Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
Automated Combinatorial Testing

- Goals – reduce testing cost, improve cost-benefit ratio for software assurance
- Merge automated test generation with combinatorial methods
- New algorithms to make large-scale combinatorial testing practical
- Accomplishments – huge increase in performance, scalability + widespread use in real-world applications
- Also non-testing applications – modelling and simulation
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
- pressure < 10 & volume > 300

(1-way interaction <= all-values testing catches)
(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
}
```

A test that included pressure = 5 and volume = 400 would trigger this failure
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
- Tests all pairs (2-way combinations) of variable values
- Pairwise testing finds about 50% to 90% of flaws

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

NIST study of 15 years of FDA medical device recall data

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)
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

Branch statements
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!
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
What is Combinatorial Testing?
What is combinatorial 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:

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

  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

Each row is a test:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Finding covering arrays is NP hard
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
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

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

Step 4  Add to order.

- Quantity 1

Simplified pizza ordering:

6x2^{17}x2^{17}x2^{17}x4x3x2x2x5x2
= WAY TOO MUCH TO TEST

6x4x4x4x4x3x2x2x5x2
= 184,320 possibilities
Ordering Pizza Combinatorially

Simplified pizza ordering:

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

2-way tests: \hspace{1em} 32
3-way tests: \hspace{1em} 150
4-way tests: \hspace{1em} 570
5-way tests: \hspace{1em} 2,413
6-way tests: \hspace{1em} 8,330

If all failures involve 5 or fewer parameters, then we can have confidence after running all 5-way tests.
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
Two ways of using combinatorial testing

Use combinations here

or here

Test case | OS    | CPU | Protocol |
----------|-------|------|----------|
1         | Windows | Intel | IPv4     |
2         | Windows | AMD  | IPv6     |
3         | Linux   | Intel | IPv6     |
4         | Linux   | AMD  | IPv4     |

Configuration

System under test
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>
Configurations to Test

Degree of interaction coverage: 2
Number of parameters: 5
Maximum number of values per parameter: 3
Number of configurations: 10

-------------------------------------
Configuration #1:
1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv4
4 = CPU=Intel
5 = DBMS=MySQL

-------------------------------------
Configuration #2:
1 = OS=XP
2 = Browser=Firefox
3 = Protocol=IPv6
4 = CPU=AMD
5 = DBMS=Sybase

-------------------------------------
Configuration #3:
1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv6
4 = CPU=Intel
5 = DBMS=Oracle

. . . etc.

<table>
<thead>
<tr>
<th>t</th>
<th># Tests</th>
<th>% of Exhaustive</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>36</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>72</td>
<td>100</td>
</tr>
</tbody>
</table>
Testing Smartphone Configurations

Android configuration options:

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 NAVIGATIONHIDDEN_NO;
int NAVIGATIONHIDDEN_UNDEFINED;
int NAVIGATIONHIDDEN_YES;
int NAVIGATION_DPAD;
int NAVIGATION_NONAV;
int NAVIGATION_TRACKBALL;
int NAVIGATION_UNDEFINED;
int NAVIGATION_WHEEL;
int ORIENTATION_LANDSCAPE;
int ORIENTATION_PORTRAIT;
int ORIENTATION_SQUARE;
int ORIENTATION_UNDEFINED;
int SCREENLAYOUT_LONG_MASK;
int SCREENLAYOUT_LONG_NO;
int SCREENLAYOUT_LONG_UNDEFINED;
int SCREENLAYOUT_LONG_YES;
int SCREENLAYOUT_SIZE_LARGE;
int SCREENLAYOUT_SIZE_MASK;
int SCREENLAYOUT_SIZE_NORMAL;
int SCREENLAYOUT_SIZE_SMALL;
int SCREENLAYOUT_SIZE_UNDEFINED;
int TOUCHSCREEN_FINGER;
int TOUCHSCREEN_NOTOUCH;
int TOUCHSCREEN_STYLUS;
int TOUCHSCREEN_UNDEFINED;
## 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 tests generated

<table>
<thead>
<tr>
<th>t</th>
<th># Tests</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>
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
Evolution of design of experiments (DOE) to combinatorial testing of software and systems using covering arrays
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
Early history

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 each
Principles used (blocking, replication, randomization)
Theoretical contributor of basic ideas: Charles S Peirce
American logician, philosopher, mathematician
1939-1914, Cambridge, MA
Father of DOE: R A Fisher, 1890-1962, British geneticist
Rothamsted Experiment Station, Hertfordshire, England
Four eras of evolution of DOE

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

Era 2:(1940’s …): Use for industrial productivity

Era 3:(1980’s …): Use for designing robust products

Era 4:(2000’s …): Combinatorial Testing of Software

Hardware-Software systems, computer security, assurance of access control policy implementation (health care records), verification and validations of simulations, optimization of models, testing of cloud computing applications, platform, and infrastructure
Features of DOE

1. System under investigation
2. Variables (input, output and other), test settings
3. Objectives
4. Scope of investigation
5. Key principles
6. Experiment plans
7. Analysis method from data to conclusions
8. Some leaders (subjective, hundreds of contributors)
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
Agriculture and biological investigations-3

Analysis method from data to conclusions
  Simple statistical model for treatment effects
  ANOVA (Analysis of Variance)
  Significant factors among primary factors; better test settings

Some of the leaders
  R A Fisher, F Yates, …
  G W Snedecor, C R Henderson*, Gertrude Cox, …
  W G Cochran*, Oscar Kempthorne*, D R Cox*, …

Other: Double-blind clinical trials, biostatistics and medical application at forefront
Industrial productivity-1

System under investigation

- Chemical production process, manufacturing processes
- Mechanistic (cause-effect) process; predictability medium

Variable Types:
- Not allocation of treatments to units
- Primary test factors: process variables levels can be adjusted
- Held constant
- Continue to use terminology from agriculture
- Generally less than 10

Objectives:
- Identify important factors, predict their optimum levels
- Estimate response function for important factors
Industrial productivity-2

Scope of investigation:

- Optimum levels in range of possible values (beyond levels actually used)

Key principles

- **Replication**: Necessary
- **Randomization of test runs**: Necessary
- **Blocking (homogeneous grouping)**: Needed less often

Designs: Test runs for chosen settings

- Factorial and Fractional factorial designs
- Latin squares, Greco-Latin squares
- Central composite designs, Response surface designs
Industrial productivity-3

Analysis method from data to conclusions

Estimation of linear or quadratic statistical models for relation between factor levels and response

Linear ANOVA or regression models

Quadratic response surface models

Factor levels

Chosen for better estimation of model parameters

Main effect: average effect over level of all other factors

2-way interaction effect: how effect changes with level of another

3-way interaction effect: how 2-way interaction effect changes; often regarded as error

Estimation requires balanced DOE

Some of the leaders

G. E. P. Box*, G. J. Hahn*, C. Daniel, C. Eisenhart*,
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)

Analysis method from data to conclusions

Pseudo-statistical analysis
Signal-to-noise ratios, measures of variability

Some of the leaders: Genichi Taguchi
Use of OAs for software testing

Functional (black-box) testing

- Hardware-software systems
- Identify single and 2-way combination faults

Early papers

- Taguchi followers (mid1980’s)
- Mandl (1985) Compiler testing
- Brownlie et al (1992) AT&T

Generation of test suites using OAs

- OATS (Phadke*, AT&T-BL)
Combinatorial Testing of Software and Systems -1

System under investigation

- Hardware-software systems combined or separately
- Mechanistic (cause-effect) process; predictability full (high)
- Output unchanged (or little changed) in repeats
- Configurations of system or inputs to system

Variable Types: test-factors and held constant

- Inputs and configuration variables having more than one option
- No limit on variables and test setting

Identification of factors and test settings

- Which could trigger malfunction, boundary conditions
- Understand functionality, possible modes of malfunction

Objectives: Identify $t$-way combinations of test setting of any $t$ out of $k$ factors in tests actually conducted which trigger malfunction; $t << k$
Combinatorial Testing of Software and Systems -2

Scope of investigation:

Actual $t$-way (and higher) combinations tested; no prediction

Key principles: no background no uncontrolled factors

No need of blocking and randomization

No need of replication; greatly decrease number of test runs

Investigation of actual faults suggests: $1 < t < 7$

Complex constraints between test settings (depending on possible paths software can go through)

Designs: Covering arrays cover all $t$-way combinations

Allow for complex constraints

Other DOE can be used; CAs require fewer tests (exception when OA of index one is available which is best CA)

‘Interaction’ means number of variables in combination (not estimate of parameter of statistical model as in other DOE)
Combinatorial Testing of Software and Systems -3

Analysis method from data to conclusions

No statistical model for test setting-output relationship; no prediction

No estimation of statistical parameters (main effects, interaction effects)

Test suite need not be balanced; covering arrays unbalanced

Often output is \{0,1\}

Need algorithms to identify fault triggering combinations

Some leaders

AT&T-BL alumni (Neil Sloan*), Charlie Colbourn* (AzSU) …

NIST alumni/employees (Rick Kuhn*), Jeff Yu Lei* (UTA/NIST)

Other applications

Assurance of access control policy implementations

Computer security, health records
Components of combinatorial testing

Problem set up: identification of factors and settings
Test run: combination of one test setting for each factor
Test suite generation, high strength, constraints
Test execution, integration in testing system
Test evaluation / expected output oracle
Fault localization
Generating test suites based on CAs

CATS (Bell Labs), AETG (BellCore-Telcordia)
IPO (Yu Lei) led to ACTS (IPOG, …)
Tconfig (Ottawa), CTGS (IBM), TOG (NASA),…
Jenny (Jenkins), TestCover (Sherwood),…
PICT (Microsoft),…
ACTS (NIST/UTA) free, open source intended
   Effective efficient for \( t \)-way combinations for \( t = 2, 3, 4, 5, 6, \ldots \)
   Allow complex constraints
Mathematics underlying DOE/CAs

1829-32 Évariste Galois (French, shot in dual at age 20)
1940’s R. C. Bose (father of math underlying DOE)
1947 C. R. Rao* (concept of orthogonal arrays)

Hadamard (1893), RC Bose, KA Bush, Addelman, Taguchi,
1960’s G. Taguchi* (catalog of OAs, industrial use)

Covering arrays (Sloan* 1993) as math objects

Renyi (1971, probabilist, died at age 49)
Roux (1987, French, disappeared leaving PhD thesis)
Katona (1973), Kleitman and Spencer (1973), Sloan* (1993),

CAs connection to software testing: key papers

Dalal* and Mallows* (1997), Cohen, Dalal, Fredman, Patton (1997),
Alan Hartman* (2003), …

Catalog of Orthogonal Arrays (N J A Sloan*, AT&T)
Sizes of Covering Arrays (C J Colbourn*, AzSU)
Concluding remarks

DOE: approach to gain information to improve things

Combinatorial Testing is a special kind of DOE

Chosen input → function → observe output

Highly predictable system; repeatability high understood

Input space characterized in terms of factors, discrete settings

Critical event when certain $t$-way comb encountered $t << k$

Detect such $t$-way combinations or assure absence

Exhaustive testing of all $k$-way combinations not practical

No statistical model assumed

Unbalanced test suites

Smaller size test suites than other DOE plans, which can be used

Many applications
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
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>IPOG</th>
<th>ITCH (IBM)</th>
<th>Jenny (Open Source)</th>
<th>TConfig (U. of Ottawa)</th>
<th>TVG (Open Source)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>0.8</td>
<td>120</td>
<td>0.73</td>
<td>108</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
<td>0.36</td>
<td>2388</td>
<td>1020</td>
<td>413</td>
</tr>
<tr>
<td>4</td>
<td>1363</td>
<td>3.05</td>
<td>1484</td>
<td>5400</td>
<td>1536</td>
</tr>
<tr>
<td>5</td>
<td><strong>4226</strong></td>
<td>18s</td>
<td>NA</td>
<td><strong>&gt;1 day</strong></td>
<td>4580</td>
</tr>
<tr>
<td>6</td>
<td>10941</td>
<td>65.03</td>
<td>NA</td>
<td><strong>&gt;1 day</strong></td>
<td>11625</td>
</tr>
</tbody>
</table>

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

Times in seconds

That's fast!

Unlike diet plans, results ARE typical!
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:
Defining a new system
Variable interaction strength
Constraints
Covering array output

<table>
<thead>
<tr>
<th>CUR_V...</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>OTH...</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>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</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>1</td>
<td>0</td>
<td>0</td>
<td>DO_NO...</td>
<td>OTHER</td>
</tr>
<tr>
<td>3</td>
<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>0</td>
<td>DO_NO...</td>
<td>OTHER</td>
</tr>
<tr>
<td>4</td>
<td>601</td>
<td>false</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>DO_NO...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>6</td>
<td>301</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DO_NO...</td>
<td>OTHER</td>
</tr>
<tr>
<td>7</td>
<td>299</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>DO_NO...</td>
<td>OTHER</td>
</tr>
<tr>
<td>9</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>OTHER</td>
</tr>
<tr>
<td>10</td>
<td>299</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>DO_NO...</td>
<td>OTHER</td>
</tr>
<tr>
<td>11</td>
<td>300</td>
<td>false</td>
<td>true</td>
<td>2</td>
<td>2</td>
<td>600</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>12</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>13</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>14</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>15</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>16</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>17</td>
<td>300</td>
<td>true</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>600</td>
<td>1</td>
<td>0</td>
<td>0</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>2</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>19</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>20</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>21</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>22</td>
<td>601</td>
<td>true</td>
<td>false</td>
<td>2</td>
<td>2</td>
<td>601</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NO_INT...</td>
<td>TCAS_TA</td>
</tr>
<tr>
<td>23</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>24</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>25</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>26</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>27</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>28</td>
<td>601</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>29</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>31</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>32</td>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>400</td>
<td>400</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

### MappableView values

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

---

<table>
<thead>
<tr>
<th>Configuration #1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = Cur_Vertical_Sep=299</td>
</tr>
<tr>
<td>2 = High_Confidence=true</td>
</tr>
<tr>
<td>3 = Two_of_Three_Reports=true</td>
</tr>
<tr>
<td>4 = Own_Tracked_Alt=1</td>
</tr>
<tr>
<td>5 = Other_Tracked_Alt=1</td>
</tr>
<tr>
<td>6 = Own_Tracked_Alt_Rate=600</td>
</tr>
<tr>
<td>7 = Alt_Layer_Value=0</td>
</tr>
<tr>
<td>8 = Up_Separation=0</td>
</tr>
<tr>
<td>9 = Down_Separation=0</td>
</tr>
<tr>
<td>10 = Other_RAC=NO_INTENT</td>
</tr>
<tr>
<td>11 = Other_Capability=TCAS_CA</td>
</tr>
<tr>
<td>12 = Climb_Inhibit=true</td>
</tr>
</tbody>
</table>

---

### Human readable

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

---

<table>
<thead>
<tr>
<th>Configuration #1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = Cur_Vertical_Sep=299</td>
</tr>
<tr>
<td>2 = High_Confidence=true</td>
</tr>
<tr>
<td>3 = Two_of_Three_Reports=true</td>
</tr>
<tr>
<td>4 = Own_Tracked_Alt=1</td>
</tr>
<tr>
<td>5 = Other_Tracked_Alt=1</td>
</tr>
<tr>
<td>6 = Own_Tracked_Alt_Rate=600</td>
</tr>
<tr>
<td>7 = Alt_Layer_Value=0</td>
</tr>
<tr>
<td>8 = Up_Separation=0</td>
</tr>
<tr>
<td>9 = Down_Separation=0</td>
</tr>
<tr>
<td>10 = Other_RAC=NO_INTENT</td>
</tr>
<tr>
<td>11 = Other_Capability=TCAS_CA</td>
</tr>
<tr>
<td>12 = Climb_Inhibit=true</td>
</tr>
</tbody>
</table>

---

Etc.
Eclipse Plugin for ACTS

Work in progress
Eclipse Plugin for ACTS

Defining parameters and values
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
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

  • **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>Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00-0.50</td>
<td>nval=2</td>
</tr>
<tr>
<td>0.50-1.00</td>
<td>nval=6</td>
</tr>
<tr>
<td>1.00-1.50</td>
<td>nval=10</td>
</tr>
<tr>
<td>1.50-2.00</td>
<td></td>
</tr>
<tr>
<td>2.00-2.50</td>
<td></td>
</tr>
<tr>
<td>2.50-3.00</td>
<td></td>
</tr>
<tr>
<td>3.00-3.50</td>
<td></td>
</tr>
<tr>
<td>3.50-4.00</td>
<td></td>
</tr>
<tr>
<td>4.00-4.50</td>
<td></td>
</tr>
<tr>
<td>4.50-5.00</td>
<td></td>
</tr>
</tbody>
</table>
Built-in Self Test through 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;
Built-in Self Test

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
Model Checking Example

- 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 $t$ values](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:
Expensive but can be highly effective
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)
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
## Document Object Model Events

<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>Focus</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>KeyDown</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>KeyUp</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Param.</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<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
### World Wide Web Consortium
Document Object Model Events

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>% of Orig.</th>
<th>Pass</th>
<th>Fail</th>
<th>Not Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>702</td>
<td>1.92%</td>
<td>202</td>
<td>27</td>
<td>473</td>
</tr>
<tr>
<td>3</td>
<td>1342</td>
<td>3.67%</td>
<td>786</td>
<td>27</td>
<td>529</td>
</tr>
<tr>
<td>4</td>
<td>1818</td>
<td>4.96%</td>
<td>437</td>
<td>72</td>
<td>1309</td>
</tr>
<tr>
<td>5</td>
<td>2742</td>
<td>7.49%</td>
<td>908</td>
<td>72</td>
<td>1762</td>
</tr>
<tr>
<td>6</td>
<td>4227</td>
<td>11.54%</td>
<td>1803</td>
<td>72</td>
<td>2352</td>
</tr>
</tbody>
</table>

All failures found using < 5% of original pseudo-exhaustive test set
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) {
5.         pPostData = conn[sid].PostData;
8.             x += rc;
9.         } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11. }
4.     pPostData = conn[sid].PostData;
7.     pPostData += rc;

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 = realloc(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. }

**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.     } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
8.     x += rc;
9.     conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
10.    }
11.   }
12. 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
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
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. }

true branch
Allocate  -1000 + 1024 bytes = 24 bytes
Boom!
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>

\[5 \times 3 \times 4 \times 4 \times 4 \times 2 \times 2 \times 2 \times 4 \times 4 \times 4 \times 4 \times 4 = 31,457,280\] 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:
\[ \frac{14}{31,457,280} = 4.4 \times 10^{-7} \]

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)
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
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 \( \binom{n}{t} \) combinations

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

142,506 combinations in each test
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
## Combinatorial Coverage Measurement

<table>
<thead>
<tr>
<th>Tests</th>
<th>Variables</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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
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

Detect all values automatically
Set boundaries for equivalence classes

Parameter 0

Values for this parameter:
0.1

Chart:
X = proportion of combinations
Y = combination variable-value coverage

2 way stats:
Combinations: 3,321
Var/val coms: 14,761
Total coverage: 0.940

3 way stats:
Combinations: 88,560
Var/val coms: 828,135
Total coverage: 0.831

Coverage:
- 2-way coverage:
- 3-way coverage:

Coverage for Nc: 7489 tests, 82 parameters loaded.
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
Combinatorial 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
Outline

1. Why we are doing this?
2. Number of variables involved in actual software failures
3. What is combinatorial testing (CT)?
4. Design of expts (DoE) vs CT based on covering arrays (CA)
5. Number of tests in t-way testing based on CAs
6. Tool to generate combinatorial test suites based on CAs
7. Determining expected output for each test run
8. Applications (Modeling and simulation, Security vulnerability)
9. Fault localization
10. Combinatorial coverage measurement
11. Sequence covering arrays
12. Conclusion
Industrial Usage Reports

- Work with US Air Force on sequence covering arrays, submitted for publication
- World Wide Web Consortium DOM Level 3 events conformance test suite
- Cooperative Research & Development Agreement with Lockheed Martin Aerospace - report to be released 3rd or 4th quarter 2011
Technology Transfer

Tools obtained by 700+ organizations; NIST “textbook” on combinatorial testing downloaded 8,000+ times since Oct. 2010

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!)