, 2013.

[26] Bernhard Garn, D. E. Simos, Stefan Zauner, Rick Kuhn, and R Kacker. Browser fingerprinting using combinatorial sequence testing. In *Proc of the 6th Annual Sym. Hot Topics in the Science of Security*, pages 1–9, 2019.

[27] M. Ozcan. Applications of practical combinatorial testing methods at siemens industry inc., building technologies division. In *2017 IEEE Intl Conf on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 208–215, 2017.

[28] Z. B Ratliff. *Black-box Testing Mobile Applications Using Sequence Covering Arrays*. PhD thesis, 2018.

[29] Z. B Ratliff, D. R. Kuhn, and Daniel J Ragsdale. Detecting vulnerabilities in android applications using event sequences. In *2019 IEEE 19th Intl Conf on Software Quality, Reliability and Security (QRS)*, pages 159–166, 2019.

[30] Dr Carl Elks, Christopher Deloglos, Athira Jayakumar, Dr Ashraf Tantawy, Rick Hite, and Smitha Gautham. Realization of a automated t-way combinatorial testing approach for a software based embedded digital device. Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2019.

[31] Graciela Becci, Gunwant Dhadyalla, Alexandros Mouzakitis, James Marco, and Andrew David Moore. Robustness testing of real-time automotive systems using sequence covering arrays. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 6(2013-01-1228):287–293, 2013.

[32] NIST. Combinatorial coverage measurement tool user guide. Technical report, National Institute of Standards and Technology, January 2011.

[33] D. R. Kuhn, M S Raunak, and R. N. Kacker. Ordered t-way combinations for testing state-based systems. Technical Report CSWP 26, April 2022.

[34] Gregor V Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, 1994.

[35] A Petrenko, N Yevtushenko, and G v Bochmann. Testing deterministic implementations from nondeterministic fsm specifications. In *Testing of Communicating Systems*, pages 125–140. Springer, 1996.

[36] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

Disclaimer