Combinatorial Testing

Rick Kuhn

National Institute of Standards and Technology
Gaithersburg, MD

Carnegie-Mellon University, 26 January 2010
Tutorial Overview

1. Why are we doing this?
2. What is combinatorial testing?
3. How is it used and how long does it take?
4. What tools are available?
5. What's next?
What is NIST and why are we doing this?

• A US Government agency

• The nation’s measurement and testing laboratory – 3,000 scientists, engineers, and support staff including 3 Nobel laureates

Research in physics, chemistry, materials, manufacturing, computer science

Analysis of engineering failures, including buildings, materials, and ...
Software Failure Analysis

• We studied software failures in a variety of fields including 15 years of FDA medical device recall data

• What causes software failures?
  • logic errors?
  • calculation errors?
  • interaction faults?
  • inadequate input checking? Etc.

• What testing and analysis would have prevented failures?

• Would statement coverage, branch coverage, all-values, all-pairs etc. testing find the errors?

**Interaction faults**: e.g., failure occurs if
  pressure < 10  (1-way interaction <= all-values testing catches)
  pressure < 10 & volume > 300  (2-way interaction <= all-pairs testing catches)
Software Failure Internals

• How does an interaction fault manifest itself in code?

Example: pressure < 10 & volume > 300 (2-way interaction)

```java
if (pressure < 10) {
    // do something
    if (volume > 300) { faulty code! BOOM! } else { good code, no problem
} else {
    // do something else
    }
```
Pairwise testing is popular, but is it enough?

- Pairwise testing commonly applied to software
- Intuition: some problems only occur as the result of an interaction between parameters/components
- Pairwise testing finds about 50% to 90% of flaws
  - Cohen, Dalal, Parelius, Patton, 1995 – 90% coverage with pairwise, all errors in small modules found
  - Dalal, et al. 1999 – effectiveness of pairwise testing, no higher degree interactions
  - Smith, Feather, Muscetolla, 2000 – 88% and 50% of flaws for 2 subsystems

90% of flaws. Sounds pretty good!
Finding 90% of flaws is pretty good, right?

“Relax, our engineers found 90 percent of the flaws.”

I don't think I want to get on that plane.
How about hard-to-find flaws?

- 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)
  - The most complex failure reported required 4-way interaction to trigger

Interesting, but that's just one kind of application.
How about other applications?

Browser (green)

These faults more complex than medical device software!!

Why?
And other applications?

Server (magenta)
Still more?

NASA distributed database
(light blue)

% detected

Interactions

0 10 20 30 40 50 60 70 80 90 100

1 2 3 4 5 6
Even more?

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

Network security (Bell, 2006)
(orange)

Curves appear to be similar across a variety of application domains.

Why this distribution?
What causes this distribution?

One clue: branches in avionics software.
7,685 expressions from *if* and *while* statements
Comparing with Failure Data
So, how many parameters are involved in really tricky faults?

- Maximum interactions for fault triggering for these applications was 6
- Much 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?

Biologists have a “central dogma”, and so do we:

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!
Tutorial Overview

1. Why are we doing this?

2. **What is combinatorial testing?**

3. How is it used and how long does it take?

4. What tools are available?

5. What's next?
What is combinatorial testing?
A simple example

![Image of Font dialog box with various options circled]
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:

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

We can pack a lot into one test, so what’s the smallest number of tests we need?
A covering array

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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Each row is a test:

Each column is a parameter:

Each test covers $\binom{10}{3} = 120$ 3-way combinations

Finding covering arrays is NP hard
<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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0 = effect off  
1 = effect on

13 tests for all 3-way combinations

$2^{10} = 1,024$ tests for all combinations
Another familiar example

**No silver bullet** because:
- Many values per variable
- Need to abstract values
- But we can still increase information per test

Plan: flt, flt+hotel, flt+hotel+car
From: CONUS, HI, Europe, Asia …
To: CONUS, HI, Europe, Asia …
Compare: yes, no
Date-type: exact, 1to3, flex
Depart: today, tomorrow, 1yr, Sun, Mon …
Return: today, tomorrow, 1yr, Sun, Mon …
Adults: 1, 2, 3, 4, 5, 6
Minors: 0, 1, 2, 3, 4, 5
Seniors: 0, 1, 2, 3, 4, 5
A larger example

- Suppose we have a system with on-off switches:
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
Tutorial Overview

1. Why are we doing this?
2. What is combinatorial testing?
3. How is it used and how long does it take?
4. What tools are available?
5. What's next?
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

<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>
Combinatorial testing with existing test set

1. Use t-way coverage for system configuration values
2. Apply existing tests

<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>

- Common practice in telecom industry
Modeling & Simulation Application

• “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>

$$5 \times 3 \times 4 \times 4 \times 4 \times 2 \times 2 \times 2 \times 4 \times 4 \times 4 \times 4$$

$$= 31,457,280$$

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>
Network Deadlock Detection

Detected 14 configurations that can cause deadlock:
14/31,457,280 = 4.4 x 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)
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>
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: **Expensive but can be highly effective**
Cost and Volume of Tests

- Number of tests: proportional to $v^t \log n$
  for $v$ values, $n$ variables, $t$-way interactions
- Thus:
  - Tests increase exponentially with interaction strength $t$: BAD, but unavoidable
  - But only logarithmically with the number of parameters: GOOD!
- Example: suppose we want all 4-way combinations of $n$ parameters, 5 values each:

![Graph showing the relationship between variables and tests](chart.png)

- Cost and Volume of Tests
- 0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000
- Variables

- Tests
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.     }  
5.     pPostData = conn[sid].PostData;
6.     do {
7.         rc = recv(conn[sid].socket, pPostData, 1024, 0);
8.     } while (rc > 0);  
9.         pPostData += rc;
10.        x += rc;
11.     while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
12.     conn[sid].PostData[conn[sid].dat->in_ContentLength] = '0';
13. }
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.     }
12. }

Allocate -1000 + 1024 bytes = 24 bytes

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.    }

true branch
Allocate -1000 + 1024 bytes = 24 bytes

Boom!
Ordering Pizza

Step 1 Select your favorite size and pizza crust.

- Large Original Crust

Step 2 Select your favorite pizza toppings from the pull down. Whole toppings cover the entire pizza. First ½ and second ½ toppings cover half the pizza. For a regular cheese pizza, do not add toppings.

- I want to add or remove toppings on this pizza -- add on whole or half pizza.
- Add toppings whole pizza
- Add toppings 1st half
- Add toppings 2nd half

= WAY TOO MUCH TO TEST

Simplified pizza ordering:

\[ 6 \times 2^{17} \times 2^{17} \times 2^{17} \times 4 \times 3 \times 2 \times 2 \times 5 \times 2 \]

= 184,320 possibilities

Step 3 Select your pizza instructions.

- I want to add special instructions for this pizza -- light, extra or no sauce; light or no cheese; well done bake
- Regular Sauce
- Normal Cheese
- Normal Bake
- Normal Cut

Step 4 Add to order.

Quantity 1

Add To Order
Add To Order & Checkout
Ordering Pizza Combinatorially

Simplified pizza ordering:

\[ 6 \times 4 \times 4 \times 4 \times 3 \times 2 \times 2 \times 5 \times 2 = 184,320 \text{ possibilities} \]

2-way tests: 32
3-way tests: 150
4-way tests: 570
5-way tests: 2,413
6-way tests: 8,330

If all failures involve 5 or fewer parameters, then we can have confidence after running all 5-way tests.

So what? Who has time to check 2,413 test results?
How to 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

  • **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

- 2-way: nval=2
- 3-way: nval=6
- 4-way: nval=10

- Values per variable:
  - 4.50-5.00
  - 4.00-4.50
  - 3.50-4.00
  - 3.00-3.50
  - 2.50-3.00
  - 2.00-2.50
  - 1.50-2.00
  - 1.00-1.50
  - 0.50-1.00
  - 0.00-0.50
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:

\begin{verbatim}
ensures balance == \old(balance) - amount
&& \result == balance;
\end{verbatim}

• 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
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:

- **A** φ - All: φ holds on all paths starting from the current state.
- **E** φ - Exists: φ holds on some paths starting from the current state.
- **G** φ - Globally: φ has to hold on the entire subsequent path.
- **F** φ - Finally: φ eventually has to hold
- **X** φ - Next: φ has to hold at the next state

[others not listed]

execution paths

states on the execution paths

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 $\text{AG}(P \rightarrow AX(R))$
  “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. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \& P \rightarrow AX ! (R))$
  2. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow AX ! (1))$
  3. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow AX ! (R))$
What happens with these assertions?

1. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \land P \to AX \neg(R))$

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

1. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \to AX \neg(1))$

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. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \to AX \neg(R))$

Forces production of a counterexample for each R.
This is just right!
Tradeoffs

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

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

1. Why are we doing this?
2. What is combinatorial testing?
3. What is it good for?
4. How much does it cost?

5. **What tools are available?**
6. What's next?
New algorithms to make it practical

- **Tradeoffs to minimize calendar/staff time:**
  - FireEye (extended IPO) – Lei – roughly optimal, can be used for most cases under 40 or 50 parameters
    - Produces minimal number of tests at cost of run time
    - Currently integrating algebraic methods
  - Adaptive distance-based strategies – Bryce – dispensing one test at a time w/ metrics to increase probability of finding flaws
    - Highly optimized covering array algorithm
    - Variety of distance metrics for selecting next test
  - PRMI – Kuhn – for more variables or larger domains
    - Parallel, randomized algorithm, generates tests w/ a few tunable parameters; computation can be distributed
    - Better results than other algorithms for larger problems
## 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><strong>IPOG</strong></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><strong>Size</strong></td>
<td><strong>Time</strong></td>
<td><strong>Size</strong></td>
<td><strong>Time</strong></td>
<td><strong>Size</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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): $27^{24}11^{10^2}$

Times in seconds

That's fast!

Unlike diet plans, results ARE typical!
# ACTS Tool

![FireEye 1.0 - FireEye Main Window](image)

## System View
- **Root node**
- **SYSTEM-TCAST**
  - Cur_Veritical_Sep
    - 299
    - 300
    - 601
  - High_Confidence
    - true
    - false
  - Two_of_Three_Reports
    - true
    - false
  - Own_Tracked_Alt
    - 1
    - 2
  - Other_Tracked_Alt
    - 1
    - 2
  - Own_Tracked_Alt_Rate
    - 600
    - 601
  - Alt_Layer_Value
    - 0
    - 1
    - 2
    - 3
  - Up_Separation
    - 0
    - 399
    - 400
    - 499
    - 500
    - 639
    - 601

## Test Result

<table>
<thead>
<tr>
<th>CUR_V...</th>
<th>HIGH...</th>
<th>TWO...</th>
<th>OWN...</th>
<th>OTHER...</th>
<th>OWN...</th>
<th>ALT_L...</th>
<th>UP_SE...</th>
<th>DOWN...</th>
<th>THE...</th>
<th>OTHER...</th>
<th>CLIMB...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>299</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>1</td>
<td>600</td>
<td>0</td>
<td>399</td>
<td>399</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>2</td>
<td>300</td>
<td>false</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>0</td>
<td>400</td>
<td>400</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>3</td>
<td>601</td>
<td>true</td>
<td>True</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>0</td>
<td>499</td>
<td>499</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>4</td>
<td>299</td>
<td>true</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>639</td>
<td>NO_INT...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
<td>false</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>0</td>
<td>500</td>
<td>500</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>6</td>
<td>601</td>
<td>false</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>639</td>
<td>639</td>
<td>NO_INT...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>7</td>
<td>299</td>
<td>false</td>
<td>false</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>640</td>
<td>NO_INT...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>739</td>
<td>NO_INT...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>9</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>0</td>
<td>740</td>
<td>740</td>
<td>DO_NO...</td>
<td>TCAS_TA true</td>
</tr>
<tr>
<td>10</td>
<td>299</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>840</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>11</td>
<td>300</td>
<td>false</td>
<td>true</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>2</td>
<td>399</td>
<td>399</td>
<td>DO_NO...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>12</td>
<td>601</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>3</td>
<td>399</td>
<td>399</td>
<td>DO_NO...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>13</td>
<td>299</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>0</td>
<td>399</td>
<td>399</td>
<td>NO_INT...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>14</td>
<td>300</td>
<td>false</td>
<td>true</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>399</td>
<td>399</td>
<td>NO_INT...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>15</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>600</td>
<td>2</td>
<td>399</td>
<td>500</td>
<td>NO_INT...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>16</td>
<td>299</td>
<td>false</td>
<td>true</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>399</td>
<td>639</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>17</td>
<td>300</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>600</td>
<td>0</td>
<td>399</td>
<td>640</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>18</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>1</td>
<td>399</td>
<td>739</td>
<td>NO_INT...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>19</td>
<td>299</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>399</td>
<td>740</td>
<td>NO_INT...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>20</td>
<td>300</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>3</td>
<td>399</td>
<td>840</td>
<td>NO_INT...</td>
<td>TCAS_TA true</td>
</tr>
<tr>
<td>21</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>1</td>
<td>400</td>
<td>400</td>
<td>NO_INT...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>22</td>
<td>299</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>0</td>
<td>400</td>
<td>400</td>
<td>NO_INT...</td>
<td>TCAS_TA false</td>
</tr>
<tr>
<td>23</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>400</td>
<td>400</td>
<td>NO_INT...</td>
</tr>
<tr>
<td>24</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>400</td>
<td>400</td>
<td>NO_INT...</td>
</tr>
<tr>
<td>25</td>
<td>299</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>400</td>
<td>500</td>
<td>NO_INT...</td>
</tr>
<tr>
<td>26</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>400</td>
<td>639</td>
<td>DO_NO...</td>
</tr>
<tr>
<td>27</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>400</td>
<td>640</td>
<td>DO_NO...</td>
</tr>
<tr>
<td>28</td>
<td>299</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>400</td>
<td>739</td>
<td>DO_NO...</td>
</tr>
<tr>
<td>29</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>400</td>
<td>740</td>
<td>DO_NO...</td>
</tr>
<tr>
<td>30</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>400</td>
<td>840</td>
<td>DO_NO...</td>
</tr>
<tr>
<td>31</td>
<td>299</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>1</td>
<td>600</td>
<td>3</td>
<td>499</td>
<td>499</td>
<td>DO_NO...</td>
<td>OTHER...</td>
</tr>
<tr>
<td>32</td>
<td>300</td>
<td>false</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>2</td>
<td>499</td>
<td>499</td>
<td>DO_NO...</td>
<td>TCAS_TA false</td>
</tr>
</tbody>
</table>
Defining a new system
Variable interaction strength

[Software interface showing parameters and strength values]
Constraints
Covering array output

<table>
<thead>
<tr>
<th>CUR_...</th>
<th>HIGH_...</th>
<th>TWO_...</th>
<th>OWN_...</th>
<th>OTHER_...</th>
<th>OWN_...</th>
<th>ALT_...</th>
<th>UP_SE_...</th>
<th>DOWN_...</th>
<th>THE_...</th>
<th>OTHER_...</th>
<th>CLIMATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>299</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>1</td>
<td>600</td>
<td>0</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>300</td>
<td>false</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>DO_NO_...</td>
<td>OTHER_false</td>
</tr>
<tr>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>600</td>
<td>2</td>
<td>0</td>
<td>499</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>299</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>1</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>499</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>300</td>
<td>false</td>
<td>true</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>NO_INT_...</td>
<td>OTHER_true</td>
</tr>
<tr>
<td>601</td>
<td>false</td>
<td>true</td>
<td>1</td>
<td>1</td>
<td>600</td>
<td>2</td>
<td>0</td>
<td>639</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>299</td>
<td>false</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>640</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>300</td>
<td>false</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>739</td>
<td>0</td>
<td>NO_INT_...</td>
<td>OTHER_true</td>
</tr>
<tr>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>1</td>
<td>600</td>
<td>0</td>
<td>0</td>
<td>740</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>100</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>1</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>840</td>
<td>0</td>
<td>NO_INT_...</td>
<td>OTHER_false</td>
</tr>
<tr>
<td>129</td>
<td>false</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>300</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>129</td>
<td>false</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>300</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>143</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>143</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>300</td>
<td>false</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>399</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>143</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>600</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
<tr>
<td>143</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>1</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>NO_INT_...</td>
<td>TCAS_TA_true</td>
</tr>
</tbody>
</table>
Output

- Variety of output formats:
  - XML
  - Numeric
  - CSV
  - Excel

- Separate tool to generate .NET configuration files from ACTS output

- Post-process output using Perl scripts, etc.
# Output options

## Mappable values

| Degree of interaction coverage: 2 |
|-----------------|-----------------|-----------------|
| Number of parameters: 12          |
| Number of tests: 100              |

![Table of values]

## Human readable

<table>
<thead>
<tr>
<th>Degree of interaction coverage: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parameters: 12</td>
</tr>
<tr>
<td>Maximum number of values per parameter: 10</td>
</tr>
<tr>
<td>Number of configurations: 100</td>
</tr>
</tbody>
</table>

### 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  

Etc.
Eclipse Plugin for ACTS

Work in progress
Eclipse Plugin for ACTS

Defining parameters and values
Tutorial Overview

1. Why are we doing this?
2. What is combinatorial testing?
3. How is it used and how long does it take?
4. What tools are available?
5. What's next?
## Combinatorial Coverage Measurement

<table>
<thead>
<tr>
<th>Test</th>
<th>Variables</th>
<th>Variable pairs</th>
<th>Variable-value combinations covered</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, c, d</td>
<td>ab</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, d</td>
<td>ac</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>3</td>
<td>a, b, c, d</td>
<td>ad</td>
<td>00, 01, 11</td>
<td>.75</td>
</tr>
<tr>
<td>4</td>
<td>a, b, c, d</td>
<td>bc</td>
<td>00, 11</td>
<td>.50</td>
</tr>
<tr>
<td>5</td>
<td>a, b, c, d</td>
<td>bd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>a, b, c, d</td>
<td>cd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
<tr>
<td>7</td>
<td>a, b, c, d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a, b, c, d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Combinatorial Coverage Measurement

Configuration coverage for $2^{793}141691$ inputs.

- Measure coverage provided by existing test sets
- Compare across methodologies

What this means:
for 70% of 4-way variable combinations, tests cover at least 40% of variable-value configurations
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

These are the ones we want

variable/value combinations in failing tests
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 $v^n_t$ combinations

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

142,506 combinations in each test
Conclusions

- Empirical research suggests that all software failures caused by interaction of few parameters.
- Combinatorial testing can exercise all t-way combinations of parameter values in a very tiny fraction of the time needed for exhaustive testing.
- New algorithms and faster processors make large-scale combinatorial testing possible.
- Project could produce better quality testing at lower cost for US industry and government.
- Beta release of tools available, to be open source.
- New public catalog of covering arrays.
Future directions

Real-world examples will help answer these questions
What kinds of software does it work best on?
What kinds of errors does it miss?

• Other applications:
  • Modelling and simulation
    • Testing the simulation
    • Finding interesting combinations:
      performance problems, denial of service attacks
  • Maybe biotech applications. Others?

Please contact us if you are interested!

Rick Kuhn
kuhn@nist.gov

Raghu Kacker
raghu.kacker@nist.gov

http://csrc.nist.gov/acts
(Or just search “combinatorial testing”. We’re #1!)