Introducing Combinatorial Testing in Large Organizations

Rick Kuhn
National Institute of Standards and Technology
Gaithersburg, MD

ASTQB Software Testing Conference
March 26, 2014
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 goals – reduce testing cost, improve cost-benefit ratio for testing
What good is combinatorial testing?

• Joint project w/ Lockheed Martin
• 2.5 year study, 8 Lockheed Martin pilot projects in aerospace software
• Results: “Our initial estimate is that this method supported by the technology can save up to 20% of test planning/design costs if done early on a program while increasing test coverage by 20% to 50%”.

• We will discuss this and other examples
How did we get here?

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

Interaction faults: e.g., failure occurs if
  pressure < 10 && volume>300
  (interaction between 2 factors)

Example from FDA failure analysis:

Failure when “altitude adjustment set on 0 meters and total flow volume set at delivery rate of less than 2.2 liters per minute.”
What does a 2-way fault look like in code?

How does an interaction fault manifest itself in code?
Example: `altitude_adj == 0 && volume < 2.2` (2-way interaction)

```java
if (altitude_adj == 0) {
    // do something
    if (volume < 2.2) { faulty code! BOOM! }
    else { good code, no problem}
} else {
    // do something else
}
```

A test with `altitude_adj == 0` and `volume = 1` would find this.

Again, ~ 90% of the FDA failures were 2-way or 1-way
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 beyond 2-way before
• The most complex medical device failure reported required 4-way interaction to trigger.

Interesting, but that's just one kind of application!
Examples from the National Vulnerability Database

Single variable, 1-way interaction
example: Heap-based buffer overflow in the SFTP protocol handler for Panic Transmit … allows remote attackers to execute arbitrary code via a long ftps:// URL.

2-way interaction
example: single character search string in conjunction with a single character replacement string, which causes an "off by one overflow"

3-way interaction
example: Directory traversal vulnerability when register_globals is enabled and magic_quotes is disabled and .. (dot dot) in the page parameter
What about other applications?

**Server** (green)

These faults more complex than medical device software!!

Why?
Others?

Browser *(magenta)*

% detected

Number of factors involved in faults
Still more?

NASA Goddard distributed database (light blue)

![Graph showing the number of factors involved in faults vs. % detected.](chart.png)
Even more?

FAA Traffic Collision Avoidance System module (seeded errors) (purple)

Number of factors involved in faults

% detected

0 10 20 30 40 50 60 70 80 90 100

1 2 3 4 5 6
Finally

Network security (Bell, 2006) (orange)

Curves appear to be similar across a variety of application domains.
Number of factors involved in faults

- New algorithms make it practical to test these combinations
- We test large number of combinations with very few tests

- **Number of factors involved in failures is small**
Interaction Rule

- Refers to how many parameters are involved in faults:
  
  **Interaction rule**: most failures are triggered by one or two parameters, and progressively fewer by three, four, or more parameters, and the maximum interaction degree is small.

- Maximum interactions for fault triggering was 6
- Popular “pairwise testing” not enough
- More empirical work needed
- Reasonable evidence that maximum interaction strength for fault triggering is relatively small

How does it help me to know this?
How does this knowledge help?

If all faults are triggered by the interaction of $t$ or fewer variables, then testing all $t$-way combinations can provide strong assurance.

(taking into account: value propagation issues, equivalence partitioning, timing issues, more complex interactions, . . . )

Still no silver bullet. Rats!
Let’s see how to use this knowledge 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.
- Naively $120 \times 2^3 = 960$ tests.
- Since we can pack 3 triples into each test, we need no more than 320 tests.
- Each test exercises many triples:

  \[
  \begin{array}{cccccccc}
  0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
  \end{array}
  \]

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

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

- Developed 1990s
- Extends Design of Experiments concept
- Difficult mathematically but good algorithms now
A larger example

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 = $1.7 \times 10^{10}$ tests
What if we knew no failure involves more than 3 switch settings interacting?

- 34 switches = $2^{34} = 1.7 \times 10^{10}$ possible inputs = $1.7 \times 10^{10}$ tests
- If only 3-way interactions, need only 33 tests
- For 4-way interactions, need only 85 tests
33 tests for this range of fault detection

85 tests for this range of fault detection

That’s way better than 17 billion!
Two ways of using combinatorial testing

Use combinations here or here

<table>
<thead>
<tr>
<th>Test case</th>
<th>OS</th>
<th>CPU</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows</td>
<td>Intel</td>
<td>IPv4</td>
</tr>
<tr>
<td>2</td>
<td>Windows</td>
<td>AMD</td>
<td>IPv6</td>
</tr>
<tr>
<td>3</td>
<td>Linux</td>
<td>Intel</td>
<td>IPv6</td>
</tr>
<tr>
<td>4</td>
<td>Linux</td>
<td>AMD</td>
<td>IPv4</td>
</tr>
</tbody>
</table>

System under test

Configuration
Testing Configurations

• Example: app must run on any configuration of OS, browser, protocol, CPU, and DBMS

• Very effective for interoperability testing, being used by NIST for DoD Android phone testing

<table>
<thead>
<tr>
<th>Test</th>
<th>OS</th>
<th>Browser</th>
<th>Protocol</th>
<th>CPU</th>
<th>DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>10</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
Testing Smartphone Configurations

Some Android configuration options:

```c
int HARDKEYBOARDHIDDEN_NO;
int HARDKEYBOARDHIDDEN_UNDEFINED;
int HARDKEYBOARDHIDDEN_YES;
int KEYBOARDHIDDEN_NO;
int KEYBOARDHIDDEN_UNDEFINED;
int KEYBOARDHIDDEN_YES;
int KEYBOARD_12KEY;
int KEYBOARD_NOKEYS;
int KEYBOARD_QWERTY;
int KEYBOARD_UNDEFINED;
int NAVIGATIONHIDDEN_NO;
int NAVIGATIONHIDDEN_UNDEFINED;
int NAVIGATIONHIDDEN_YES;
int NAVIGATION_DPAD;
int NAVIGATION_NONAV;
int NAVIGATION_TRACKBALL;
int NAVIGATION_UNDEFINED;
int NAVIGATION_WHEEL;
```
## Configuration option values

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Values</th>
<th># Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARDKEYBOARDHIDDEN</td>
<td>NO, UNDEFINED, YES</td>
<td>3</td>
</tr>
<tr>
<td>KEYBOARDHIDDEN</td>
<td>NO, UNDEFINED, YES</td>
<td>3</td>
</tr>
<tr>
<td>KEYBOARD</td>
<td>12KEY, NOKEYS, QWERTY, UNDEFINED</td>
<td>4</td>
</tr>
<tr>
<td>NAVIGATIONHIDDEN</td>
<td>NO, UNDEFINED, YES</td>
<td>3</td>
</tr>
<tr>
<td>NAVIGATION</td>
<td>DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL</td>
<td>5</td>
</tr>
<tr>
<td>ORIENTATION</td>
<td>LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED</td>
<td>4</td>
</tr>
<tr>
<td>SCREENLAYOUT_LONG</td>
<td>MASK, NO, UNDEFINED, YES</td>
<td>4</td>
</tr>
<tr>
<td>SCREENLAYOUT_SIZE</td>
<td>LARGE, MASK, NORMAL, SMALL, UNDEFINED</td>
<td>5</td>
</tr>
<tr>
<td>TOUCHSCREEN</td>
<td>FINGER, NOTOUCH, STYLUS, UNDEFINED</td>
<td>4</td>
</tr>
</tbody>
</table>

**Total possible configurations:**

$$3 \times 3 \times 4 \times 3 \times 5 \times 4 \times 4 \times 5 \times 4 = 172,800$$
Number of configurations generated for $t$-way interaction testing, $t = 2..6$

<table>
<thead>
<tr>
<th>$t$</th>
<th># Configs</th>
<th>% of Exhaustive</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>29</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>137</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>625</td>
<td>0.4</td>
</tr>
<tr>
<td>5</td>
<td>2532</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>9168</td>
<td>5.3</td>
</tr>
</tbody>
</table>
What tools are available?

- **Covering array generator** – basic tool for test input or configurations;

- **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

- **Domain/application specific tools:**
  - Access control policy tester
  - .NET config file generator
## 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></td>
<td>400</td>
<td>0.36</td>
<td>2388</td>
<td>1020</td>
<td>413</td>
</tr>
<tr>
<td></td>
<td>1363</td>
<td>3.05</td>
<td>1484</td>
<td>5400</td>
<td>1536</td>
</tr>
<tr>
<td>4</td>
<td>4226</td>
<td>18s</td>
<td>NA</td>
<td>&gt;1 day</td>
<td>4580</td>
</tr>
<tr>
<td>5</td>
<td>10941</td>
<td>65.03</td>
<td>NA</td>
<td>&gt;1 day</td>
<td>11625</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1070048</td>
</tr>
</tbody>
</table>

Traffic Collision Avoidance System (TCAS): $2^73^24^110^2$

Times in seconds
ACTS - Defining a new system
Variable interaction strength

![New System Form](image)

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cur_Vertical_Sep</td>
<td>4</td>
</tr>
<tr>
<td>High_Confidence</td>
<td></td>
</tr>
<tr>
<td>Two_of_Three_Reports</td>
<td></td>
</tr>
<tr>
<td>Own_Tracked_Alt</td>
<td></td>
</tr>
<tr>
<td>Other_Track_Alt</td>
<td></td>
</tr>
<tr>
<td>Own_Tracked_Alt_Rate</td>
<td></td>
</tr>
<tr>
<td>Alt_Layer_Value</td>
<td></td>
</tr>
<tr>
<td>Up_Separation</td>
<td></td>
</tr>
<tr>
<td>Down_Separation</td>
<td></td>
</tr>
<tr>
<td>Other_RAC</td>
<td></td>
</tr>
<tr>
<td>Other_Capability</td>
<td></td>
</tr>
<tr>
<td>Climb_Inhibit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter Names</th>
<th>Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cur_Vertical_Sep, High_Confidence, Two_of_Three_Reports</td>
<td>2</td>
</tr>
<tr>
<td>Alt_Layer_Value, Up_Separation, Down_Separation</td>
<td>3</td>
</tr>
</tbody>
</table>
Constraints
Covering array output

<table>
<thead>
<tr>
<th>Test Result</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR_V...</td>
<td>HIGH...</td>
</tr>
<tr>
<td>1</td>
<td>299</td>
</tr>
<tr>
<td>2</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>601</td>
</tr>
<tr>
<td>4</td>
<td>299</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
</tr>
<tr>
<td>6</td>
<td>601</td>
</tr>
<tr>
<td>7</td>
<td>299</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
</tr>
<tr>
<td>9</td>
<td>601</td>
</tr>
<tr>
<td>10</td>
<td>299</td>
</tr>
<tr>
<td>11</td>
<td>300</td>
</tr>
<tr>
<td>12</td>
<td>601</td>
</tr>
<tr>
<td>13</td>
<td>299</td>
</tr>
<tr>
<td>14</td>
<td>300</td>
</tr>
<tr>
<td>15</td>
<td>601</td>
</tr>
<tr>
<td>16</td>
<td>299</td>
</tr>
<tr>
<td>17</td>
<td>300</td>
</tr>
<tr>
<td>18</td>
<td>601</td>
</tr>
<tr>
<td>19</td>
<td>299</td>
</tr>
<tr>
<td>20</td>
<td>300</td>
</tr>
<tr>
<td>21</td>
<td>601</td>
</tr>
<tr>
<td>22</td>
<td>299</td>
</tr>
<tr>
<td>23</td>
<td>300</td>
</tr>
<tr>
<td>24</td>
<td>601</td>
</tr>
<tr>
<td>25</td>
<td>299</td>
</tr>
<tr>
<td>26</td>
<td>300</td>
</tr>
<tr>
<td>27</td>
<td>601</td>
</tr>
<tr>
<td>28</td>
<td>299</td>
</tr>
<tr>
<td>29</td>
<td>300</td>
</tr>
<tr>
<td>30</td>
<td>601</td>
</tr>
<tr>
<td>31</td>
<td>299</td>
</tr>
<tr>
<td>32</td>
<td>300</td>
</tr>
</tbody>
</table>
### Output options

#### Mappable values

<table>
<thead>
<tr>
<th>Degree of interaction coverage: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parameters: 12</td>
</tr>
<tr>
<td>Number of tests: 100</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>011</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>201</td>
<td>010</td>
<td>102</td>
<td>022</td>
<td>210</td>
<td>010</td>
<td>012</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>101</td>
<td>013</td>
<td>031</td>
<td>010</td>
<td>110</td>
<td>010</td>
<td>421</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>111</td>
<td>001</td>
<td>010</td>
<td>010</td>
<td>421</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>421</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>210</td>
<td>110</td>
<td>010</td>
<td>500</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>120</td>
<td>600</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>101</td>
<td>010</td>
<td>307</td>
<td>011</td>
<td>110</td>
<td>010</td>
<td>810</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>201</td>
<td>110</td>
<td>010</td>
<td>810</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>010</td>
<td>921</td>
<td>111</td>
<td>110</td>
<td>102</td>
<td>101</td>
<td>010</td>
<td>101</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>010</td>
<td>210</td>
<td>101</td>
<td>010</td>
<td>101</td>
<td>010</td>
<td>101</td>
<td>010</td>
<td>101</td>
<td>010</td>
<td>101</td>
</tr>
</tbody>
</table>

Etc.

#### Human readable

Degree of interaction coverage: 2
Number of parameters: 12
Maximum number of values per parameter: 10
Number of configurations: 100

Configuration #1:

1 = Cur_Vertical_Sep=299
2 = High_Confidence=true
3 = Two_of_Three_Reports=true
4 = Own_Tracked_Alt=1
5 = Other_Tracked_Alt=1
6 = Own_Tracked_Alt_Rate=600
7 = Alt_Layer_Value=0
8 = Up_Separation=0
9 = Down_Separation=0
10 = Other_RAC=NO_INTENT
11 = Other_Capability=TCAS_CA
12 = Climb_Inhibit=true
How many tests are needed?

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

- Thus:
  - Tests increase \textit{exponentially} with interaction strength $t$
  - But \textit{logarithmically} with the number of parameters

- Example: suppose we want all 4-way combinations of $n$
  parameters, 5 values each:
How do we automate checking correctness of output?

• Creating test data is the easy part!

• How do we check that the code worked correctly on the test input?
  
  • **Crash testing** server or other code to ensure it does not crash for any test input (like ‘fuzz testing’)
    - Easy but limited value

  • **Built-in self test with embedded assertions** – incorporate assertions in code to check critical states at different points in the code, or print out important values during execution

  • **Full scale model-checking** using mathematical model of system and model checker to generate expected results for each input - expensive but tractable
Crash Testing

• Like “fuzz testing” - send packets or other input to application, watch for crashes

• Unlike fuzz testing, input is non-random; cover all t-way combinations

• May be more efficient - random input generation requires several times as many tests to cover the t-way combinations in a covering array

Limited utility, but can detect high-risk problems such as:

- buffer overflows
- server crashes
Ratio of Random/Combinatorial Test Set Required to Provide t-way Coverage

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Values per variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.50-5.00</td>
<td>nval=10</td>
</tr>
<tr>
<td>4.00-4.50</td>
<td>nval=6</td>
</tr>
<tr>
<td>3.50-4.00</td>
<td></td>
</tr>
<tr>
<td>3.00-3.50</td>
<td></td>
</tr>
<tr>
<td>2.50-3.00</td>
<td></td>
</tr>
<tr>
<td>2.00-2.50</td>
<td></td>
</tr>
<tr>
<td>1.50-2.00</td>
<td></td>
</tr>
<tr>
<td>1.00-1.50</td>
<td></td>
</tr>
<tr>
<td>0.50-1.00</td>
<td></td>
</tr>
<tr>
<td>0.00-0.50</td>
<td></td>
</tr>
</tbody>
</table>

Graph showing the ratio of random/combinatorial test set required to provide t-way coverage for different combinations of interactions and values per variable.
Embedded Assertions

Simple example:
assert( x != 0);   // ensure divisor is not zero

Or pre and post-conditions:
/requires amount >= 0;

/ensures balance == \old(balance) - amount &&
\result == balance;
Embedded Assertions

Assertions check properties of expected result:

```
ensures balance  == \old(balance) - amount 
&& \result == balance;
```

• Reasonable assurance that code works correctly across the range of expected inputs

• May identify problems with handling unanticipated inputs

• Example: Smart card testing
  • Used Java Modeling Language (JML) assertions
  • Detected 80% to 90% of flaws
Using model checking to produce tests

The system can never get in this state!

Yes it can, and here's how …

- Model-checker test production: if assertion is not true, then a counterexample is generated.

- This can be converted to a test case.

Black & Ammann, 1999
Testing inputs

- Traffic Collision Avoidance System (TCAS) module
  - Used in previous testing research
  - 41 versions seeded with errors
  - 12 variables: 7 boolean, two 3-value, one 4-value, two 10-value
  - All flaws found with 5-way coverage
  - Thousands of tests - generated by model checker in a few minutes
# Tests generated

<table>
<thead>
<tr>
<th>$t$</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way:</td>
<td>156</td>
</tr>
<tr>
<td>3-way:</td>
<td>461</td>
</tr>
<tr>
<td>4-way:</td>
<td>1,450</td>
</tr>
<tr>
<td>5-way:</td>
<td>4,309</td>
</tr>
<tr>
<td>6-way:</td>
<td>11,094</td>
</tr>
</tbody>
</table>

![Bar chart showing the number of tests generated for different values of $t$.](chart.png)
Results

- Roughly consistent with data on large systems
- But errors harder to detect than real-world examples

Bottom line for model checking based combinatorial testing: Requires more technical skill but can be highly effective
How is this stuff useful in the real world??
Example 1: Document Object Model Events

- 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

**Original test set:**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Param.</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Blur</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>Click</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>Change</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>dblClick</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>DOMActivate</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>DOMAttrModified</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>DOMCharacterDataModified</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>DOMElementNameChanged</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>DOMFocusIn</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>DOMFocusOut</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>DOMNodeInserted</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMNodeInsertedIntoDocument</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMNodeRemoved</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMNodeRemovedFromDocument</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMSubTreeModified</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>Error</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>KeyDown</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>KeyUp</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Load</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>MouseDown</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseMove</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseOut</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseOver</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseUp</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseWheel</td>
<td>14</td>
<td>1024</td>
</tr>
<tr>
<td>Reset</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Resize</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>Scroll</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>Select</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Submit</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>TextInput</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Unload</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Wheel</td>
<td>15</td>
<td>4096</td>
</tr>
</tbody>
</table>

**Total Tests:** 36626

*Exhaustive testing of equivalence class values*
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
Example 2: Problem: unknown factors causing failures of F-16 ventral fin

LANTIRN = Low Altitude Navigation & Targeting Infrared for Night

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
Can the problem factors be found efficiently?

Original solution: Lockheed Martin engineers spent many months with wind tunnel tests and expert analysis to consider interactions that could cause the problem.

Combinatorial testing solution: modeling and simulation using ACTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft</td>
<td>15, 40</td>
</tr>
<tr>
<td>Altitude</td>
<td>5k, 10k, 15k, 20k, 30k, 40k, 50k</td>
</tr>
<tr>
<td>Maneuver</td>
<td>hi-speed throttle, slow accel/dwell, L/R 5 deg side slip, L/R 360 roll, R/L 5 deg side slip, Med accel/dwell, R-L-R-L banking, Hi-speed to Low, 360 nose roll</td>
</tr>
<tr>
<td>Mach (100th)</td>
<td>40, 50, 60, 70, 80, 90, 100, 110, 120</td>
</tr>
</tbody>
</table>
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
Example 3: Laptop application testing

Problem: connect many peripherals, order of connection may affect application.
# Connection Sequences

<table>
<thead>
<tr>
<th></th>
<th>Boot</th>
<th>P-1 (USB-RIGHT)</th>
<th>P-2 (USB-BACK)</th>
<th>P-3 (USB-LEFT)</th>
<th>P-4</th>
<th>P-5</th>
<th>App</th>
<th>Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Boot</td>
<td>App</td>
<td>Scan</td>
<td>P-5</td>
<td>P-4</td>
<td>P-5</td>
<td>App</td>
<td>Scan</td>
</tr>
<tr>
<td>2</td>
<td>Boot</td>
<td>P-3 (USB-RIGHT)</td>
<td>P-2 (USB-LEFT)</td>
<td>P-1 (USB-BACK)</td>
<td>App</td>
<td>Scan</td>
<td>P-5</td>
<td>P-4</td>
</tr>
<tr>
<td>3</td>
<td>Boot</td>
<td></td>
<td></td>
<td></td>
<td>P-3 (USB-RIGHT)</td>
<td>P-2 (USB-BACK)</td>
<td>P-1 (USB-BACK)</td>
<td></td>
</tr>
</tbody>
</table>

etc...
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 flow meter</td>
</tr>
<tr>
<td>b</td>
<td>connect pressure gauge</td>
</tr>
<tr>
<td>c</td>
<td>connect satellite link</td>
</tr>
<tr>
<td>d</td>
<td>connect pressure readout</td>
</tr>
<tr>
<td>e</td>
<td>start comm link</td>
</tr>
<tr>
<td>f</td>
<td>boot system</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 events in any order, then reverse)
- For > 2-way, number of tests grows with $\log n$, for $n$ events
- Simple greedy algorithm produces compact test set
- Not previously described in CS or math literature

![Graph showing tests versus number of events for 2-way, 3-way, and 4-way sequences. The number of tests increases with the number of events for each case. The graph includes a legend indicating 2-way, 3-way, and 4-way sequences.]
Example 4: Existing Test Sets

• Will this method disrupt my test process?
• What if I already have a large set of tests? Does this approach add anything?

• NASA spacecraft software test set, approx 7,500 tests
• Does it already provide 2-way, 3-way, 4-way coverage?
## Measuring Combinatorial Coverage

<table>
<thead>
<tr>
<th>Tests</th>
<th>Variables</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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 33% of combinations  
75% coverage of half of combinations  
50% coverage of 16% of combinations
Graphing Coverage Measurement

100% coverage of 33% of combinations
75% coverage of half of combinations
50% coverage of 16% of combinations

Bottom line:
All combinations covered to at least 50%
Adding a test

Coverage after adding test [1,1,0,1]
Adding another test

Coverage after adding test [1,0,1,1]
Additional test completes coverage

Coverage after adding test [1, 0, 1, 0]
All combinations covered to 100% level, so this is a covering array.
Combinatorial Coverage Measurement

Number of tests: 7489
Number of parameters: 82

Detection chart with Coverage for 3-way and 2-way combinations.
Lessons Learned and Needs

• Education and training materials – tutorial, textbook
• Greater availability of tools to support combinatorial testing – open sourcing 5 tools
• Modify approaches to using combinatorial testing – integrating combinatorial testing with other test practices; ability to adopt CT partially or gradually – measurement tool
• Incorporate combinatorial methods into DoD guidance and industry standards; develop a community of practice
  – We would be happy to work with ASTQB and others!
Where do we go next?

• “Internet of things” – testing problem enormous
  • Vast number of interacting components
  • Combinatorial testing is a natural fit
• Cyber-physical systems
  • Safety aspects
  • Another natural fit with combinatorial methods
• Test development environment
  • Define the data model – critical for testing
  • Project with CMU
• Will be open source with all other tools
Please contact us if you are interested.

Rick Kuhn
kuhn@nist.gov

Raghu Kacker
raghu.kacker@nist.gov

http://csrv.nist.gov/acts
BACKUP SLIDES FOR ADDITIONAL DISCUSSION
Background: Interaction Testing and Design of Experiments (DOE)

Complete sequence of steps to ensure appropriate data will be obtained, which permit objective analysis that lead to valid conclusions about cause-effect systems

Objectives stated ahead of time

Opposed to observational studies of nature, society …

Minimal expense of time and cost

Multi-factor, not one-factor-at-a-time

DOE implies design and associated data analysis

Validity of inferences depends on design

A DOE plan can be expressed as matrix

Rows: tests, columns: variables, entries: test values or treatment allocations to experimental units
Where did these ideas come from?

Scottish physician James Lind determined cure of scurvy

Ship HM Bark Salisbury in 1747

12 sailors “were as similar as I could have them”

6 treatments 2 sailors for each – cider, sulfuric acid, vinegar, seawater, orange/lemon juice, barley water

Principles used (blocking, replication, randomization)

Did not consider interactions, but otherwise used basic Design of Experiments principles
Father of DOE:
R A Fisher, 1890-1962, British geneticist

Key features of DoE
- Blocking
- Replication
- Randomization
- Orthogonal arrays to test interactions between factors

<table>
<thead>
<tr>
<th>Test</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Each combination occurs **same number** of times, usually once.

Example: P1, P2 = 1,2
Four eras of evolution of DOE

Era 1:(1920’s …): Beginning in agricultural then animal science, clinical trials, medicine

Era 2:(1940’s …): Industrial productivity – new field, same basics

Era 3:(1980’s …): Designing robust products – new field, same basics

Then things begin to change . . .

Era 4:(2000’s …): Combinatorial Testing of Software
Agriculture and biological investigations-1

System under investigation
- Crop growing, effectiveness of drugs or other treatments
- Mechanistic (cause-effect) process; predictability limited

Variable Types
- Primary test factors (farmer can adjust, drugs)
  Held constant
- Background factors (controlled in experiment, not in field)
- Uncontrolled factors (Fisher’s genius idea; randomization)

Numbers of treatments
- Generally less than 10

Objectives: compare treatments to find better
- Treatments: qualitative or discrete levels of continuous
Agriculture and biological investigations-2

Scope of investigation:

Treatments actually tested, direction for improvement

Key principles

**Replication:** minimize experimental error (which may be large) replicate each test run; averages less variable than raw data

**Randomization:** allocate treatments to experimental units at random; then error treated as draws from normal distribution

**Blocking** (homogeneous grouping of units): systematic effects of background factors eliminated from comparisons

Designs: Allocate treatments to experimental units

Randomized Block designs, Balanced Incomplete Block Designs, Partially balanced Incomplete Block Designs
Robust products-1

System under investigation

- Design of product (or design of manufacturing process)

Variable Types

- Control Factors: levels can be adjusted
- Noise factors: surrogates for down stream conditions

AT&T-BL 1985 experiment with 17 factors was large

Objectives:

- Find settings for robust product performance: product lifespan under different operating conditions across different units
- Environmental variable, deterioration, manufacturing variation
Robust products-2

Scope of investigation:

Optimum levels of control factors at which variation from noise factors is minimum

Key principles

Variation from noise factors
Efficiency in testing; accommodate constraints

Designs: Based on Orthogonal arrays (OAs)
Taguchi designs (balanced 2-way covering arrays)

This stuff is great! Let’s use it for software!
Orthogonal Arrays for Software Interaction Testing

Functional (black-box) testing
- Hardware-software systems
- Identify single and 2-way combination faults

Early papers
- Taguchi followers (mid 1980’s)
- Mandl (1985) Compiler testing
- Brownlie et al (1992) AT&T

Generation of test suites using OAs
- OATS (Phadke, AT&T-BL)
Interaction Failure Internals

How does an interaction fault manifest itself in code?

Example: `altitude_adj == 0 && volume < 2.2` (2-way interaction)

```java
if (altitude_adj == 0) {
    // do something
    if (volume < 2.2) { faulty code! BOOM! }
    else { good code, no problem}
} else {
    // do something else
}
```

A test that included `altitude_adj == 0` and `volume = 1` would trigger this failure
What’s different about software?

Traditional DoE
- Continuous variable results
- Small number of parameters
- Interactions typically increase or decrease output variable

DoE for Software
- Binary result (pass or fail)
- Large number of parameters
- Interactions affect path through program

Does this difference make any difference?
So how did testing interactions work in practice for software?

- Pairwise testing commonly applied to software
- Intuition: some problems only occur as the result of an interaction between parameters/components
- Tests all pairs (2-way combinations) of variable values
- Pairwise testing finds about 50% to 90% of flaws

90% of flaws! Sounds pretty good!
Model checking example

-- specification for a portion of tcas - altitude separation.
-- The corresponding C code is originally from Siemens Corp. Research
-- Vadim Okun 02/2002
MODULE main
VAR
   Cur_Vertical_Sep : { 299, 300, 601 };
   High_Confidence : boolean;
...
init(alt_sep) := START_
next(alt_sep) := case
   enabled & (intent_not_known | !tcas_equipped) : case
      need_upward_RA & need_downward_RA : UNRESOLVED;
      need_upward_RA : UPWARD_RA;
      need_downward_RA : DOWNWARD_RA;
      1 : UNRESOLVED;
   esac;
   1 : UNRESOLVED;
   esac;
...
SPEC AG ((enabled & (intent_not_known | !tcas_equipped) &
!need_downward_RA & need_upward_RA) -> AX (alt_sep = UPWARD_RA))
-- "FOR ALL executions,
-- IF enabled & (intent_not_known ....
-- THEN in the next state alt_sep = UPWARD_RA"
Computation Tree Logic

The usual logic operators, plus temporal:

- **All**: $\forall \phi$ - $\phi$ holds on all paths starting from the current state.
- **Exists**: $\exists \phi$ - $\phi$ holds on some paths starting from the current state.
- **Globally**: $\forall \phi$ - $\phi$ has to hold on the entire subsequent path.
- **Finally**: $\exists \phi$ - $\phi$ eventually has to hold
- **Next**: $\forall \phi$ - $\phi$ has to hold at the next state

[others not listed]

 SPEC AG ((enabled & (intent_not_known | !tcas_equipped) & !need_downward_RA & need_upward_RA) -> AX (alt_sep = UPWARD_RA))

"FOR ALL executions,
IF enabled & (intent_not_known ....
THEN in the next state alt_sep = UPWARD_RA"
What is the most effective way to integrate combinatorial testing with model checking?

• Given $AG \left( P \rightarrow AX \left( R \right) \right)$
  “for all paths, in every state,
   if P then in the next state, R holds”

• For k-way variable combinations, $v_1 \& v_2 \& \ldots \& v_k$

• $v_i$ abbreviates “var1 = val1”

• Now combine this constraint with assertion to produce counterexamples. Some possibilities:
  1. $AG\left(v_1 \& v_2 \& \ldots \& v_k \& P \rightarrow AX \left( !\left( R \right) \right) \right)$
  2. $AG\left(v_1 \& v_2 \& \ldots \& v_k \rightarrow AX \left( !\left( 1 \right) \right) \right)$
  3. $AG\left(v_1 \& v_2 \& \ldots \& v_k \rightarrow AX \left( !\left( R \right) \right) \right)$
What happens with these assertions?

1. \[ AG(v_1 \land v_2 \land \ldots \land v_k \land P \rightarrow AX \neg(R) ) \]

P may have a negation of one of the \( v_i \), so we get
\[ 0 \rightarrow AX \neg(R) \]
always true, so no counterexample, no test.
This is too restrictive!

1. \[ AG(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg(1) ) \]

The model checker makes non-deterministic choices for variables not in \( v_1..v_k \), so all R values may not be covered by a counterexample.
This is too loose!

2. \[ AG(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg(R) ) \]

Forces production of a counterexample for each R.
This is just right!
Modeling & Simulation

1. Aerospace - Lockheed Martin – analyze structural failures for aircraft design

2. Network defense/offense operations - NIST – analyze network configuration for vulnerability to deadlock
Example 3: Network Simulation

- “Simured” network simulator
  - Kernel of ~5,000 lines of C++ (not including GUI)

- Objective: detect configurations that can produce deadlock:
  - Prevent connectivity loss when changing network
  - Attacks that could lock up network

- Compare effectiveness of random vs. combinatorial inputs

- Deadlock combinations discovered

- Crashes in >6% of tests w/ valid values (Win32 version only)
## Simulation Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DIMENSIONS</td>
<td>1,2,4,6,8</td>
</tr>
<tr>
<td>2 NODOSDIM</td>
<td>2,4,6</td>
</tr>
<tr>
<td>3 NUMVIRT</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>4 NUMVIRTINJ</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>5 NUMVIRTEJE</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>6 LONBUFFER</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>7 NUMDIR</td>
<td>1,2</td>
</tr>
<tr>
<td>8 FORWARDING</td>
<td>0,1</td>
</tr>
<tr>
<td>9 PHYSICAL</td>
<td>true, false</td>
</tr>
<tr>
<td>10 ROUTING</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>11 DELFIFO</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>12 DELCROSS</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>13 DELCHANNEL</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>14 DELSWITCH</td>
<td>1,2,4,6</td>
</tr>
</tbody>
</table>

5x3x4x4x4x4x2x2x2x4x4x4x4 = 31,457,280 configurations

Are any of them dangerous?

If so, how many?

Which ones?
Network Deadlock Detection

Deadlocks Detected: combinatorial

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>500 pkts</th>
<th>1000 pkts</th>
<th>2000 pkts</th>
<th>4000 pkts</th>
<th>8000 pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>752</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

Average Deadlocks Detected: random

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>500 pkts</th>
<th>1000 pkts</th>
<th>2000 pkts</th>
<th>4000 pkts</th>
<th>8000 pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>0.63</td>
<td>0.25</td>
<td>0.75</td>
<td>0.50</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>752</td>
<td>10.13</td>
<td>11.75</td>
<td>10.38</td>
<td>13</td>
<td>13.25</td>
</tr>
</tbody>
</table>
Detected 14 configurations that can cause deadlock:

\[
\frac{14}{31,457,280} = 4.4 \times 10^{-7}
\]

Combinatorial testing found more deadlocks than random, including some that might never have been found with random testing.

Why do this testing? Risks:

- accidental deadlock configuration: low
- deadlock config discovered by attacker: much higher

(because they are looking for it)
Example 4: Buffer Overflows

- Empirical data from the National Vulnerability Database
  - Investigated > 3,000 denial-of-service vulnerabilities reported in the NIST NVD for period of 10/06 – 3/07
  - Vulnerabilities triggered by:
    - Single variable – 94.7%
      example: *Heap-based buffer overflow in the SFTP protocol handler for Panic Transmit ... allows remote attackers to execute arbitrary code via a long ftps:// URL.*
    - 2-way interaction – 4.9%
      example: *single character search string in conjunction with a single character replacement string, which causes an "off by one overflow"*
    - 3-way interaction – 0.4%
      example: *Directory traversal vulnerability when register_globals is enabled and magic_quotes is disabled and .. (dot dot) in the page parameter*
Finding Buffer Overflows

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {

    ...

3.     conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));

    ...

4.     pPostData = conn[sid].PostData;
5.     do {
6.         rc = recv(conn[sid].socket, pPostData, 1024, 0);

    ...

7.         pPostData += rc;
8.         x += rc;
9.     } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11. }
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
2.     if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {

......

3.     conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024, sizeof(char));

......

4.     pPostData=conn[sid].PostData;

5.     do {

6.         rc=recv(conn[sid].socket, pPostData, 1024, 0);

......

7.         pPostData+=rc;

8.         x+=rc;

9.     } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));

10.    conn[sid].PostData[conn[sid].dat->in_ContentLength]=\0;

11. }
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {
3.         conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));
4.         pPostData = conn[sid].PostData;
5.         do {
6.             rc = recv(conn[sid].socket, pPostData, 1024, 0);
7.             pPostData += rc;
8.             x += rc;
9.         } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.        conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11.     }
}

true branch
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1.   if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {
      ....
3.         conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));
      ....
4.         pPostData = conn[sid].PostData;
5.      do {
6.         rc = recv(conn[sid].socket, pPostData, 1024, 0);
        ....
7.         pPostData += rc;
8.         x += rc;
9.      } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11.   }
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {
      
3.         conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));
      
4.         pPostData = conn[sid].PostData;
5.         do {
6.             rc = recv(conn[sid].socket, pPostData, 1024, 0);
       
7.             pPostData += rc;
8.             x += rc;
9.         } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11. }

Allocate -1000 + 1024 bytes = 24 bytes
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
2.     if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
3.         conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024,
sizeof(char));
4.         pPostData=conn[sid].PostData;
5.         do {
6.             rc=recv(conn[sid].socket, pPostData, 1024, 0);
7.             pPostData+=rc;
8.             x+=rc;
9.         } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
10.        conn[sid].PostData[conn[sid].dat->in_ContentLength]="\0";
11.     }

Allocate -1000 + 1024 bytes = 24 bytes

true branch

Boom!
Tutorial Overview

1. Why are we doing this?
2. What is combinatorial testing?
3. What tools are available?
4. Is this stuff really useful in the real world?
5. What's next?
Fault location

Given: a set of tests that the SUT fails, which combinations of variables/values triggered the failure?

- Variable/value combinations in passing tests
- Variable/value combinations in failing tests

These are the ones we want
Fault location – what's the problem?

If they're in failing set but not in passing set:
1. which ones triggered the failure?
2. which ones don't matter?

out of \( \binom{n}{t} \) combinations

Example:
30 variables, 5 values each
= 445,331,250
  5-way combinations

142,506 combinations in each test
Tutorial Overview

1. Why are we doing this?
2. What is combinatorial testing?
3. What tools are available?
4. Is this stuff really useful in the real world?
5. What's next?
Tutorial Overview

1. Why are we doing this?

2. What is combinatorial testing?

3. What tools are available?

4. Is this stuff really useful in the real world?

5. What's next?
Tutorial Overview

1. Why are we doing this?
2. What is combinatorial testing?
3. What tools are available?
4. Is this stuff really useful in the real world?
5. What's next?
Tradeoffs

- **Advantages**
  - Tests rare conditions
  - Produces high code coverage
  - Finds faults faster
  - May be lower overall testing cost

- **Disadvantages**
  - Expensive at higher strength interactions (>4-way)
  - May require high skill level in some cases (if formal models are being used)