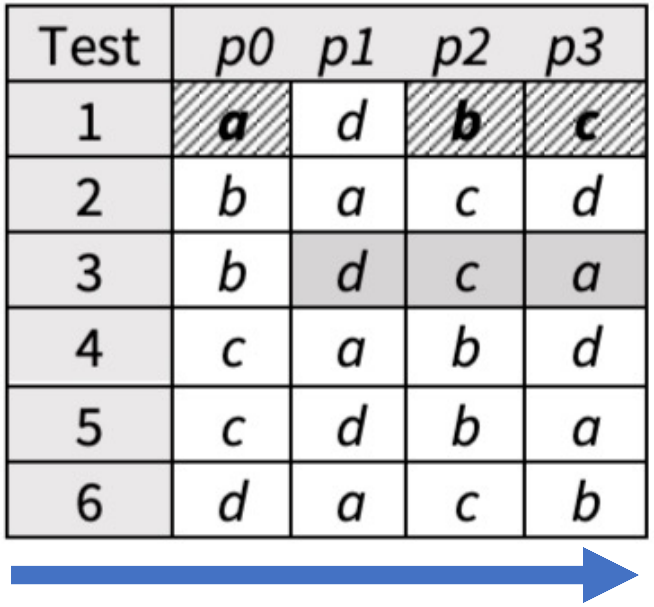
# Ordered *t*-way Combinations for Testing State-based Systems

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

# Why do we need ordered input combinations?

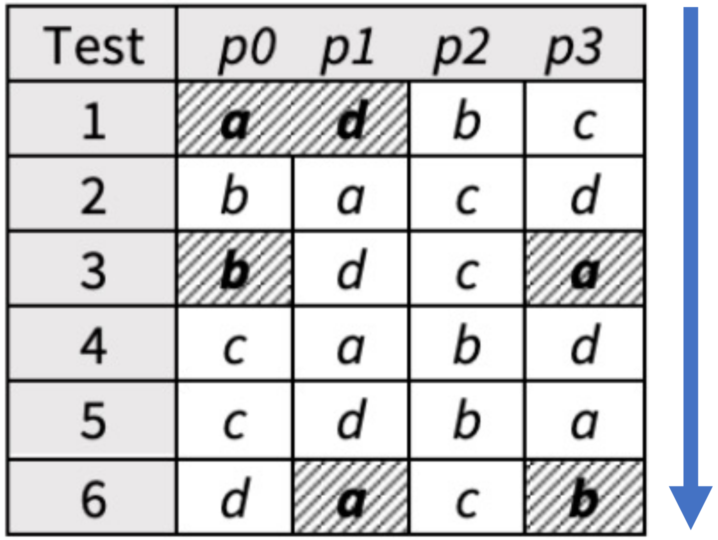
- Covering arrays are great, but sequence of inputs can affect results when state is maintained by the system (nearly all)
- Sequence covering arrays handle sequences of events, but events may be complex and involve multiple parameters, combinations
- States change according to inputs, combinations of input values
- So we want to consider the order of inputs of combinations in test set

# What are ordered combinations?



| 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 |

- Sequence covering, of <u>events</u> *a, b, c, d*
- Sequence covering array has all sequences of events for some specified length, non-repeating
- Each row is one test <u>sequence</u>

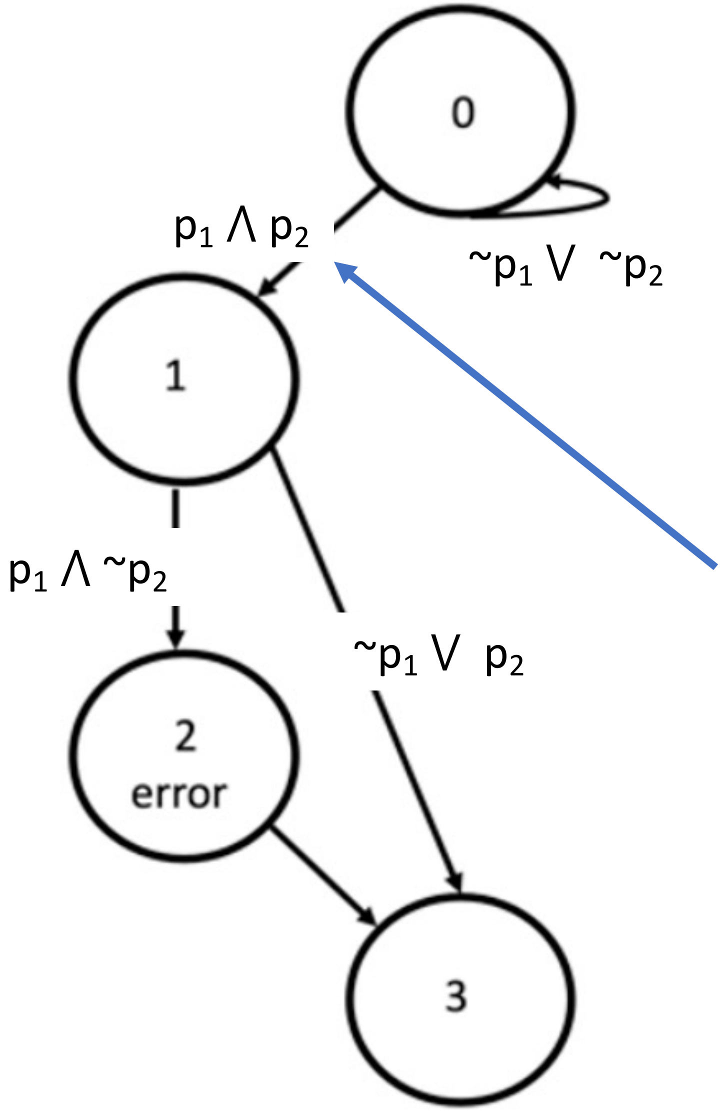


| Test | p0 | p1 | p2 | p3 |
|---|---|---|---|---|
| 1 | **a** | **d** | b | c |
| 2 | b | a | c | d |
| 3 | **b** | d | c | **a** |
| 4 | c | a | b | d |
| 5 | c | d | b | a |
| 6 | d | **a** | c | **b** |

Ordered combinations

- Combination order $c_1 *\rightarrow c_2 *\rightarrow \ldots *\rightarrow c_s$ of <u>*s* combinations</u> of <u>*t* parameter values</u>, abbreviated <u>*s*-order</u>, is a set of <u>*t*-way</u> combinations in *s* rows
- Each row is one set of test inputs
- Ordering of combinations as rows entered sequentially
- Example: $p0p1 = ad *\rightarrow p2p3 = cb$

# Order of covering array tests affects error detection



| 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 |

Example:

$p_1 \wedge p_2$

not followed by

$p_1 \wedge \sim p_2$

Reordering tests to:
1
2
4
3
5
6

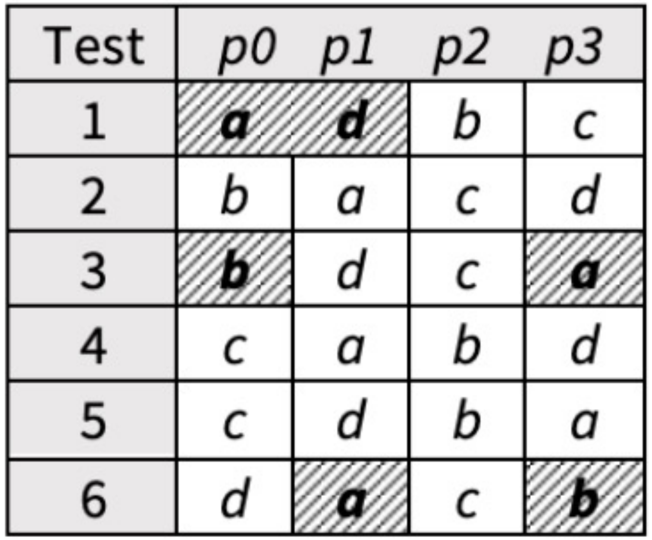
solves the problem

# Ordered combination cover

- 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.
- Number of combination order tuples to cover, for $s$-orders of $t$-way combinations of $p$ parameters with $v$ values each:

$$v^t C(p, t)^s$$

How can we find these ordered combination covers (OCC) efficiently?



| 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*** |

# Generating ordered combination covers (OCC) ?

- The problem turns out to be easy!
- Theorem (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.*

*So,*

1. make a *t*-way covering array
2. write *s* copies of it

# Ordered coverage of adjacent combinations

- An adjacent combination order $c_1 \rightarrow c_2 \rightarrow . . . \rightarrow c_s$ of *t*-way combinations, abbreviated *s*-order, is a set of *t*-way combinations in *s* consecutive rows.
- No interleaving between the ordered combinations, i.e.,

  Ordered: $c_1 \; {*}\!\rightarrow c_2$ $c_1$ is *eventually* followed by $c_2$
  Adjacent ordered : $c_1 \rightarrow c_2$ $c_1$ is *immediately* followed by $c_2$
- Ordered combinations with added constraint that rows are adjacent, i.e., for $c_1 \rightarrow c_2$ where $c_1$ and $c_2$ are in consecutive rows
- We need to produce an ordering of combinations such that every *t*-length permutation of combinations occurs as tests (rows) are input sequentially

➡ This can be done with a deBruijn sequence

# deBruijn sequences

- Studied in early 20th century, many properties proved by deBruijn
- For a given set *S* of *k* symbols, a deBruijn sequence D(*k*, *n*) includes every *n*-length permutation of the symbols in *S*
- length of a deBruijn sequence is $k^n$, and no shorter length sequence covering all the *n*-length permutations is possible
- Probably re-invented by every hacker on the planet (to crack key code locks)

$$D(3,2) = \underbrace{001021122}_{\underline{\text{9 digits}}}00102\ldots$$

key codes length 2: <u>18 digits</u>

$$\underbrace{00,01,02,10,11,12,20,21,22}_{001021122}$$

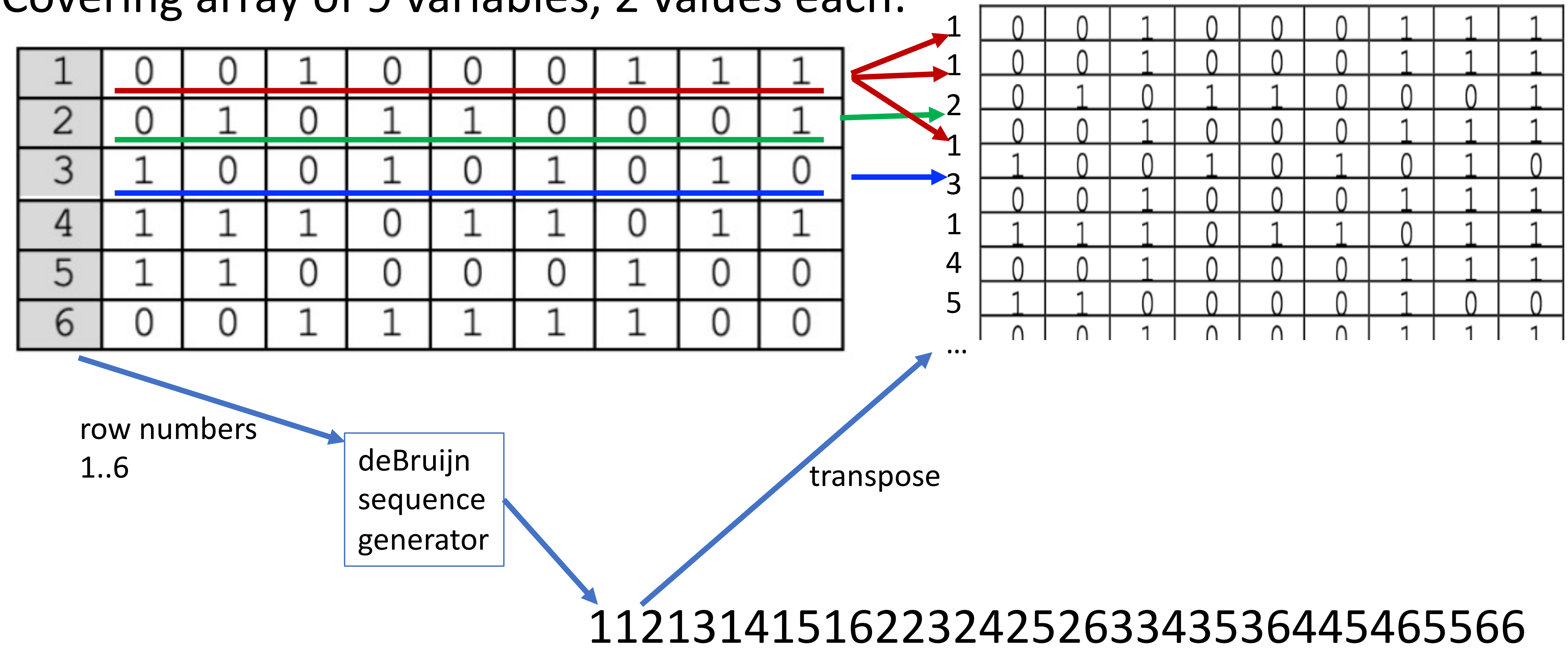
No ‘enter’ key:
628 key presses
instead of
3,125 for 4-digit code

# Generating adjacent **ordered combination covers**

1. Generate a covering array of desired strength for the input model of the system under test.
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.

# Example

- Covering array of 9 variables, 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 |

row numbers
1..6

deBruijn
sequence
generator

112131415162232425263343536445465566

transpose

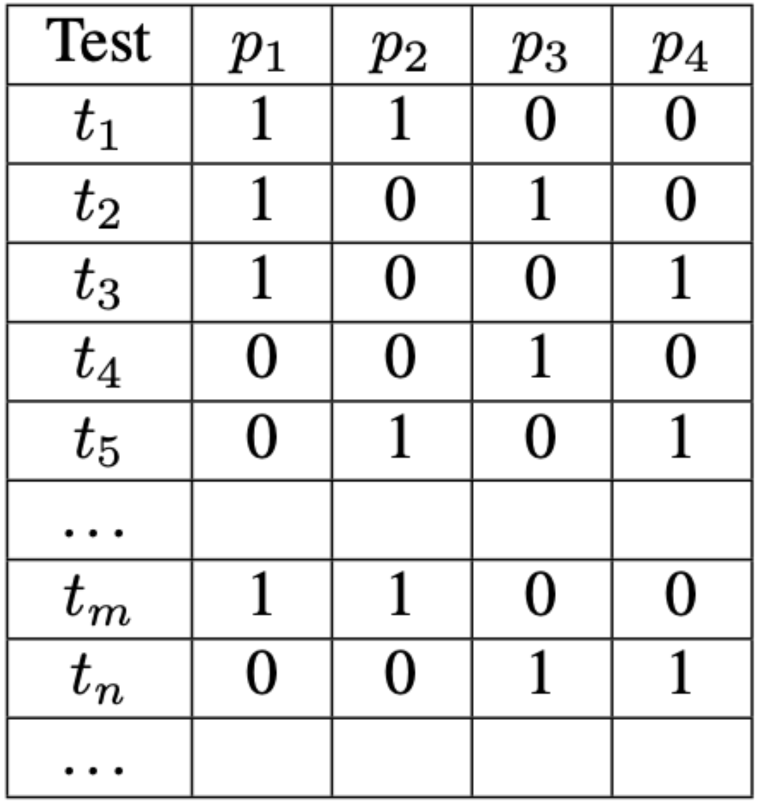
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| … | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

# Using adjacent ordered combinations

- Therac-25 example - radiation therapy machine fatal errors, 1985-1987 - widely known in software safety
- Multiple bugs and safety failures
- Critical, fatal race condition - error occurs if X-ray beam selected, changed to electron without min time between selections

# Testing to detect error

$p_1$ = min time between option selections
$p_2$ = X-ray beam selected
$p_3$ = electron beam selected
$p_4$ = start 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 |
| $\dots$ | | | | |
| $t_m$ | 1 | 1 | 0 | 0 |
| $t_n$ | 0 | 0 | 1 | 1 |
| $\dots$ | | | | |

Then, error only detected if test set contains a sequence of:

$p_1p_2p_3p_4$ = 1100 (X-ray beam selected)

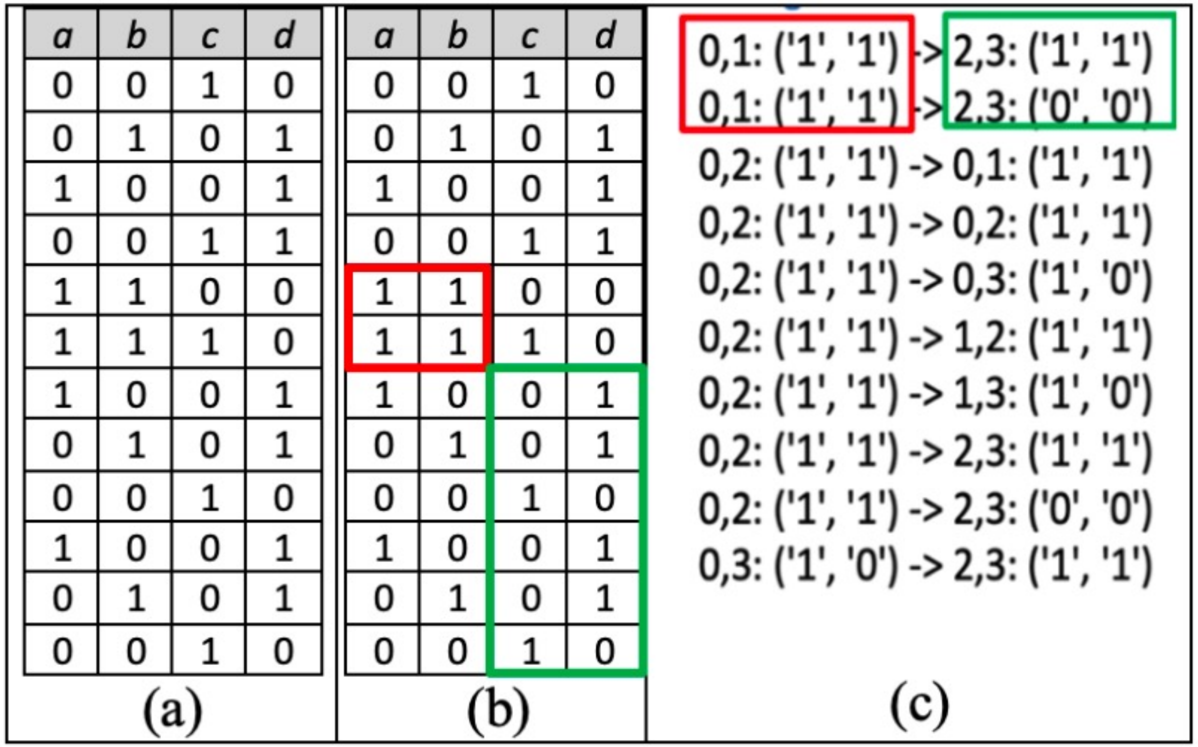
followed by

$p_1p_2p_3p_4$ = 0011 (not min time so X-ray still on, electron selected & beam started)

OCCa guaranteed to contain this sequence.
Unlikely that other test set would, even if very large

# Combination order coverage measurement

- Prototype tool to analyze coverage and output combination orders that are not found in test set
- Example:
    - test set (a) with 4 variables, 12 tests
    - for ab = 11, covered combinations for cd following are  cd = 01 and cd = 10
    - missing combinations output (c)



| *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 |

(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)

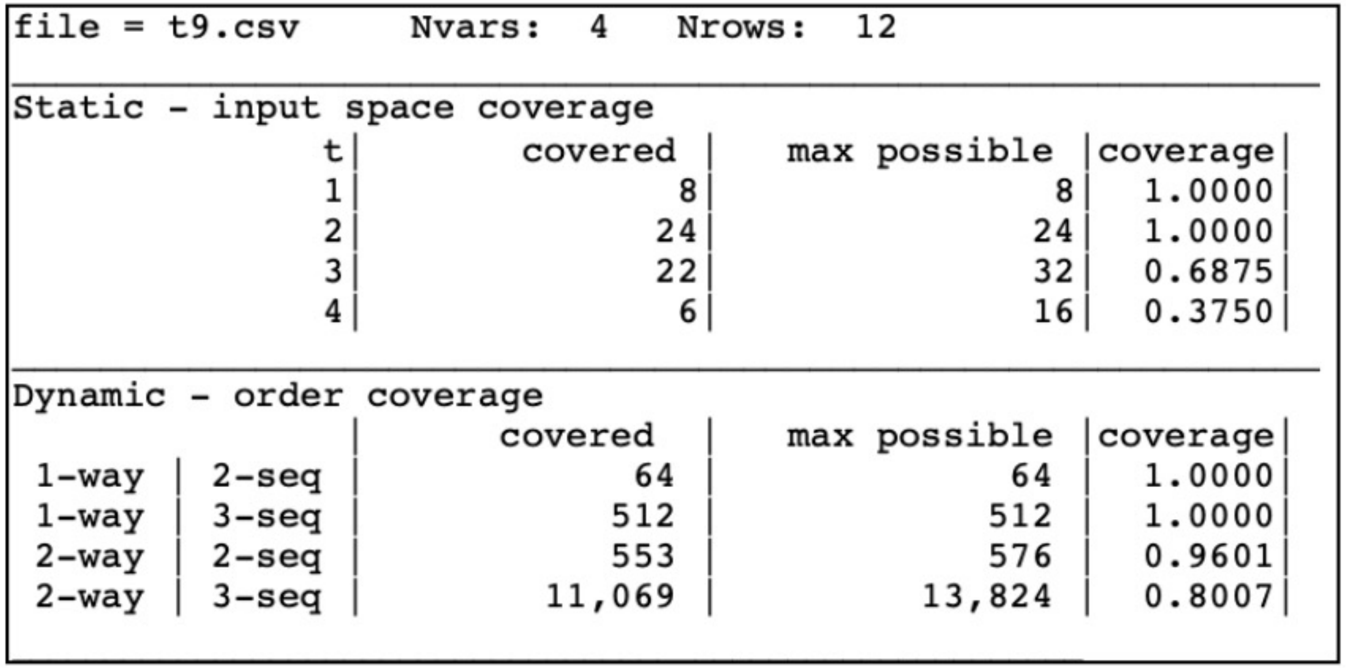
```
0,1: ('1', '1') -> 2,3: ('1', '1')
0,1: ('1', '1') -> 2,3: ('0', '0')
0,2: ('1', '1') -> 0,1: ('1', '1')
0,2: ('1', '1') -> 0,2: ('1', '1')
0,2: ('1', '1') -> 0,3: ('1', '0')
0,2: ('1', '1') -> 1,2: ('1', '1')
0,2: ('1', '1') -> 1,3: ('1', '0')
0,2: ('1', '1') -> 2,3: ('1', '1')
0,2: ('1', '1') ->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

(c)

# Coverage statistics

Coverage stats for

- static (simple) t-way coverage and
- dynamic (ordered combinations) coverage



```
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|
```

# Future directions

- Empirical data on real-world problems
  - many possible applications
  - network protocols
  - automated test pattern generation for sequential circuits
  - blockchain smart contracts
- Comparison with random tests, structural coverage criteria
  - e.g., fuzz testing
  - also see if we can improve on standard CT
- Inclusion of constraints on sequencing
- Tool support

# Please contact us if you're interested!

Rick Kuhn, M S Raunak, Raghu Kacker
{kuhn, raunak, raghu.kacker,}@nist.gov

http://csrc.nist.gov/acts