Combinatorial Methods for System and Software Testing

Rick Kuhn
National Institute of Standards and Technology
Gaithersburg, MD

NASA/OSD Science of Test Workshop
April 4, 2017
What is NIST and why are we doing this?

• US Government agency, whose mission is to support US industry through developing better measurement and test methods
• 3,000 scientists, engineers, and staff including 4 Nobel laureates
• Project goal – improve cost-benefit ratio for testing
What is the empirical basis?

- NIST studied software failures in 15 years of FDA medical device recall data
- What causes software failures?

**Interaction faults:** e.g., failure occurs if

```plaintext
altitude = 0 && volume < 2.2
```

(interaction between 2 factors)

So this is a 2-way interaction

=> testing all pairs of values can find this fault
How are interaction faults distributed?

- Interactions e.g., failure occurs if
  - pressure < 10 (1-way interaction)
  - pressure < 10 & volume > 300 (2-way interaction)
  - pressure < 10 & volume > 300 & velocity = 5 (3-way interaction)

- Surprisingly, no one had looked at interactions > 2-way before

96% of faults caused by single factor or 2-way interactions

65% of faults caused by single factor

Interesting, but that's just one kind of application!
These faults more complex than medical device software!!

Why?
Curves appear to be similar across a variety of application domains.
NASA distributed database

Note: initial testing but .... Fault profile better than medical devices!
MySQL

Cumulative percent of faults

Number of parameters involved in faults

- FDA
- Browser
- Server
- NASA DB
- MySQL
Wait, there’s more

Cumulative proportion of faults for \( t = 1..6 \)

- Number of factors involved in failures is small
- No failure involving more than 6 variables has been seen
How does this knowledge help?

Interaction rule: When all faults are triggered by the interaction of \( t \) or fewer variables, then testing all \( t \)-way combinations is \textit{pseudo-exhaustive} and can provide strong assurance.

It is nearly always impossible to exhaustively test all possible input combinations

The interaction rule says we don’t have to
(within reason; we still have value propagation issues, equivalence partitioning, timing issues, more complex interactions, \ldots )

Still no silver bullet.  Rats!
Let’s see how to use this in testing. A simple example:
How Many Tests Would It Take?

- There are 10 effects, each can be on or off
- All combinations is $2^{10} = 1,024$ tests
- What if our budget is too limited for these tests?
- Instead, let’s look at all 3-way interactions ...
Now How Many Would It Take?

- There are \( \binom{10}{3} = 120 \) 3-way interactions.
- Each triple has \( 2^3 = 8 \) settings: 000, 001, 010, 011, ...
- \( 120 \times 8 = 960 \) combinations
- Each test exercises many triples:

  0 1 1 0 0 0 0 1 1 0

OK, OK, what’s the **smallest** number of tests we need?
A covering array of 13 tests

All triples in only 13 tests, covering \( \binom{10}{3} 2^3 = 960 \) combinations

Each row is a test:

```
  0 0 0  0 0 0  0 0 0
  1 1 1  1 1 1  1 1 1
  1 1 1  0 1 0  0 0 1
  1 0 1  1 0 1  0 1 0
  1 0 0  0 1 1  1 0 0
  0 1 1  0 0 1  0 1 0
  0 0 1  0 1 0  1 1 0
  1 1 0  1 0 0  1 0 1
  0 0 0  1 1 1  0 1 1
  0 0 1  1 0 0  1 0 1
  0 1 0  1 1 0  0 1 0
  1 0 0  0 0 0  0 1 1
  0 1 0  0 0 1  1 1 0
```

Each column is a parameter:

- Developed 1990s
- Extends Design of Experiments concept
- NP hard problem but good algorithms now
New algorithms

- Smaller test sets faster, with a more advanced user interface
- First parallelized covering array algorithm
- More information per test

<table>
<thead>
<tr>
<th>T-Way</th>
<th>IPOG</th>
<th>ITCH (IBM)</th>
<th>Jenny (Open Source)</th>
<th>TConfig (U. of Ottawa)</th>
<th>TVG (Open Source)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>0.8</td>
<td>120</td>
<td>0.73</td>
<td>108</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
<td>0.36</td>
<td>2388</td>
<td>1020</td>
<td>413</td>
</tr>
<tr>
<td>4</td>
<td>1363</td>
<td>3.05</td>
<td>1484</td>
<td>5400</td>
<td>1536</td>
</tr>
<tr>
<td>5</td>
<td>4226</td>
<td>18s</td>
<td>NA</td>
<td>&gt;1 day</td>
<td>4580</td>
</tr>
<tr>
<td>6</td>
<td>10941</td>
<td>65.03</td>
<td>NA</td>
<td>&gt;1 day</td>
<td>11625</td>
</tr>
</tbody>
</table>

Traffic Collision Avoidance System (TCAS): $2^{7}3^{2}4^{1}10^{2}$

Times in seconds
Number of tests: proportional to $v^t \log n$ for $v$ values, $n$ variables, $t$-way interactions

- Good news: tests increase \textit{logarithmically with the number of parameters} => even very large test problems are OK (e.g., 200 parameters)

- Bad news: increase \textit{exponentially with interaction strength $t$} => select small number of representative values (but we always have to do this for any kind of testing)

However:
- coverage increases rapidly
- for 30 boolean variables
- 33 tests to cover all 3-way combinations
- but only 18 tests to cover about 95% of 3-way combinations
Testing inputs – combinations of variable values

Suppose we have a system with on-off switches.

Software must produce the right response for any combination of switch settings.
How do we test this?

34 switches $= 2^{34} = 1.7 \times 10^{10}$ possible inputs $= 17$ billion tests
What if no failure involves more than 3 switch settings interacting?

- 34 switches = 17 billion tests
- For 3-way interactions, need only 33 tests
- For 4-way interactions, need only 85 tests
33 tests for this (average) range of fault detection

85 tests for this (average) range of fault detection

That’s way better than 17 billion!
Research question – validate interaction rule?

• DOM is a World Wide Web Consortium standard for representing and interacting with browser objects
• NIST developed conformance tests for DOM
• Tests covered all possible combinations of discretized values, >36,000 tests
• Question: can we use the Interaction Rule to increase test effectiveness the way we claim?
Document Object Model Events

Combinatorial test set:

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>% of Orig.</th>
<th>Test Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>2</td>
<td>702</td>
<td>1.92%</td>
<td>202</td>
</tr>
<tr>
<td>3</td>
<td>1342</td>
<td>3.67%</td>
<td>786</td>
</tr>
<tr>
<td>4</td>
<td>1818</td>
<td>4.96%</td>
<td>437</td>
</tr>
<tr>
<td>5</td>
<td>2742</td>
<td>7.49%</td>
<td>908</td>
</tr>
<tr>
<td>6</td>
<td>4227</td>
<td>11.54%</td>
<td>1803</td>
</tr>
</tbody>
</table>

All failures found using < 5% of original exhaustive test set.
Case study example: Subway control system

Real-world experiment by grad students, Univ. of Texas at Dallas

Original testing by company: 2 months

Combinatorial testing by U. Texas students: 2 weeks

Result: approximately 3X as many bugs found, in 1/4 the time
=> 12X improvement
# Results

<table>
<thead>
<tr>
<th>Package</th>
<th>Original</th>
<th>Number of test cases</th>
<th>Number of bugs found</th>
<th>Did CT find all original bugs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package 1</td>
<td>98</td>
<td>2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Package 2</td>
<td>102</td>
<td>1</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Package 3</td>
<td>116</td>
<td>2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Package 4</td>
<td>122</td>
<td>2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Package 1</td>
<td>49</td>
<td>6</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Package 2</td>
<td>77</td>
<td>5</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Package 3</td>
<td>80</td>
<td>7</td>
<td>Miss 1</td>
<td></td>
</tr>
<tr>
<td>Package 4</td>
<td>90</td>
<td>4</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Modeling and Simulation application
Problem: failures of F-16 ventral fin

(Cunningham, Hagar, Holman, Lockheed Martin)

Figure 1. LANTIRN pod carriage on the F-16.
It’s not supposed to look like this:

Figure 2. F-16 ventral fin damage on flight with LANTIRN
Results

• Interactions causing problem included Mach points .95 and .97; multiple side-slip and rolling maneuvers
• Solution analysis tested interactions of Mach points, maneuvers, and multiple fin designs
• Problem could have been found much more efficiently and quickly
• Less expert time required

• Spreading use of combinatorial testing in the corporation:
  • Community of practice of 200 engineers
  • Tutorials and guidebooks
  • Internal web site and information forum
Event Sequence Testing

- Suppose we want to see if a system works correctly regardless of the order of events. How can this be done efficiently?
- Failure reports often say something like: 'failure occurred when A started if B is not already connected'.
- Can we produce compact tests such that all t-way sequences covered (possibly with interleaving events)?

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>connect range finder</td>
</tr>
<tr>
<td>$b$</td>
<td>connect telecom</td>
</tr>
<tr>
<td>$c$</td>
<td>connect satellite link</td>
</tr>
<tr>
<td>$d$</td>
<td>connect GPS</td>
</tr>
<tr>
<td>$e$</td>
<td>connect video</td>
</tr>
<tr>
<td>$f$</td>
<td>connect UAV</td>
</tr>
</tbody>
</table>
Sequence Covering Array

- With 6 events, all sequences = 6! = 720 tests
- Only 10 tests needed for all 3-way sequences, results even better for larger numbers of events
- Example: .*c.*f.*b.* covered. Any such 3-way seq covered.

<table>
<thead>
<tr>
<th>Test</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b c d e f</td>
</tr>
<tr>
<td>2</td>
<td>f e d c b a</td>
</tr>
<tr>
<td>3</td>
<td>d e f a b c</td>
</tr>
<tr>
<td>4</td>
<td>c b a f e d</td>
</tr>
<tr>
<td>5</td>
<td>b f a d c e</td>
</tr>
<tr>
<td>6</td>
<td>e c d a f b</td>
</tr>
<tr>
<td>7</td>
<td>a e f c b d</td>
</tr>
<tr>
<td>8</td>
<td>d b c f e a</td>
</tr>
<tr>
<td>9</td>
<td>c e a d b f</td>
</tr>
<tr>
<td>10</td>
<td>f b d a e c</td>
</tr>
</tbody>
</table>
Sequence Covering Array Properties

• 2-way sequences require only 2 tests (write in any order, reverse)
• For > 2-way, number of tests grows with log $n$, for $n$ events
• Simple greedy algorithm produces compact test set
• Application not previously described in CS or math literature

![Graph showing the number of tests for 2-way, 3-way, and 4-way sequences as a function of the number of events. The number of tests increases logarithmically with the number of events.]
## Combinatorial Coverage of Input Space

<table>
<thead>
<tr>
<th>Tests</th>
<th>Variables</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b c d</td>
<td>0 0 0 0</td>
<td>0 1 0</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable pairs</th>
<th>Variable-value combinations covered</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ac</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ad</td>
<td>00, 01, 11</td>
<td>.75</td>
</tr>
<tr>
<td>bc</td>
<td>00, 11</td>
<td>.50</td>
</tr>
<tr>
<td>bd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
<tr>
<td>cd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
</tbody>
</table>

100% coverage of $8/24 = 33\%$ of combinations
75% coverage of half of combinations
≥ 50% coverage of all combinations
79% total coverage of all 2-way combinations
Graphing Coverage Measurement

Area under curve = 79% total coverage

Bottom line: All combinations covered to at least 50%

Coverage for file

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.90</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.80</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.70</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.60</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.50</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.40</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.30</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.20</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.10</td>
<td>66/66 = 1.00</td>
</tr>
<tr>
<td>0.00</td>
<td>66/66 = 1.00</td>
</tr>
</tbody>
</table>

Total 2-way = 0.792

National Institute of Standards and Technology
What else does this chart show?

Tested combinations

Untested combinations
(look for problems here)
Spacecraft software example
82 variables, 7,489 tests, conventional test design (not covering arrays)
Advanced Encryption System (AES) tests
(S. Mekesis & D. Simos)

- Coverage measurement of known answer tests for various AES configurations (key size, mode)
- Tests used worldwide to verify conformance to AES standard
- Measurement identified weaknesses in tests for some configurations
Application to testing and assurance

• Measurable values with direct relevance to assurance
  • Theorem relating (static) combinatorial coverage with (dynamic) code coverage
• To answer the question: How thorough is this test set?
  We can provide a defensible answer

Examples:
• Fuzz testing (random values) – good for finding bugs and security vulnerabilities, but how do you know you’ve done enough?
• Contract monitoring – How do you justify testing has been sufficient? Identify duplication of effort?
Oracle-free testing

Some current approaches:

- **Fuzz testing** – send random values until system fails, then analyze memory dump, execution traces
- **Metamorphic testing** – e.g. \( \cos(x) = \cos(x+360) \), so compare outputs for both, with a difference indicating an error.
- **Partial test oracle** – e.g., insert element \( x \) in data structure \( S \), check \( x \in S \)

**New method** – using two-layer covering arrays

- requires only definition of equivalence classes
- we envision as part of the tool chain in development
Can this really work on practical code?

Experiment: TCAS code (standard set used to evaluate test methods)
- Small C module, 12 variables
- Seeded faults in 41 variants

- Results:

<table>
<thead>
<tr>
<th>Primary x secondary</th>
<th>#tests</th>
<th>total</th>
<th>faults detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-way x 3-way</td>
<td>285x8</td>
<td>2,280</td>
<td>6</td>
</tr>
<tr>
<td>4-way x 3-way</td>
<td>970x8</td>
<td>7,760</td>
<td>22</td>
</tr>
</tbody>
</table>

- More than half of faults detected
- Large number of tests -> but fully automated, no human intervention
- We envision this type of checking as part of the build process; can be used in parallel with static analysis, type checking
Available Tools

- **Covering array generator** – basic tool for test input or configurations;
- **Input modeling tool** – design inputs to covering array generator using classification tree editor; useful for partitioning input variable values
- **Fault location tool** – identify combinations and sections of code likely to cause problem
- **Sequence covering array generator** – new concept; applies combinatorial methods to event sequence testing
- **Combinatorial coverage measurement** – detailed analysis of combination coverage; automated generation of supplemental tests; helpful for integrating c/t with existing test methods
ACTS Users > 2,000 organizations

- Defense
- Finance
- Information Technology
- Telecom
- Other sectors (Airlines, Defense/govt, Electronics, IT, Language, Med/pharma, Retail/sales, Transportation)
Software failures are triggered by a small number of factors interacting – 1 to 6 in known cases.

Therefore covering all t-way combinations, for small t, is pseudo-exhaustive and provides strong assurance.

Strong t-way interaction coverage can be provided using covering arrays.

Combinatorial testing is practical today using existing tools for real-world software.

Combinatorial methods have been shown to provide significant cost savings with improved test coverage, and proportional cost savings increases with the size and complexity of problem.
Bottom line: Significant cost savings and better testing shown in an extensive variety of application domains

Please contact us if you’re interested!

Rick Kuhn
kuhn@nist.gov

Raghu Kacker
raghu.kacker@nist.gov

http://csrc.nist.gov/acts
Applications

Software testing – primary application of these methods
• functionality testing and security vulnerabilities
• approx 2/3 of vulnerabilities from implementation faults

Modeling and simulation – ensure coverage of complex cases
• measure coverage of traditional Monte Carlo sim
• faster coverage of input space than randomized input

Performance tuning – determine most effective combination of configuration settings among a large set of factors

>> systems with a large number of factors that interact <<
Why combinatorial testing? - examples

- Cooperative R&D Agreement w/ Lockheed Martin
  - 2.5 year study, 8 Lockheed Martin pilot projects in aerospace software
  - Results: save 20% of test costs; increase test coverage by 20% to 50%

- Rockwell Collins applied NIST method and tools on testing to FAA life-critical standards
  - Found practical for industrial use
  - Enormous cost reduction

Average software: testing typically 50% of total dev cost
Civil aviation: testing >85% of total dev cost (NASA rpt)