National Institute of Standards and Technology

Combinatorial Testing

Rick Kuhn Raghu Kacker

National Institute of
Standards and Technology
Gaithersburg, MD

Institute for Defense Analyses 6 April 2011

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

NIST National Institute of Standards and Technology

- 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

UNIVERSITY OF TEXAS ARLINGTON

NIST

NC STATE

GEORGE MASON UNIVERSITY

UtahState University

U.S. AIR FORCE

46th TEST SQ

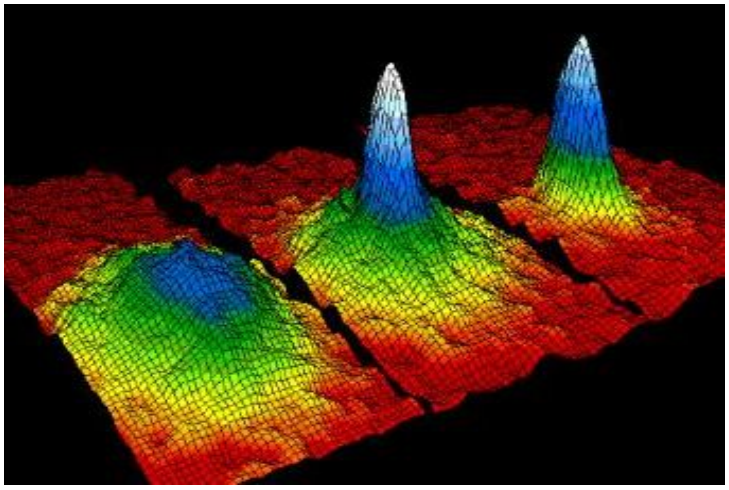
UMBC

APL The Johns Hopkins University Applied Physics Laboratory

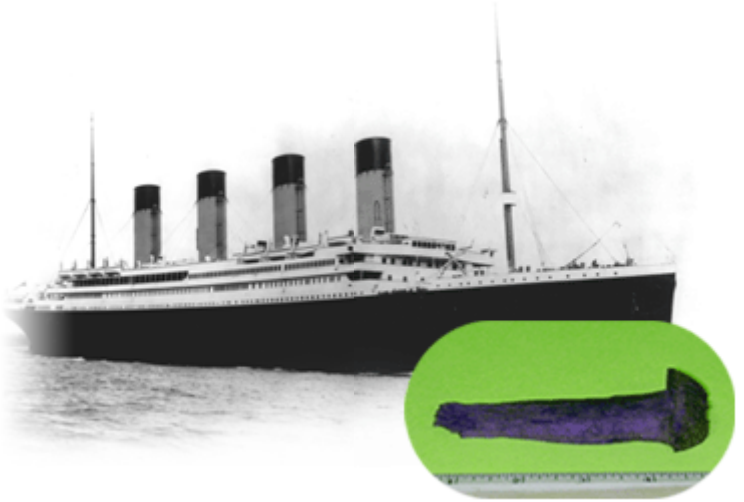
LOCKHEED MARTIN
We never forget who we're working for™

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 ...**

NIST National Institute of Standards and Technology

Software Failure Analysis

NIST National Institute of Standards and Technology

- 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

NIST National Institute of Standards and Technology

- How does an interaction fault manifest itself in code?

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

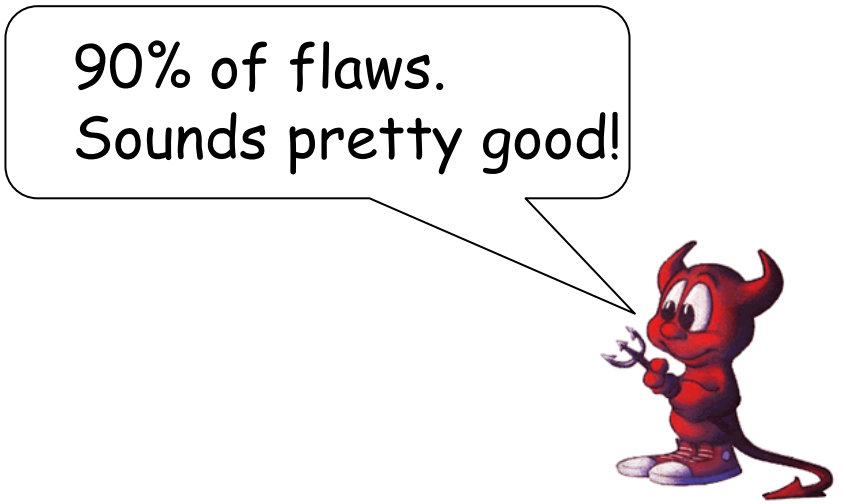
```
if (pressure < 10) {
    // do something
    if (volume > 300)  { faulty code!  BOOM! }
    else { good code, no problem}
} else {
    // do something else
}
```

A test that included pressure = 5 and volume = 400 would trigger this failure

Pairwise testing is popular, but is it enough?

NIST National Institute of Standards and Technology

- 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



Finding 90% of flaws is pretty good, right?

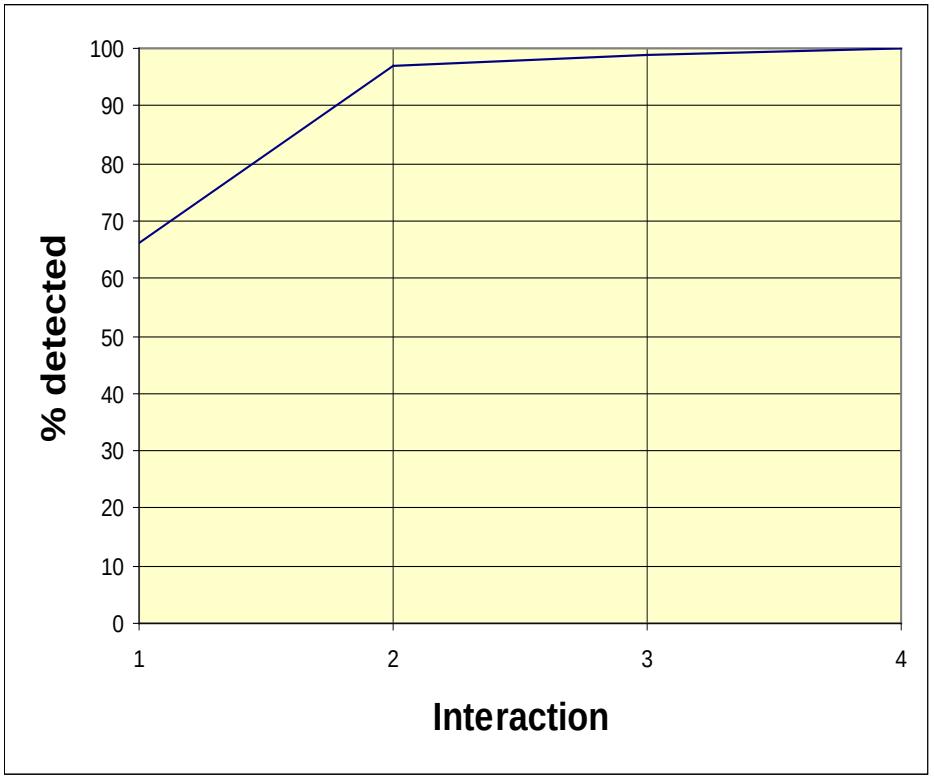
"Relax, our engineers found 90 percent of the flaws."

NIST
National Institute of Standards and Technology

How about hard-to-find flaws?

NIST National Institute of Standards and Technology

- 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

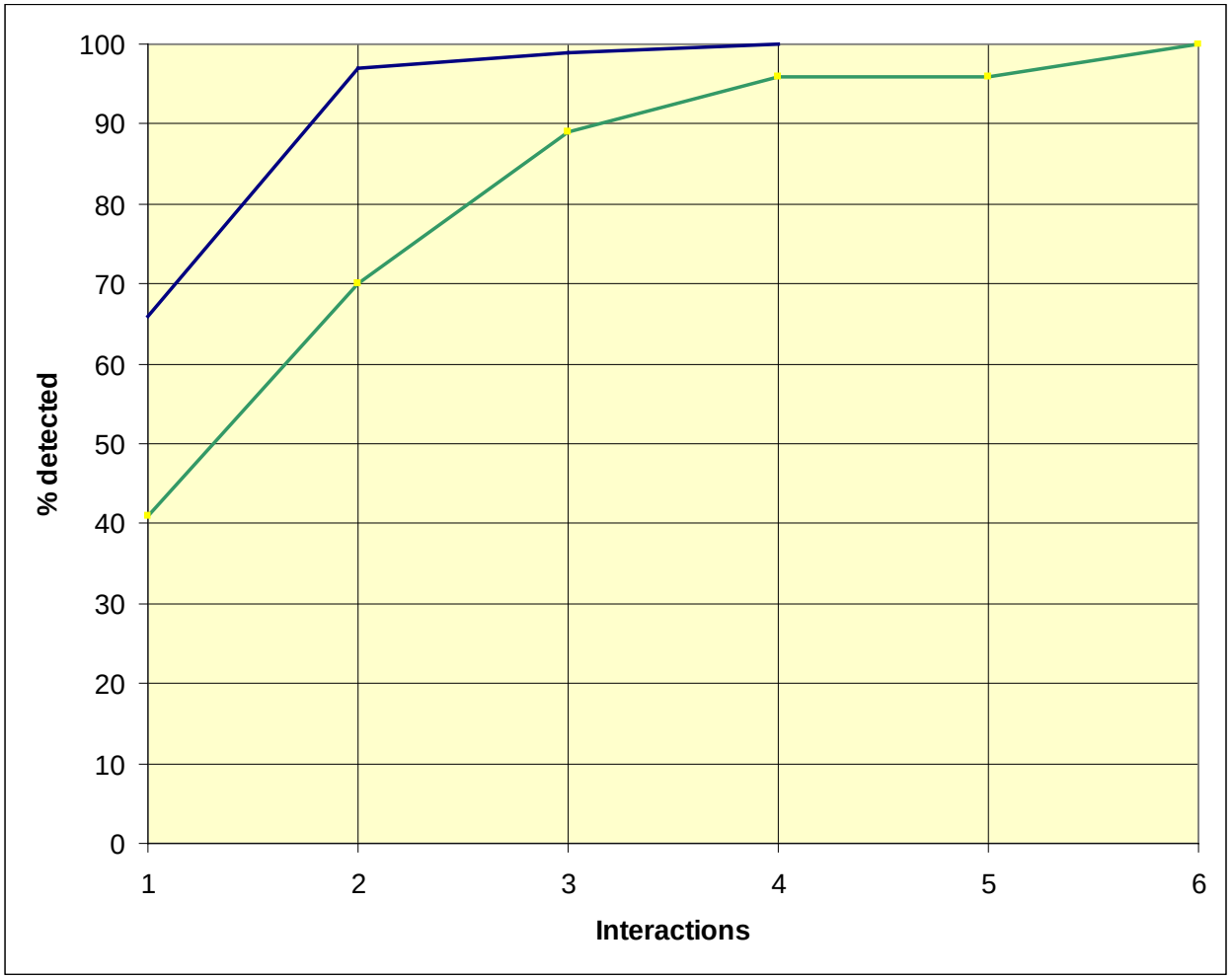


NIST study of 15 years of FDA medical device recall data

How about other applications?

NIST National Institute of Standards and Technology

Browser (green)



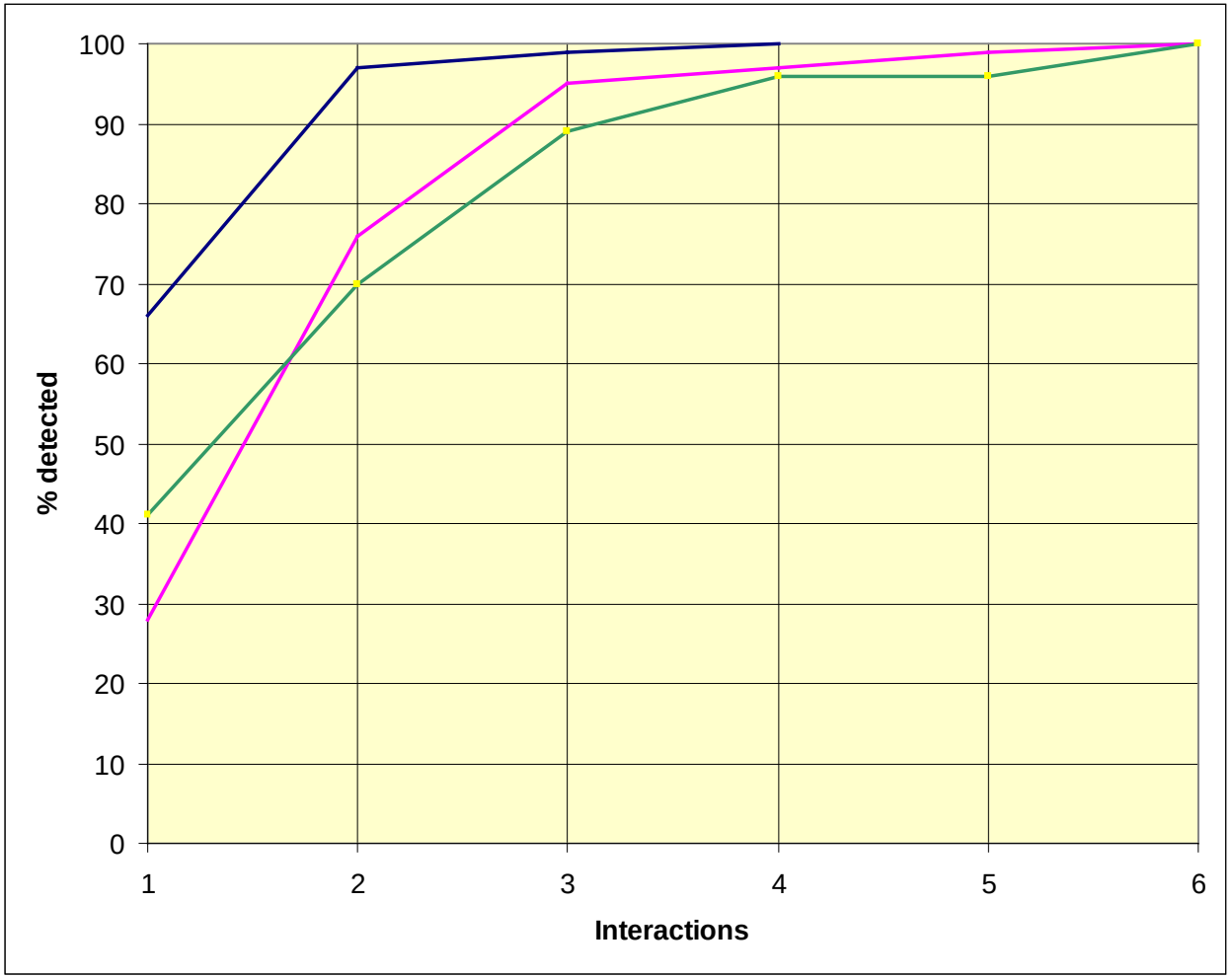
These faults more complex than medical device software!!

Why?

And other applications?

National Institute of Standards and Technology

Server (magenta)

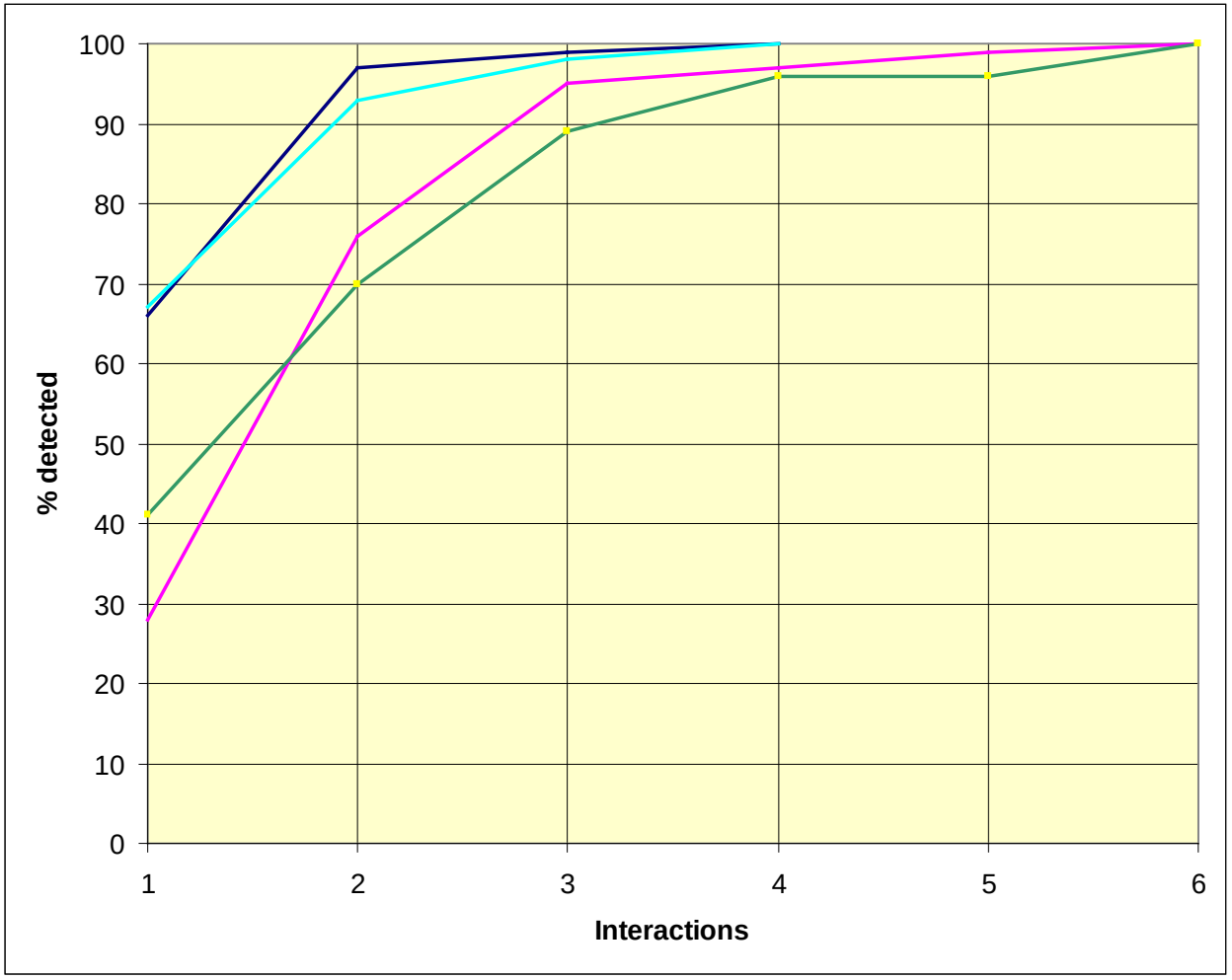


Still more?

NIST National Institute of Standards and Technology

NASA distributed database

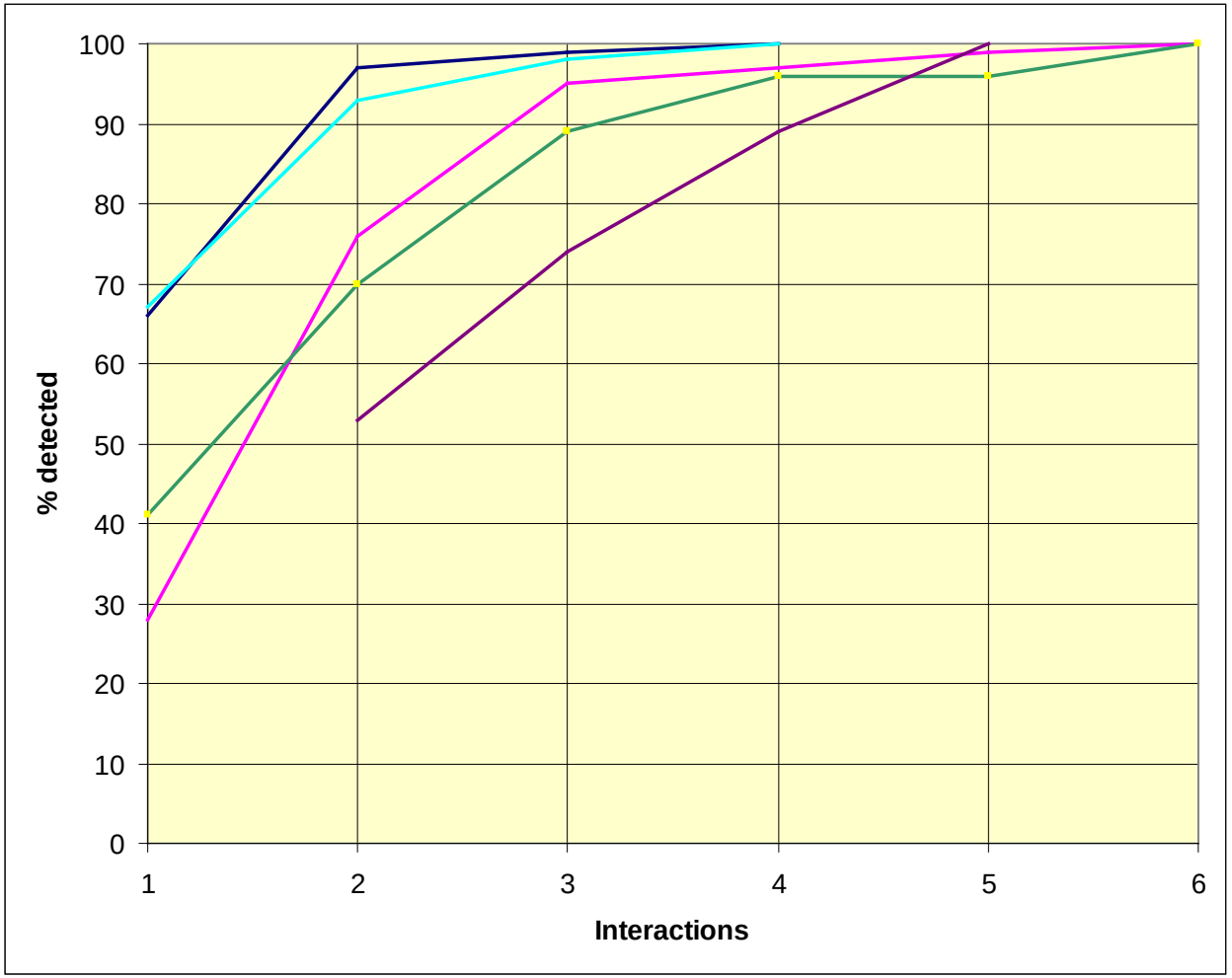
(light blue)



Even more?

National Institute of Standards and Technology

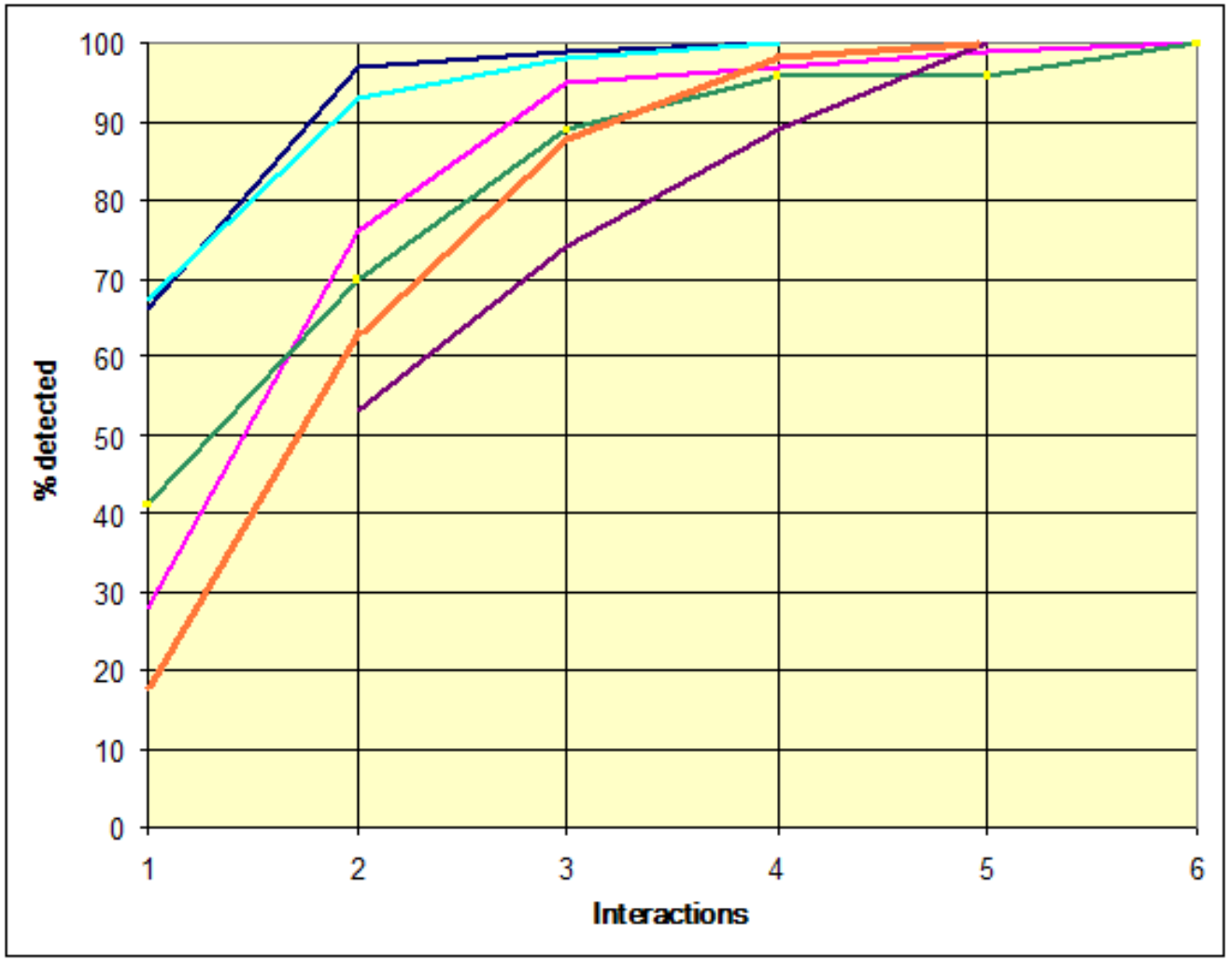
Traffic Collision Avoidance System module (seeded errors) (purple)



Finally

Network security (Bell, 2006)

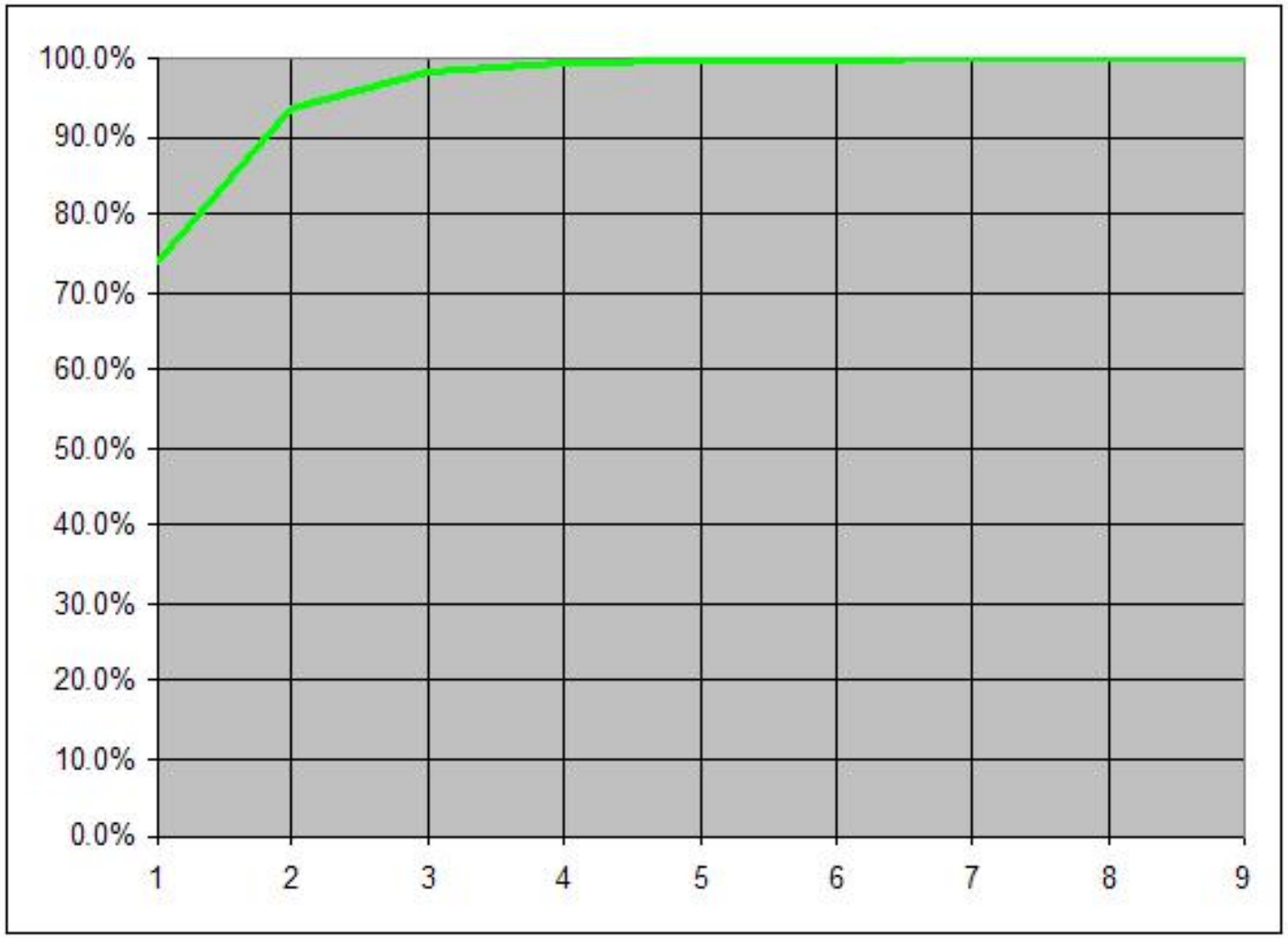
(orange)



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

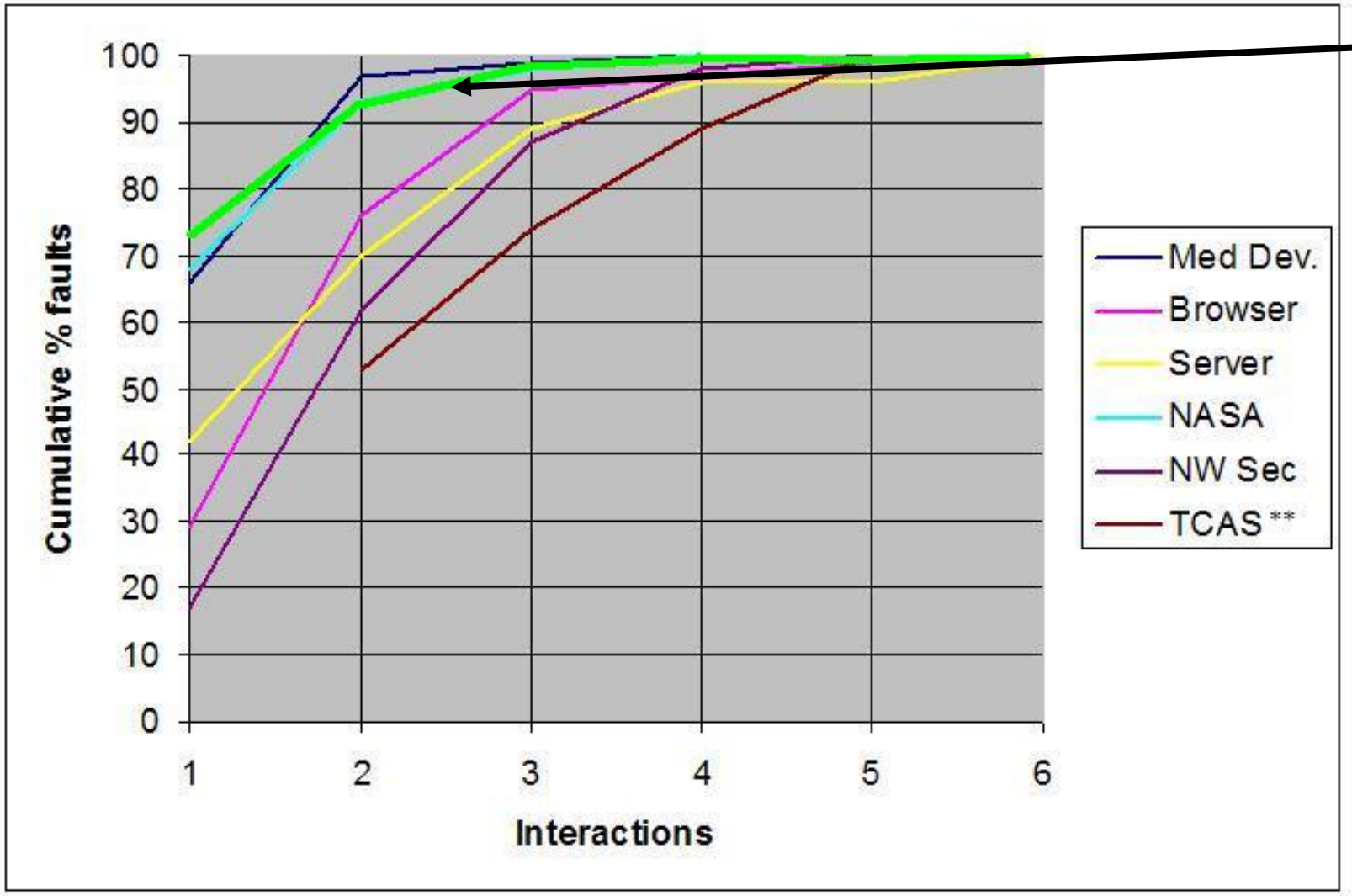
Why this distribution?

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 this knowledge help?

NIST National Institute of Standards and Technology

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

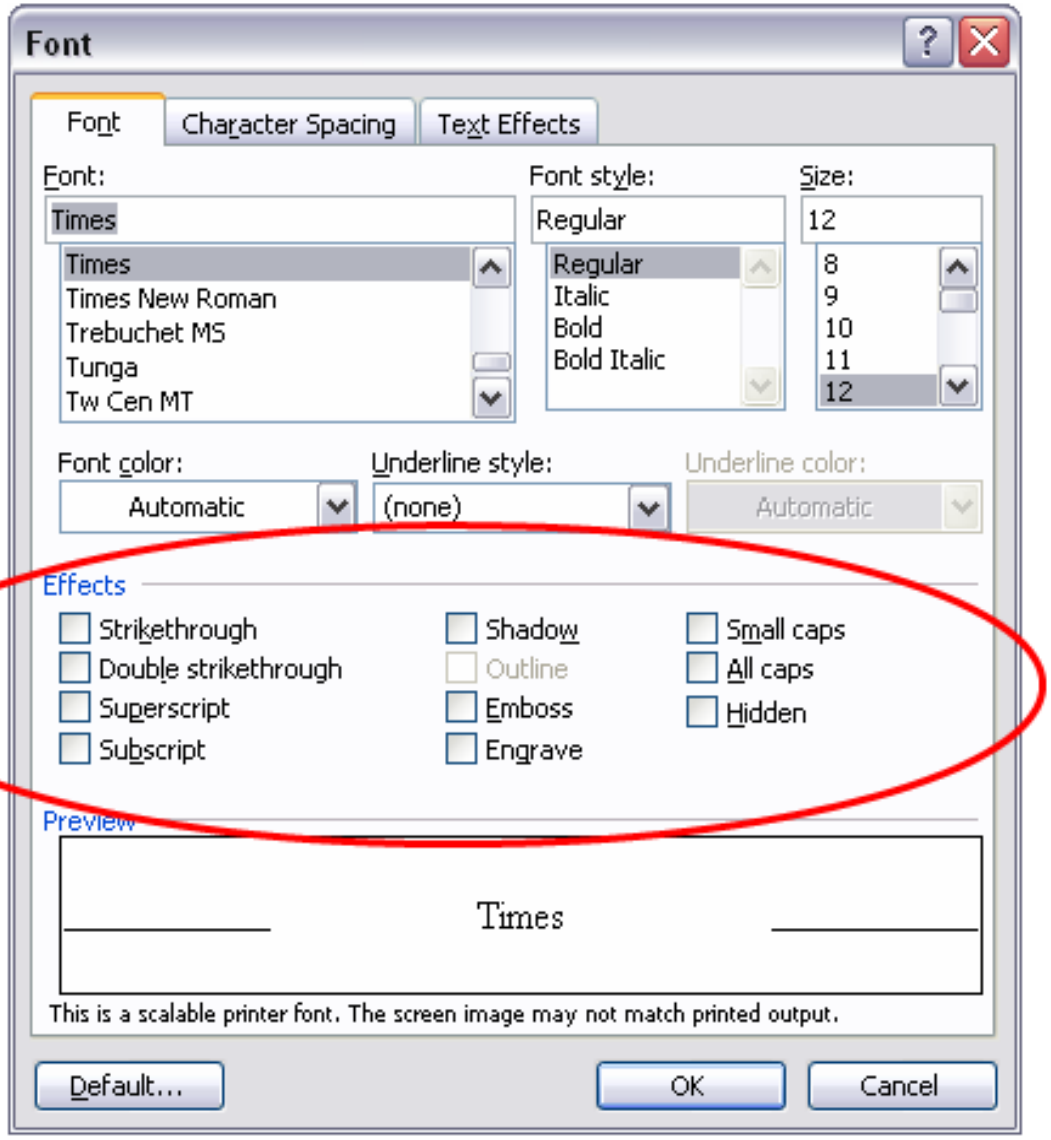
(taking into account: value propagation issues, equivalence partitioning, timing issues, more complex interactions, . . .)

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

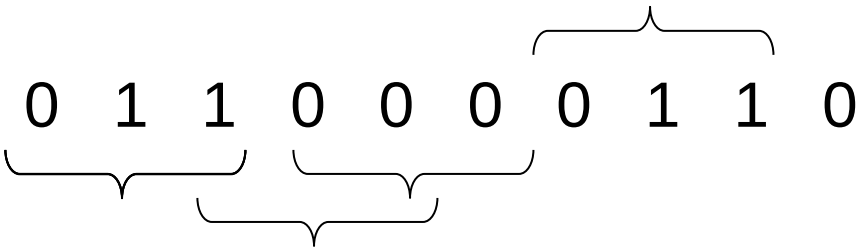


How Many Tests Would It Take?

- 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 x 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:



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

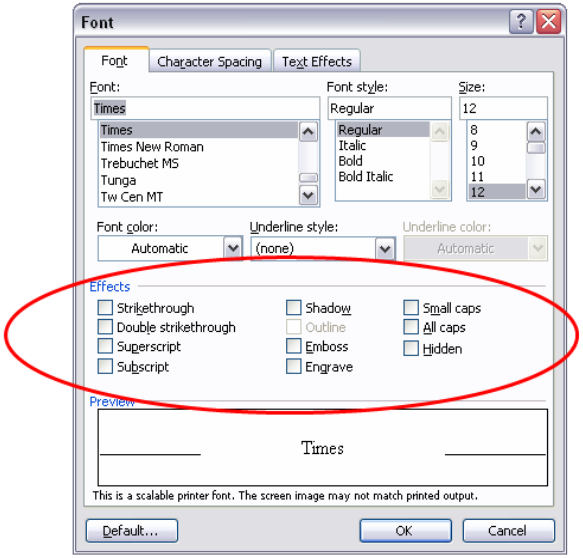
A covering array

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

Each row is a test:

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Each column is a parameter:

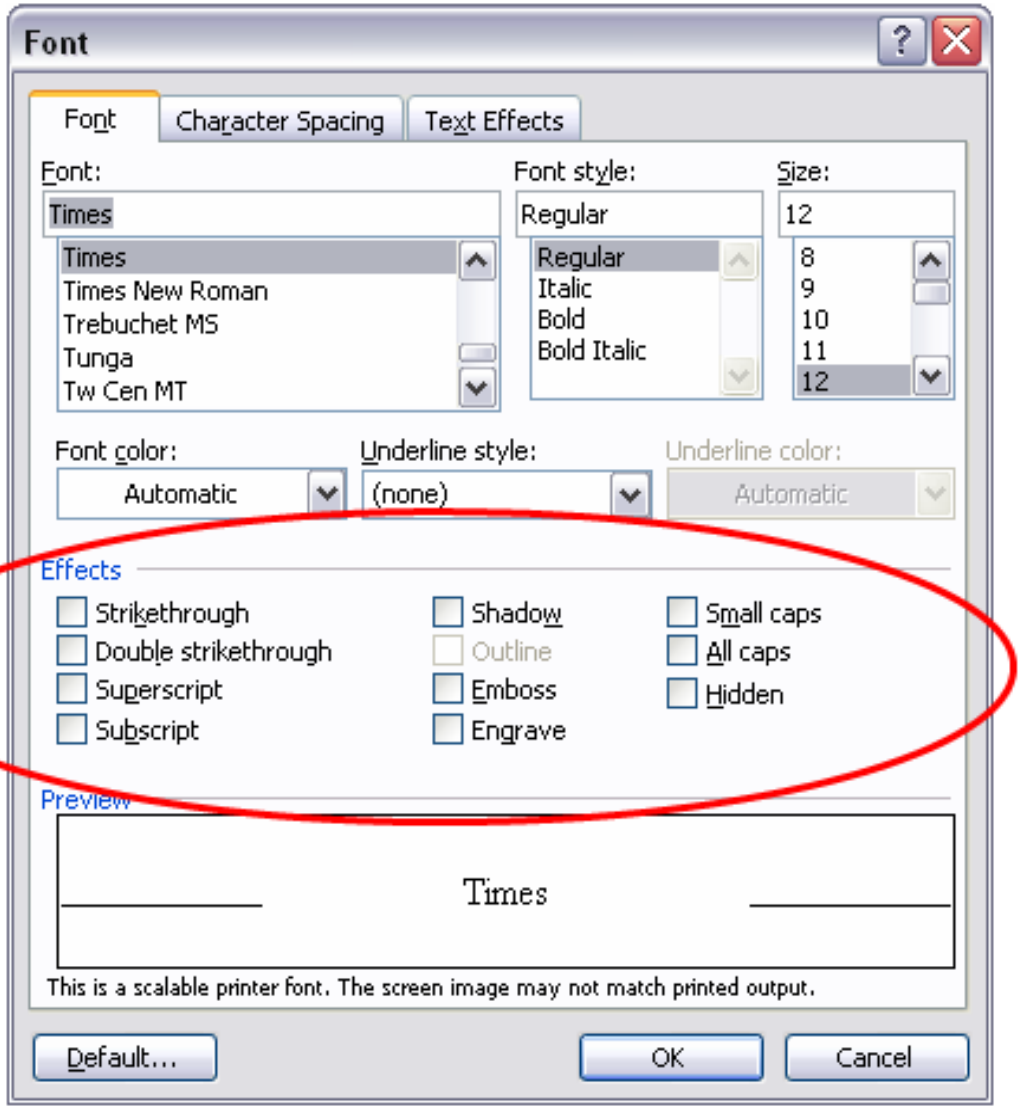


Each test covers $\binom{10}{3}$ = 120 3-way combinations

Finding covering arrays is NP hard

NIST National Institute of Standards and Technology

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

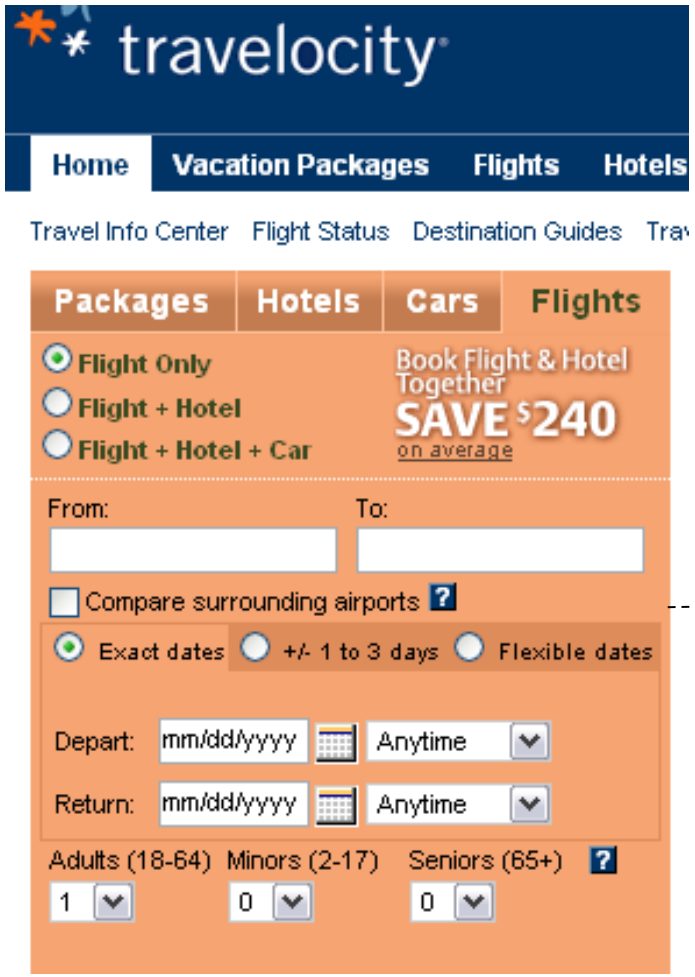


0 = effect off
1 = effect on

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

NIST National Institute of Standards and Technology

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.

Extra Cheese Remove | Bacon Remove | Black Olives Remove

Add toppings whole pizza

Add toppings 1st half

Add toppings 2nd half

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

Simplified pizza ordering:

6x4x4x4x4x3x2x2x5x2
= 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

NIST National Institute of Standards and Technology

Simplified pizza ordering:

6x4x4x4x4x3x2x2x5x2
= 184,320 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.

A larger example

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

How do we test this?

- 34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = 1.7 x 10^{10} tests

What if we knew no failure involves more than 3 switch settings interacting?

- 34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = **1.7 x 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

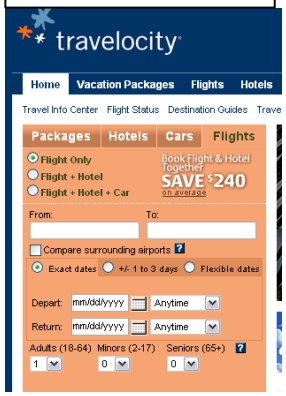
Use combinations here

or here

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

Configuration

Test data inputs



System under test

NIST National Institute of Standards and Technology

Testing Configurations

- Example: app must run on any configuration of OS, browser, protocol, CPU, and DBMS
- Very effective for interoperability testing

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

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.

t	# Tests	% of Exhaustive
2	10	14
3	18	25
4	36	50
5	72	100

NIST
National Institute of Standards and Technology

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 KEYBOARD_UNDEFINED;
int NAVIGATIONHIDDEN_NO;
int NAVIGATIONHIDDEN_UNDEFINED;
int NAVIGATIONHIDDEN_YES;
int NAVIGATION_DPAD;
int NAVIGATION_NONAV;
int NAVIGATION_TRACKBALL;
int NAVIGATION_UNDEFINED;
int NAVIGATION_WHEEL;
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

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

Total possible configurations:

3 x 3 x 4 x 3 x 5 x 4 x 4 x 5 x 4 = 172,800

NIST National Institute of Standards and Technology

Number of tests generated

t	# Tests	% of Exhaustive
2	29	0.02
3	137	0.08
4	625	0.4
5	2532	1.5
6	9168	5.3

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
- Tatsumi et al (1987) Fujitsu
- Sacks et al (1989) Computer experiments
- 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

NIST National Institute of Standards and Technology

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, …
- 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

T-Way	IPOG		ITCH (IBM)		Jenny (Open Source)		TConfig (U. of Ottawa)		TVG (Open Source)	
	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
2	100	0.8	120	0.73	108	0.001	108	>1 hour	101	2.75
3	400	0.36	2388	1020	413	0.71	472	>12 hour	9158	3.07
4	1363	3.05	1484	5400	1536	3.54	1476	>21 hour	64696	127
5	4226	18s	NA	>1 day	4580	43.54	NA	>1 day	313056	1549
6	10941	65.03	NA	>1 day	11625	470	NA	>1 day	1070048	12600

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

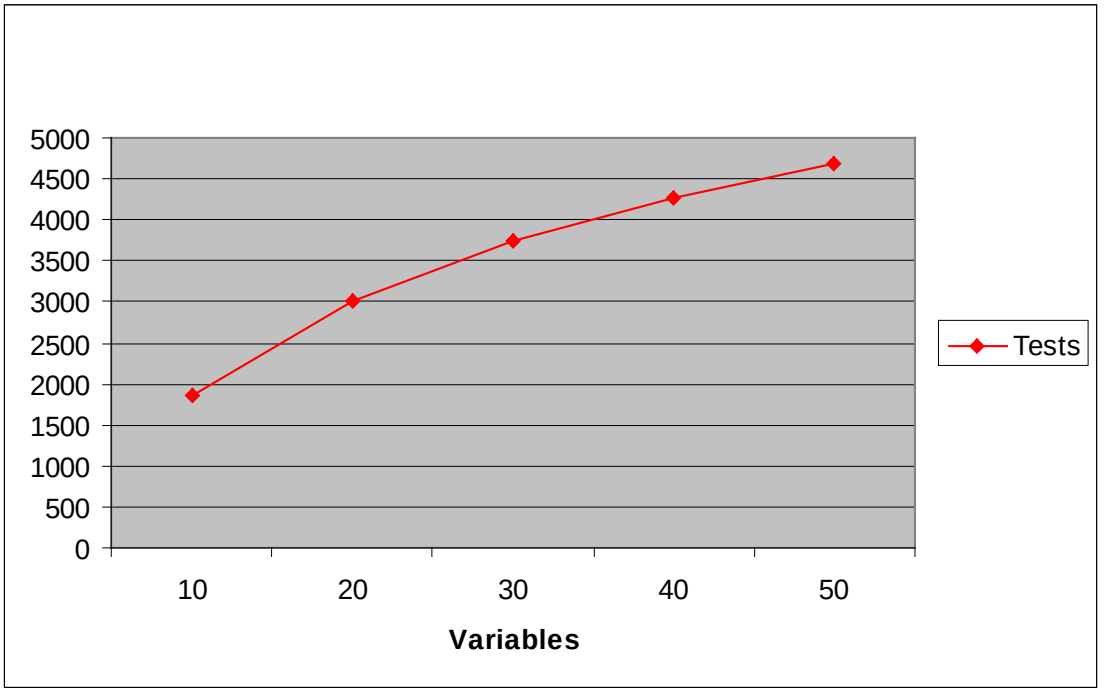
Times in seconds

That's fast!

Unlike diet plans,
results ARE typical!

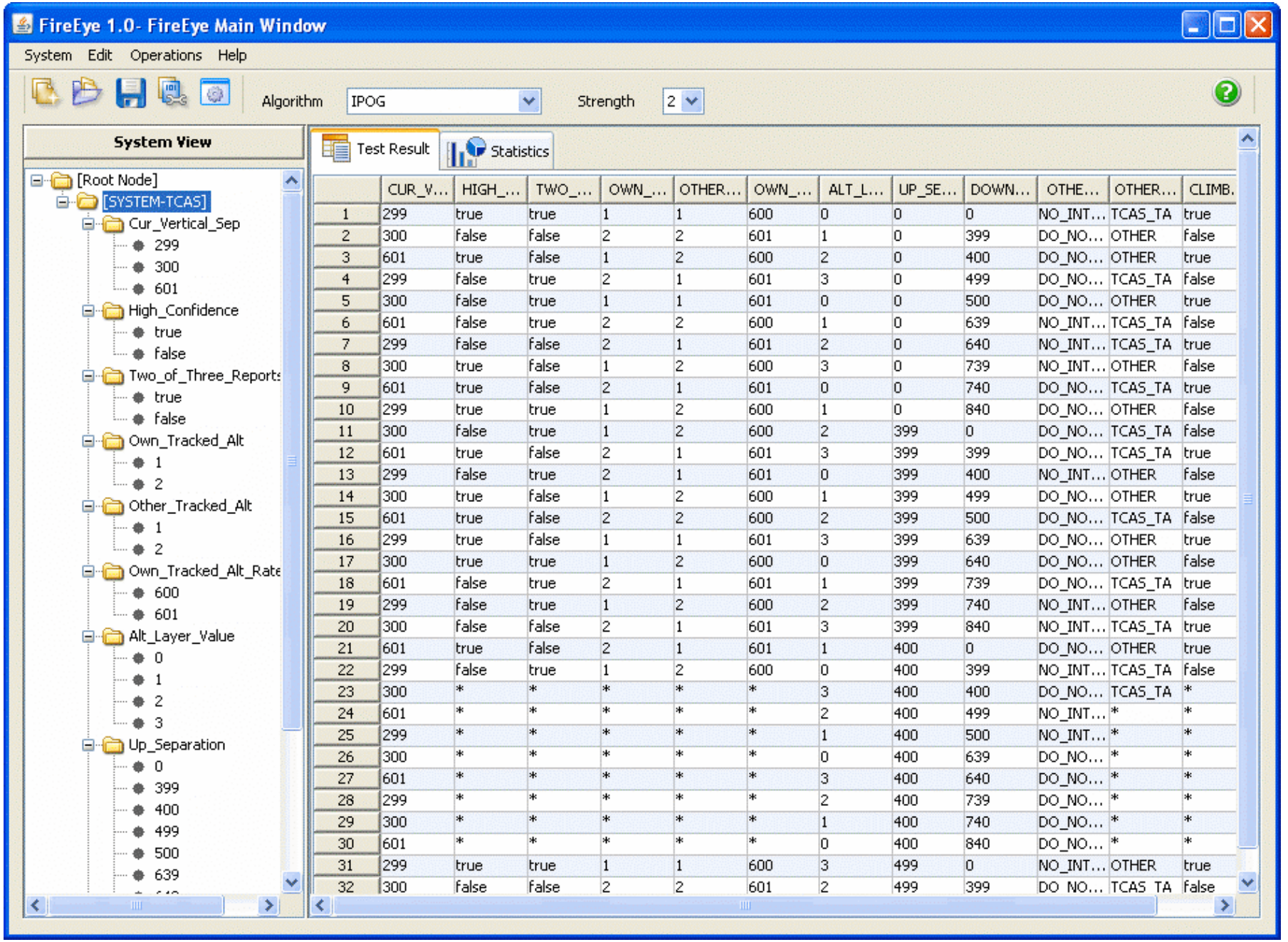
Cost and Volume of Tests

- 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:



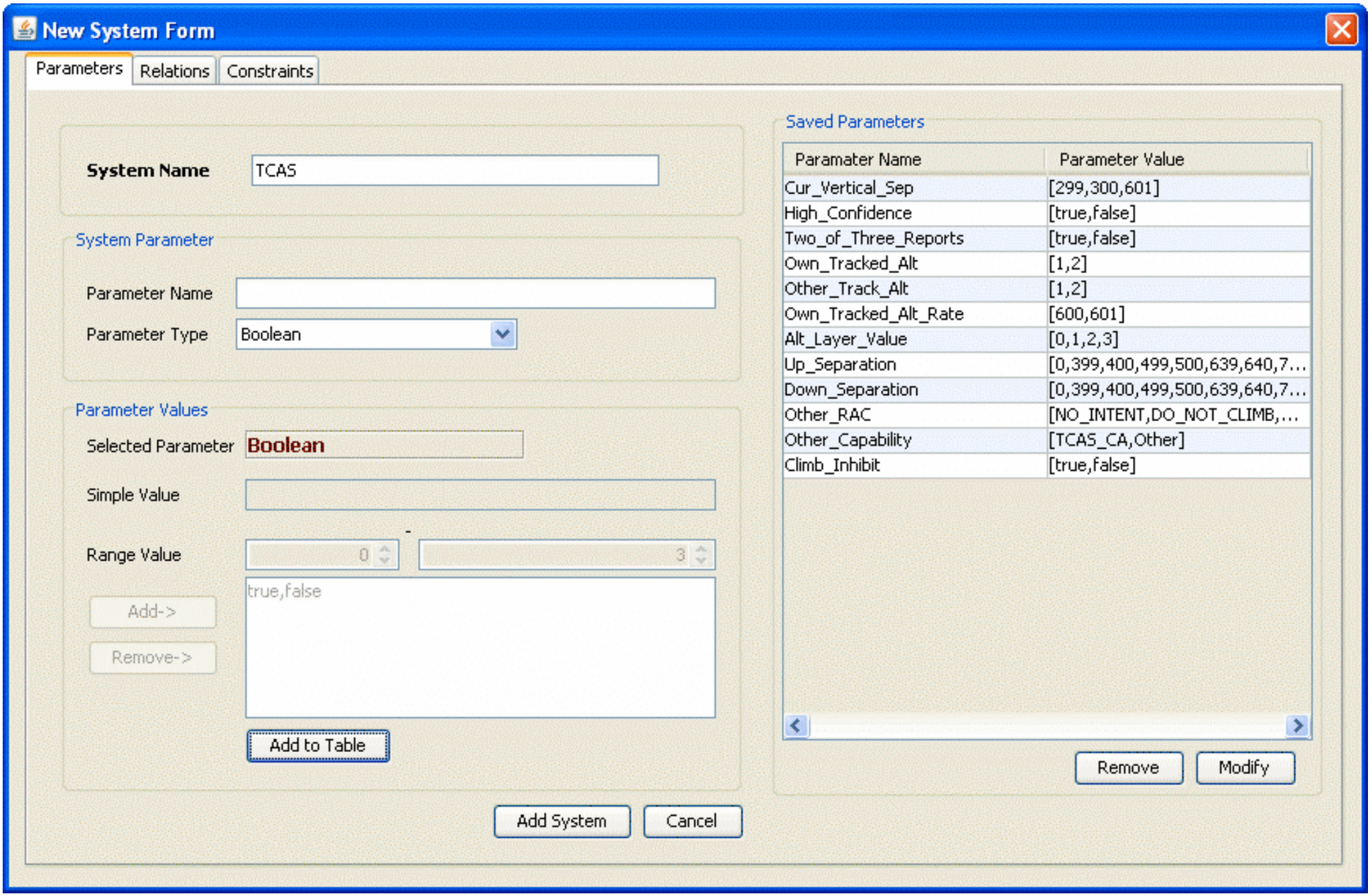
ACTS Tool

NIST National Institute of Standards and Technology

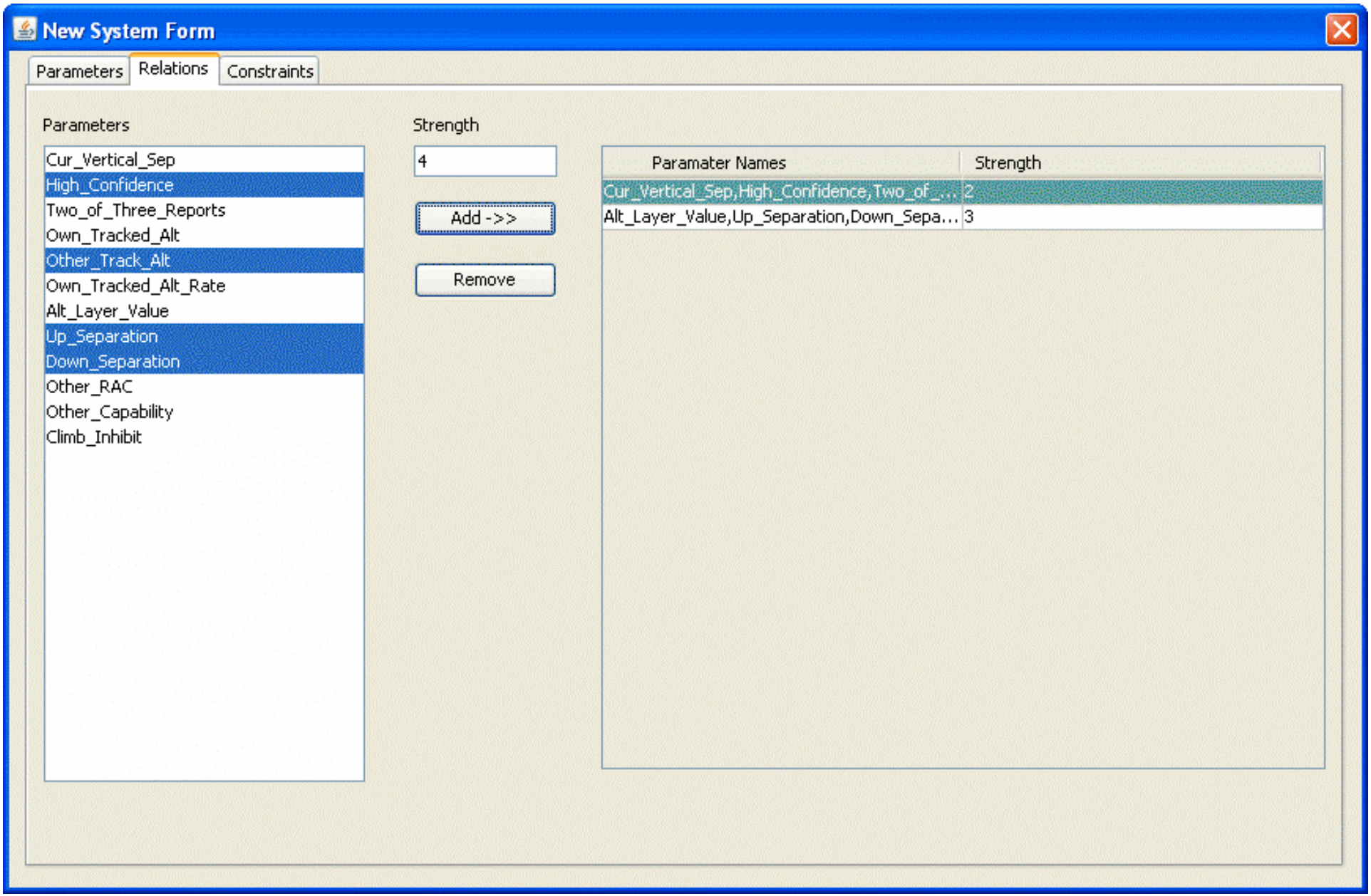


	CUR_V...	HIGH_...	TWO_...	OWN_...	OTHER...	OWN_...	ALT_L...	UP_SE...	DOWN...	OTHE...	OTHER...	CLIMB.
1	299	true	true	1	1	600	0	0	0	NO_INT...	TCAS_TA	true
2	300	false	false	2	2	601	1	0	399	DO_NO...	OTHER	false
3	601	true	false	1	2	600	2	0	400	DO_NO...	OTHER	true
4	299	false	true	2	1	601	3	0	499	DO_NO...	TCAS_TA	false
5	300	false	true	1	1	601	0	0	500	DO_NO...	OTHER	true
6	601	false	true	2	2	600	1	0	639	NO_INT...	TCAS_TA	false
7	299	false	false	2	1	601	2	0	640	NO_INT...	TCAS_TA	true
8	300	true	false	1	2	600	3	0	739	NO_INT...	OTHER	false
9	601	true	false	2	1	601	0	0	740	DO_NO...	TCAS_TA	true
10	299	true	true	1	2	600	1	0	840	DO_NO...	OTHER	false
11	300	false	true	1	2	600	2	399	0	DO_NO...	TCAS_TA	false
12	601	true	false	2	1	601	3	399	399	DO_NO...	TCAS_TA	true
13	299	false	true	2	1	601	0	399	400	NO_INT...	OTHER	false
14	300	true	false	1	2	600	1	399	499	DO_NO...	OTHER	true
15	601	true	false	2	2	600	2	399	500	DO_NO...	TCAS_TA	false
16	299	true	false	1	1	601	3	399	639	DO_NO...	OTHER	true
17	300	true	true	1	2	600	0	399	640	DO_NO...	OTHER	false
18	601	false	true	2	1	601	1	399	739	DO_NO...	TCAS_TA	true
19	299	false	true	1	2	600	2	399	740	NO_INT...	OTHER	false
20	300	false	false	2	1	601	3	399	840	NO_INT...	TCAS_TA	true
21	601	true	false	2	1	601	1	400	0	DO_NO...	OTHER	true
22	299	false	true	1	2	600	0	400	399	NO_INT...	TCAS_TA	false
23	300	*	*	*	*	*	3	400	400	DO_NO...	TCAS_TA	*
24	601	*	*	*	*	*	2	400	499	NO_INT...	*	*
25	299	*	*	*	*	*	1	400	500	NO_INT...	*	*
26	300	*	*	*	*	*	0	400	639	DO_NO...	*	*
27	601	*	*	*	*	*	3	400	640	DO_NO...	*	*
28	299	*	*	*	*	*	2	400	739	DO_NO...	*	*
29	300	*	*	*	*	*	1	400	740	DO_NO...	*	*
30	601	*	*	*	*	*	0	400	840	DO_NO...	*	*
31	299	true	true	1	1	600	3	499	0	NO_INT...	OTHER	true
32	300	false	false	2	2	601	2	499	399	DO_NO...	TCAS_TA	false

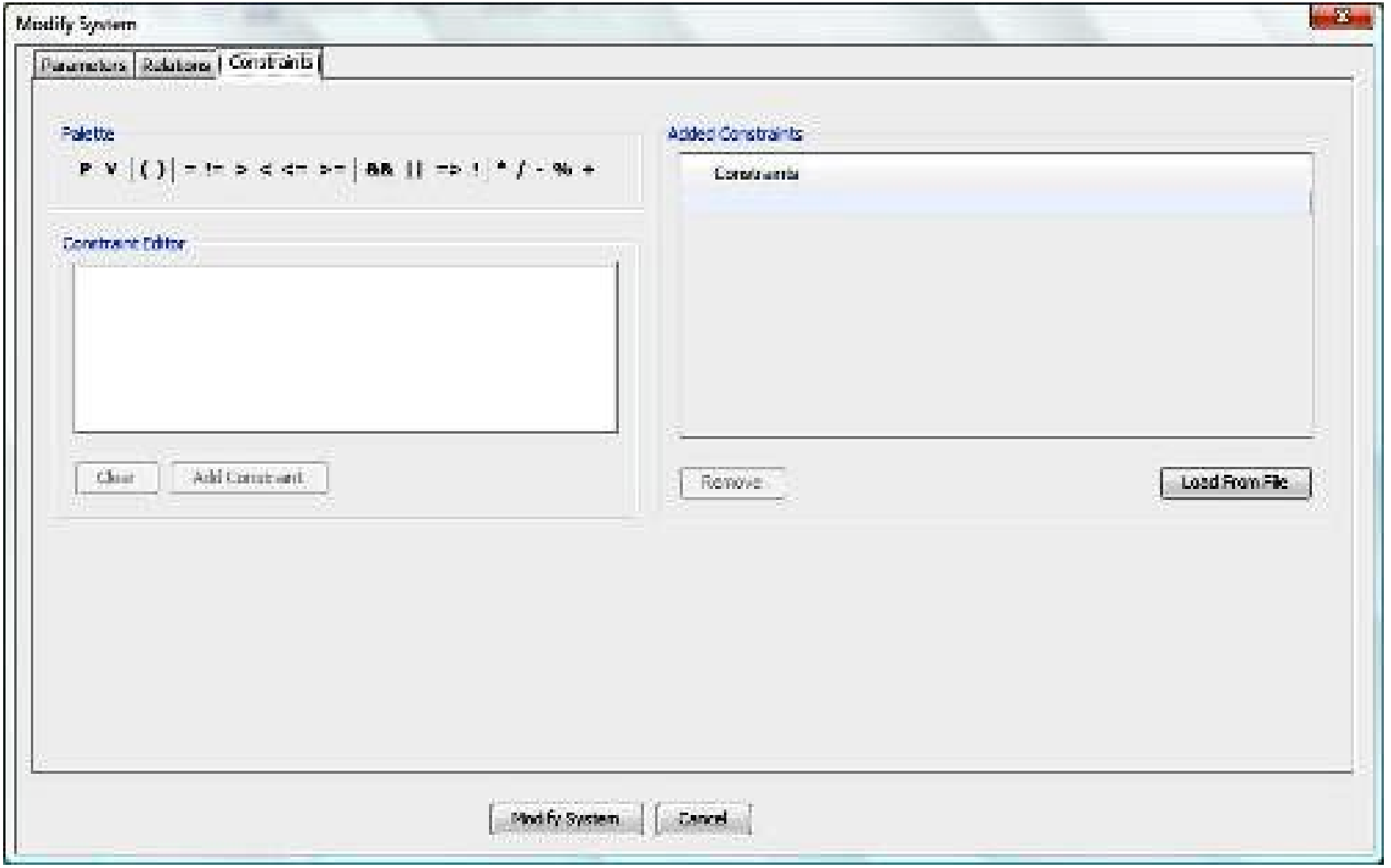
Defining a new system



Variable interaction strength

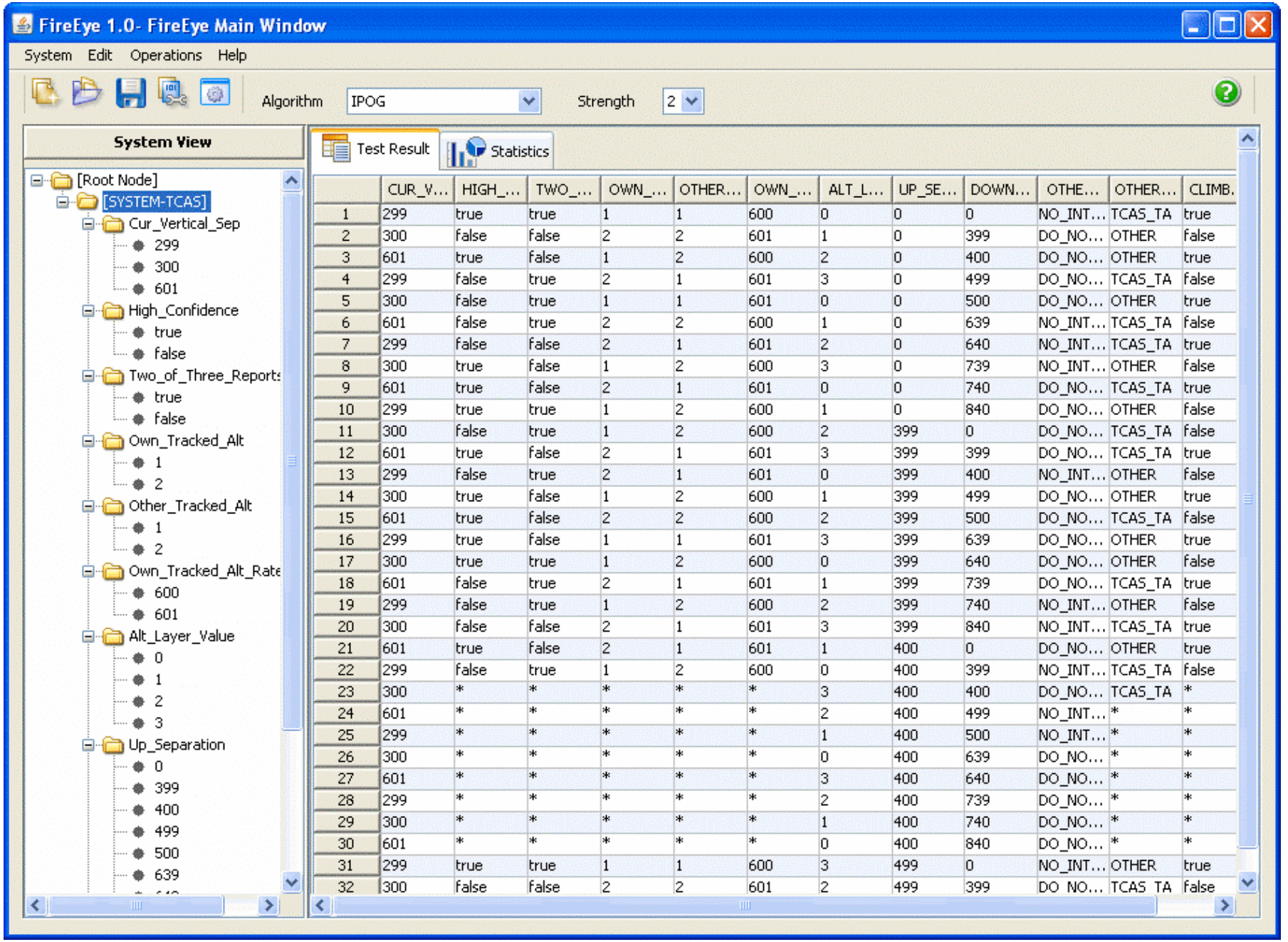


Constraints



Covering array output

National Institute of Standards and Technology



	CUR_V...	HIGH_...	TWO_...	OWN_...	OTHER...	OWN_...	ALT_L...	UP_SE...	DOWN...	OTHE...	OTHER...	CLIMB.
1	299	true	true	1	1	600	0	0	0	NO_INT...	TCAS_TA	true
2	300	false	false	2	2	601	1	0	399	DO_NO...	OTHER	false
3	601	true	false	1	2	600	2	0	400	DO_NO...	OTHER	true
4	299	false	true	2	1	601	3	0	499	DO_NO...	TCAS_TA	false
5	300	false	true	1	1	601	0	0	500	DO_NO...	OTHER	true
6	601	false	true	2	2	600	1	0	639	NO_INT...	TCAS_TA	false
7	299	false	false	2	1	601	2	0	640	NO_INT...	TCAS_TA	true
8	300	true	false	1	2	600	3	0	739	NO_INT...	OTHER	false
9	601	true	false	2	1	601	0	0	740	DO_NO...	TCAS_TA	true
10	299	true	true	1	2	600	1	0	840	DO_NO...	OTHER	false
11	300	false	true	1	2	600	2	399	0	DO_NO...	TCAS_TA	false
12	601	true	false	2	1	601	3	399	399	DO_NO...	TCAS_TA	true
13	299	false	true	2	1	601	0	399	400	NO_INT...	OTHER	false
14	300	true	false	1	2	600	1	399	499	DO_NO...	OTHER	true
15	601	true	false	2	2	600	2	399	500	DO_NO...	TCAS_TA	false
16	299	true	false	1	1	601	3	399	639	DO_NO...	OTHER	true
17	300	true	true	1	2	600	0	399	640	DO_NO...	OTHER	false
18	601	false	true	2	1	601	1	399	739	DO_NO...	TCAS_TA	true
19	299	false	true	1	2	600	2	399	740	NO_INT...	OTHER	false
20	300	false	false	2	1	601	3	399	840	NO_INT...	TCAS_TA	true
21	601	true	false	2	1	601	1	400	0	DO_NO...	OTHER	true
22	299	false	true	1	2	600	0	400	399	NO_INT...	TCAS_TA	false
23	300	*	*	*	*	*	3	400	400	DO_NO...	TCAS_TA	*
24	601	*	*	*	*	*	2	400	499	NO_INT...	*	*
25	299	*	*	*	*	*	1	400	500	NO_INT...	*	*
26	300	*	*	*	*	*	0	400	639	DO_NO...	*	*
27	601	*	*	*	*	*	3	400	640	DO_NO...	*	*
28	299	*	*	*	*	*	2	400	739	DO_NO...	*	*
29	300	*	*	*	*	*	1	400	740	DO_NO...	*	*
30	601	*	*	*	*	*	0	400	840	DO_NO...	*	*
31	299	true	true	1	1	600	3	499	0	NO_INT...	OTHER	true
32	300	false	false	2	2	601	2	499	399	DO_NO...	TCAS_TA	false

Output

NIST National Institute of Standards and Technology

- 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

NIST National Institute of Standards and Technology

Mappable values

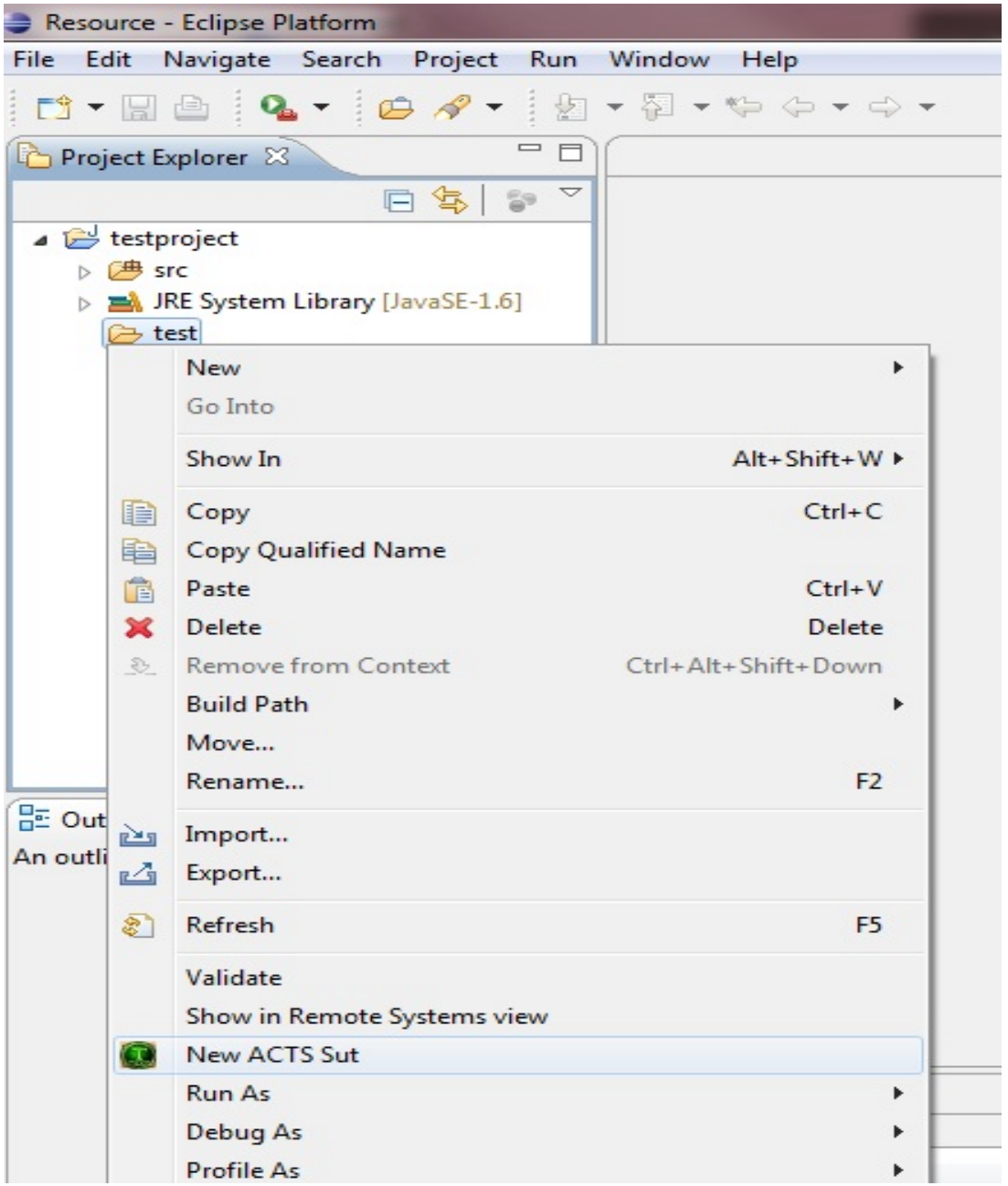
```
Degree of interaction
coverage: 2
Number of parameters: 12
Number of tests: 100

-------------------------------

0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 1 1 1 1
2 0 1 0 1 0 2 0 2 2 1 0
0 1 0 1 0 1 3 0 3 1 0 1
1 1 0 0 0 1 0 0 4 2 1 0
2 1 0 1 1 0 1 0 5 0 0 1
0 1 1 1 0 1 2 0 6 0 0 0
1 0 1 0 1 0 3 0 7 0 1 1
2 0 1 1 0 1 0 0 8 1 0 0
0 0 0 0 1 0 1 0 9 2 1 1
1 1 0 0 1 0 2 1 0 1 0 1
```

Etc.

Human readable

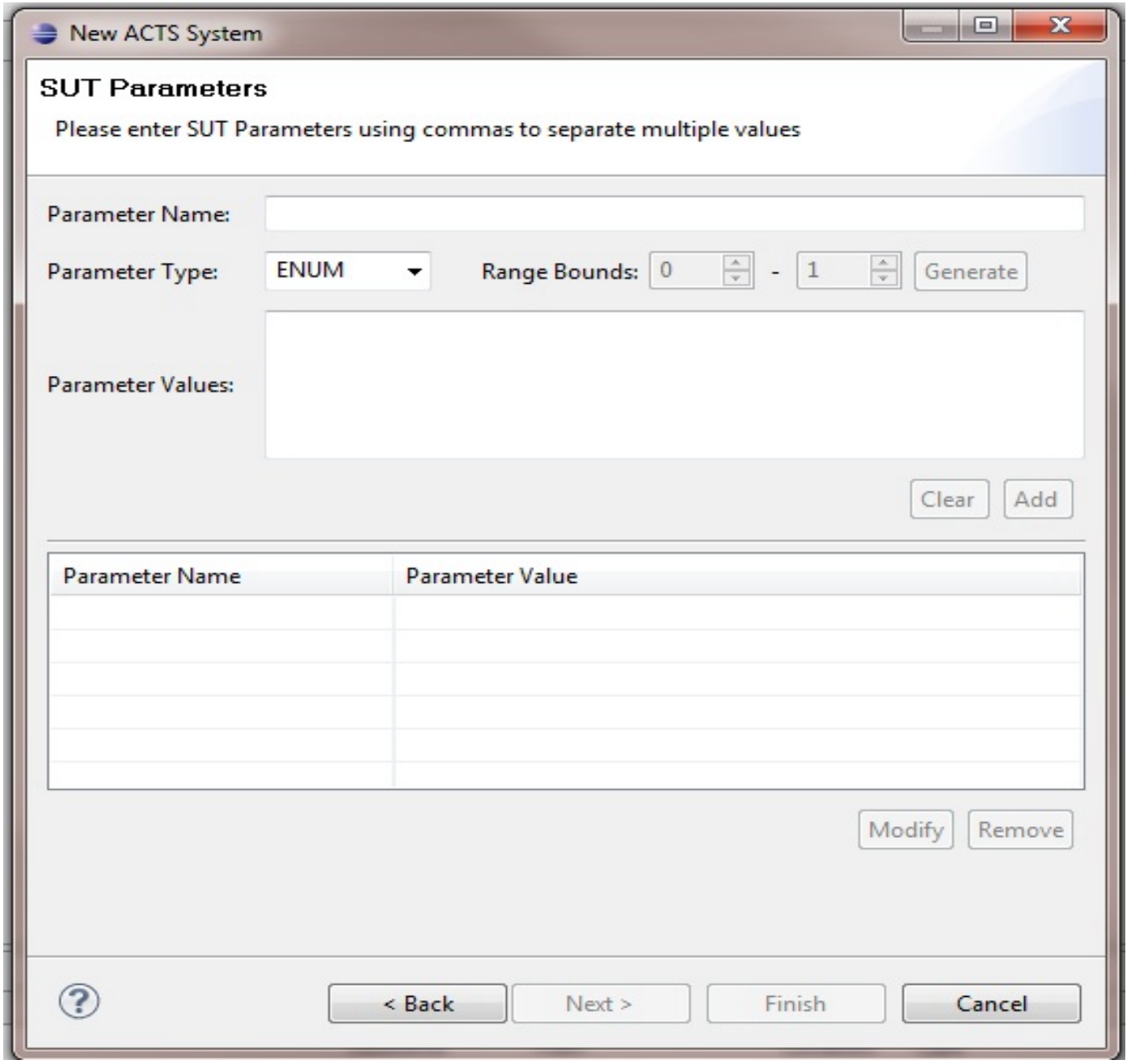
```
Degree of interaction coverage: 2
Number of parameters: 12
Maximum number of values per
parameter: 10
Number of configurations: 100
-----------------------------------
Configuration #1:

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

Eclipse Plugin for ACTS



Work in progress

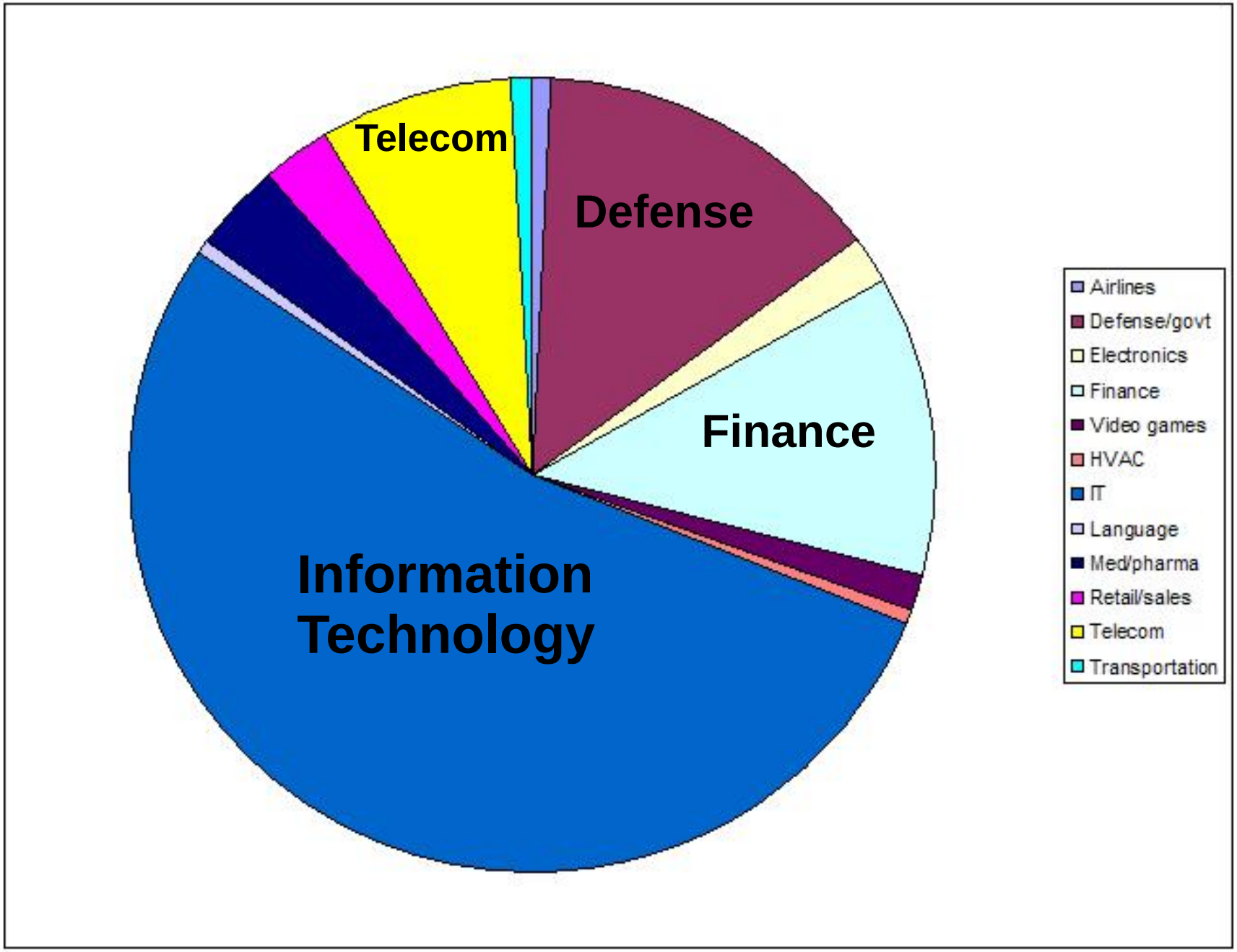
Eclipse Plugin for ACTS



Defining parameters and values

ACTS Users

NIST National Institute of Standards and Technology



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

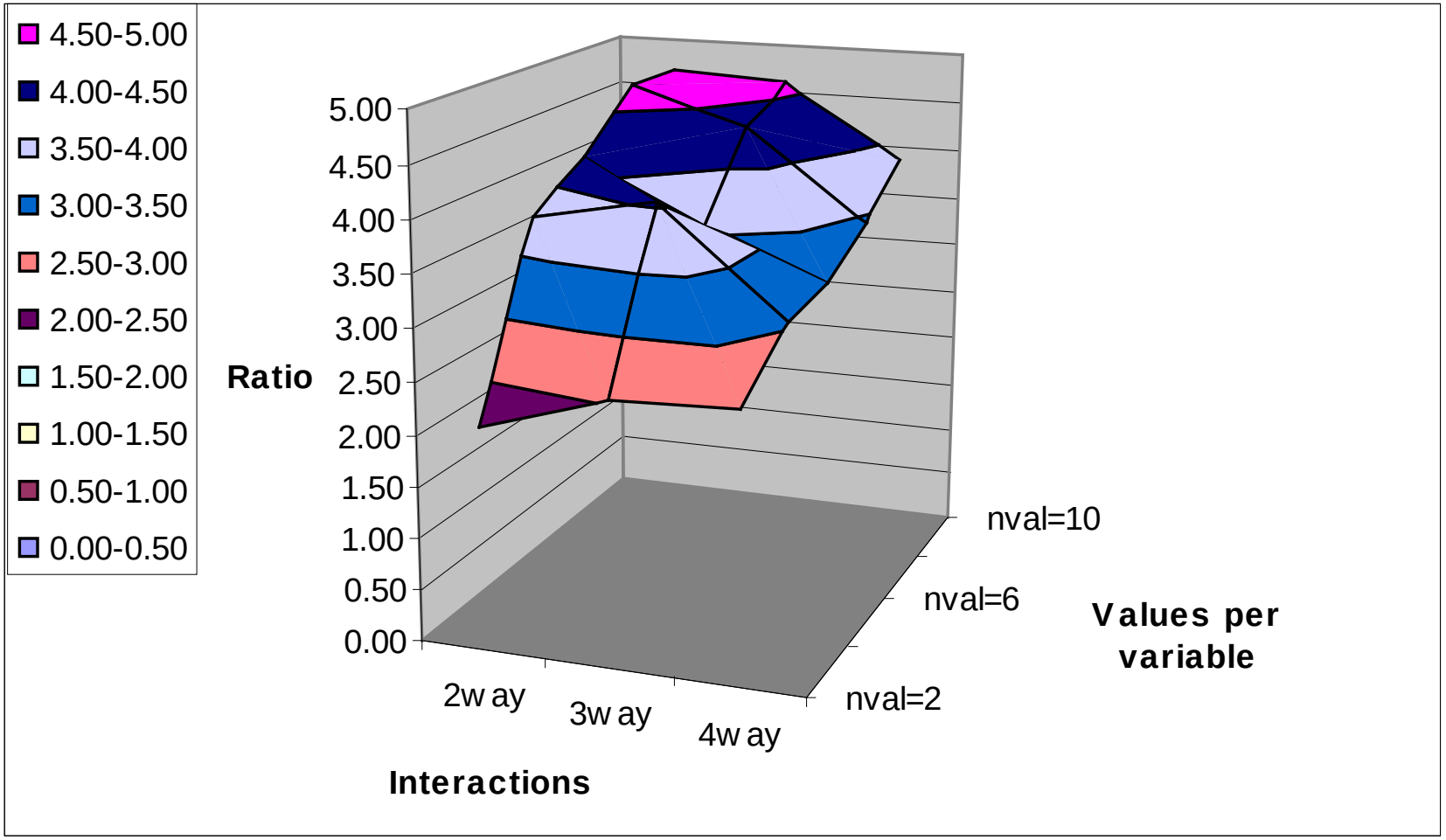
- Like “fuzz testing” - send packets or other input to application, watch for crashes
- Unlike fuzz testing, input is non-random; cover all t-way combinations
- May be more efficient - random input generation requires several times as many tests to cover the t-way combinations in a covering array

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

- buffer overflows
- server crashes

NIST National Institute of Standards and Technology

Ratio of Random/Combinatorial Test Set Required to Provide t-way Coverage



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:

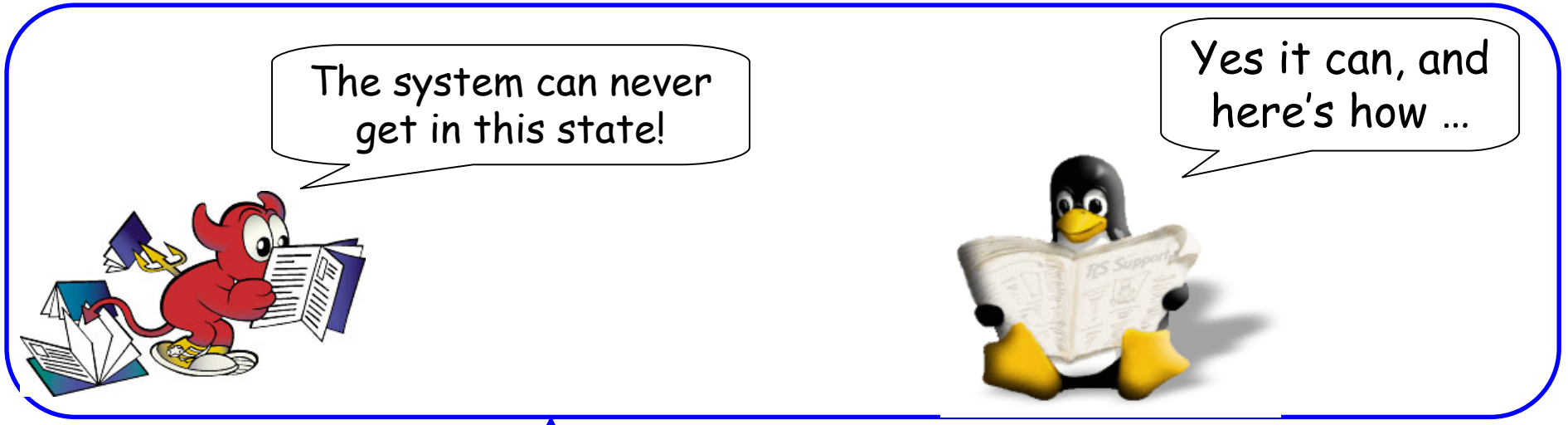
```
ensures balance  == \old(balance) - amount
  &&  \result == balance;
```

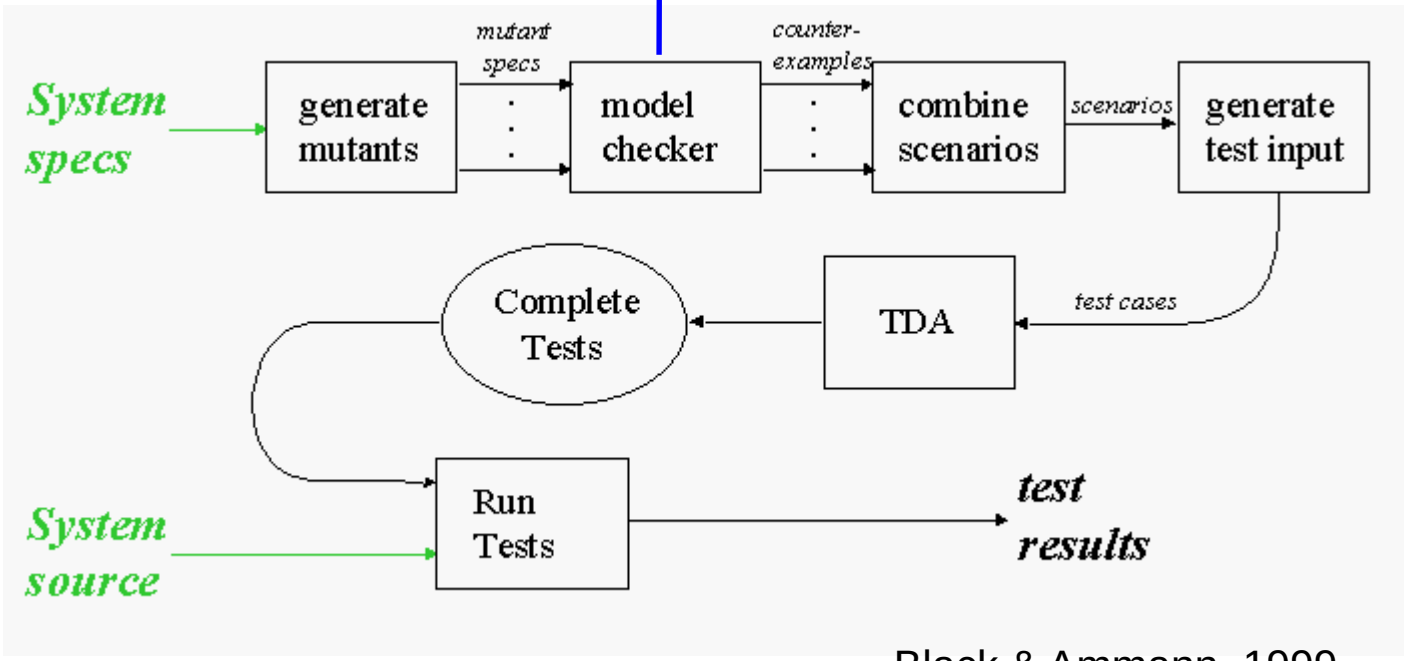
- Reasonable assurance that code works correctly across the range of expected inputs

- 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





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

- This can be converted to a test case.

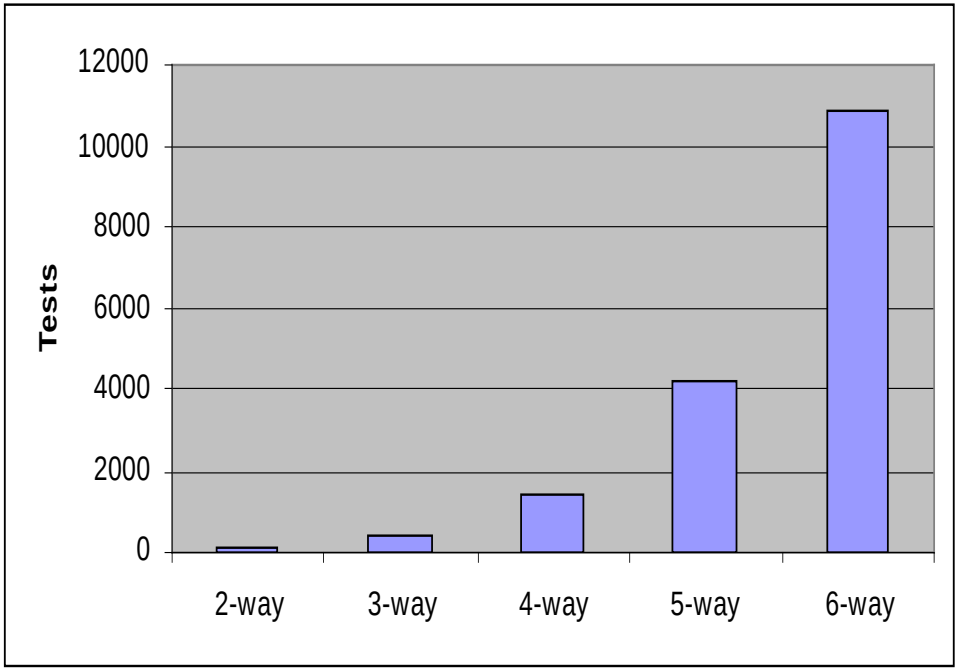
Black & Ammann, 1999

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

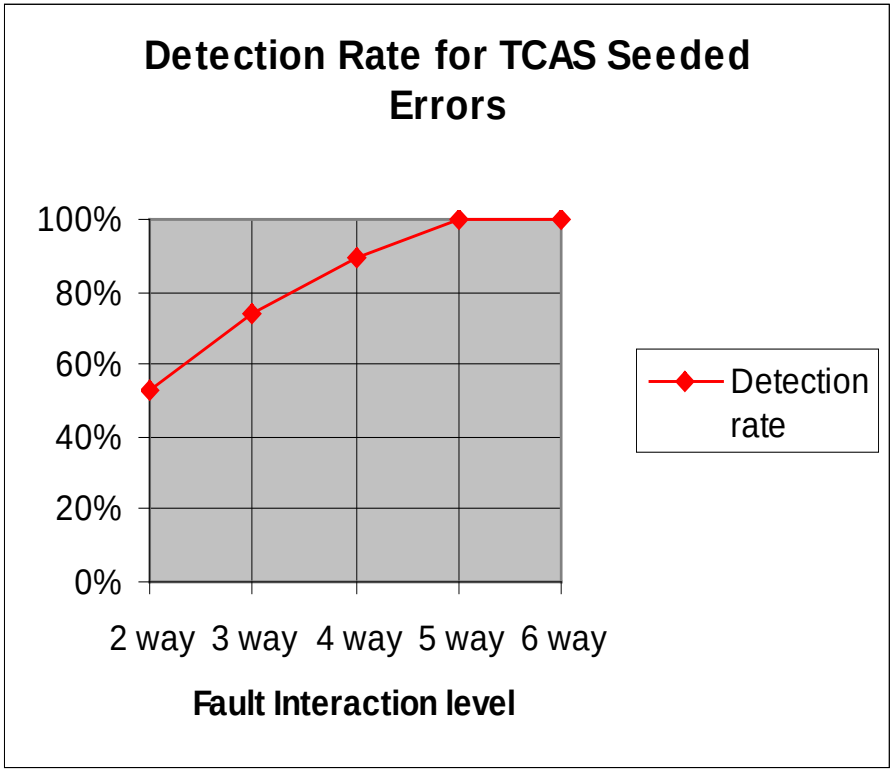
Tests generated

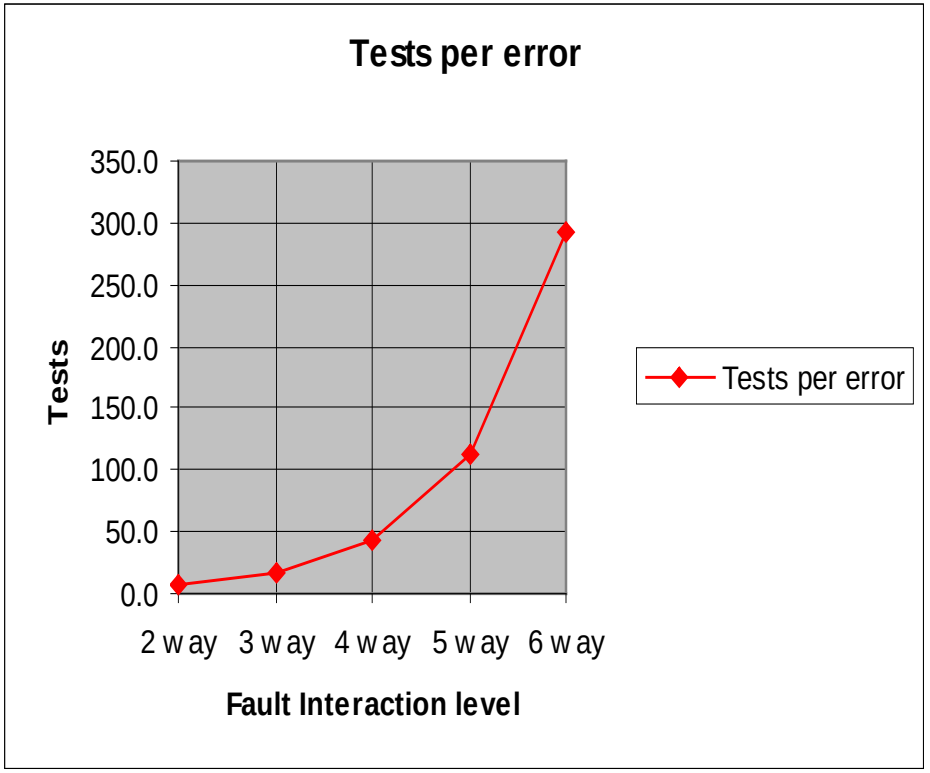
t	Test cases
2-way:	156
3-way:	461
4-way:	1,450
5-way:	4,309
6-way:	11,094



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

Event Name	Param.	Tests
Abort	3	12
Blur	5	24
Click	15	4352
Change	3	12
dblClick	15	4352
DOMActivate	5	24
DOMAttrModified	8	16
DOMCharacterDataModified	8	64
DOMElementNameChanged	6	8
DOMFocusIn	5	24
DOMFocusOut	5	24
DOMNodeInserted	8	128
DOMNodeInsertedIntoDocument	8	128
DOMNodeRemoved	8	128
DOMNodeRemovedFromDocument	8	128
DOMSubTreeModified	8	64
Error	3	12
Focus	5	24
KeyDown	1	17
KeyUp	1	17
Load	3	24
MouseDown	15	4352
MouseMove	15	4352
MouseOut	15	4352
MouseOver	15	4352
MouseUp	15	4352
MouseWheel	14	1024
Reset	3	12
Resize	5	48
Scroll	5	48
Select	3	12
Submit	3	12
TextInput	5	8
Unload	3	24
Wheel	15	4096
Total Tests		36626

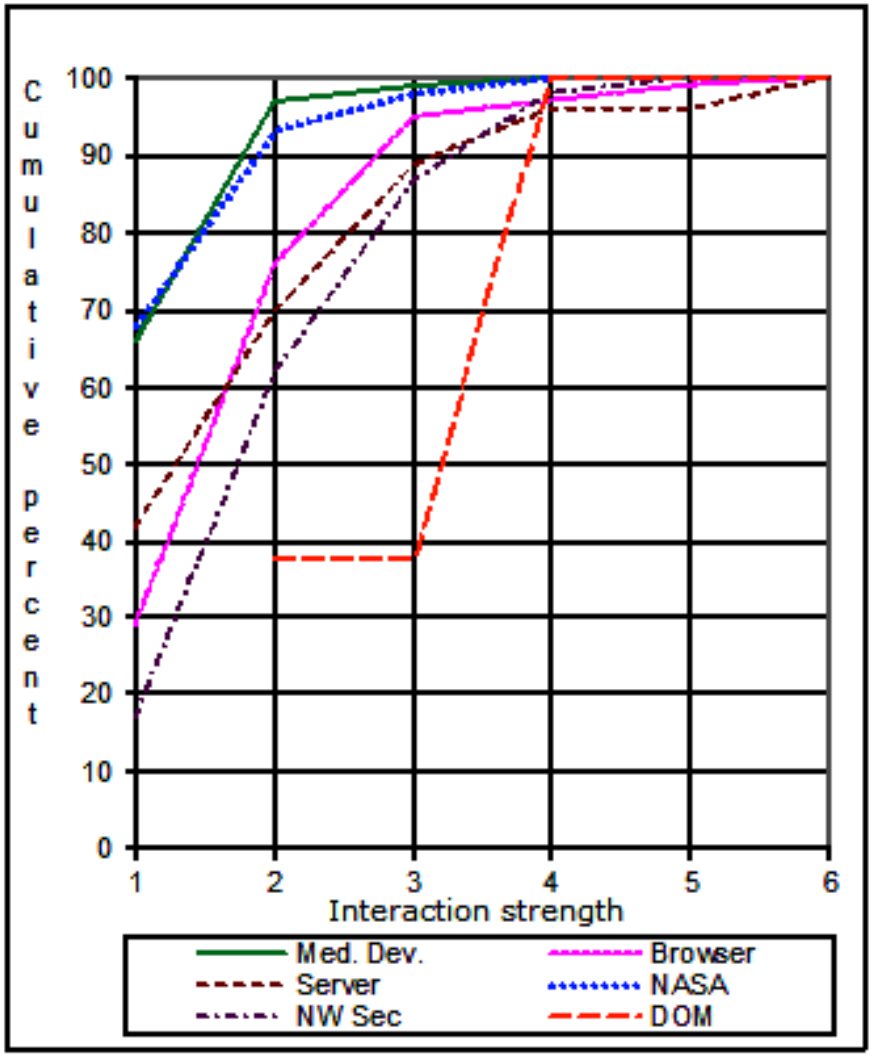
Exhaustive testing of equivalence class values

NIST National Institute of Standards and Technology

World Wide Web Consortium Document Object Model Events

t	Tests	% of Orig.	Test Results		
			Pass	Fail	Not Run
2	702	1.92%	202	27	473
3	1342	3.67%	786	27	529
4	1818	4.96%	437	72	1309
5	2742	7.49%	908	72	1762
6	4227	11.54 %	1803	72	2352

All failures found using < 5% of original pseudo-exhaustive test set



NIST National Institute of Standards and Technology

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

National Institute of Standards and Technology

```
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

NIST National Institute of Standards and Technology

true branch

```
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

National Institute of Standards and Technology

```
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.   }
```

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)

NIST
National Institute of Standards and Technology

Simulation Input Parameters

Parameter		Values
1	DIMENSIONS	1,2,4,6,8
2	NODOSDIM	2,4,6
3	NUMVIRT	1,2,3,8
4	NUMVIRTINJ	1,2,3,8
5	NUMVIRTEJE	1,2,3,8
6	LONBUFFER	1,2,4,6
7	NUMDIR	1,2
8	FORWARDING	0,1
9	PHYSICAL	true, false
10	ROUTING	0,1,2,3
11	DELFIFO	1,2,4,6
12	DELCROSS	1,2,4,6
13	DELCHANNEL	1,2,4,6
14	DELSWITCH	1,2,4,6

5x3x4x4x4x4x2x2
x2x4x4x4x4x4
= 31,457,280
configurations

Are any of them dangerous?

If so, how many?

Which ones?

NIST
National Institute of Standards and Technology

Network Deadlock Detection

Deadlocks Detected: combinatorial

t	Tests	500 pkts	1000 pkts	2000 pkts	4000 pkts	8000 pkts
2	28	0	0	0	0	0
3	161	2	3	2	3	3
4	752	14	14	14	14	14

Average Deadlocks Detected: random

t	Tests	500 pkts	1000 pkts	2000 pkts	4000 pkts	8000 pkts
2	28	0.63	0.25	0.75	0. 50	0. 75
3	161	3	3	3	3	3
4	752	10.13	11.75	10.38	13	13.25

NIST
National Institute of Standards and Technology

Network Deadlock Detection

Detected 14 configurations that can cause deadlock:

$14/ 31,457,280 = 4.4 \times 10^{-7}$

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**

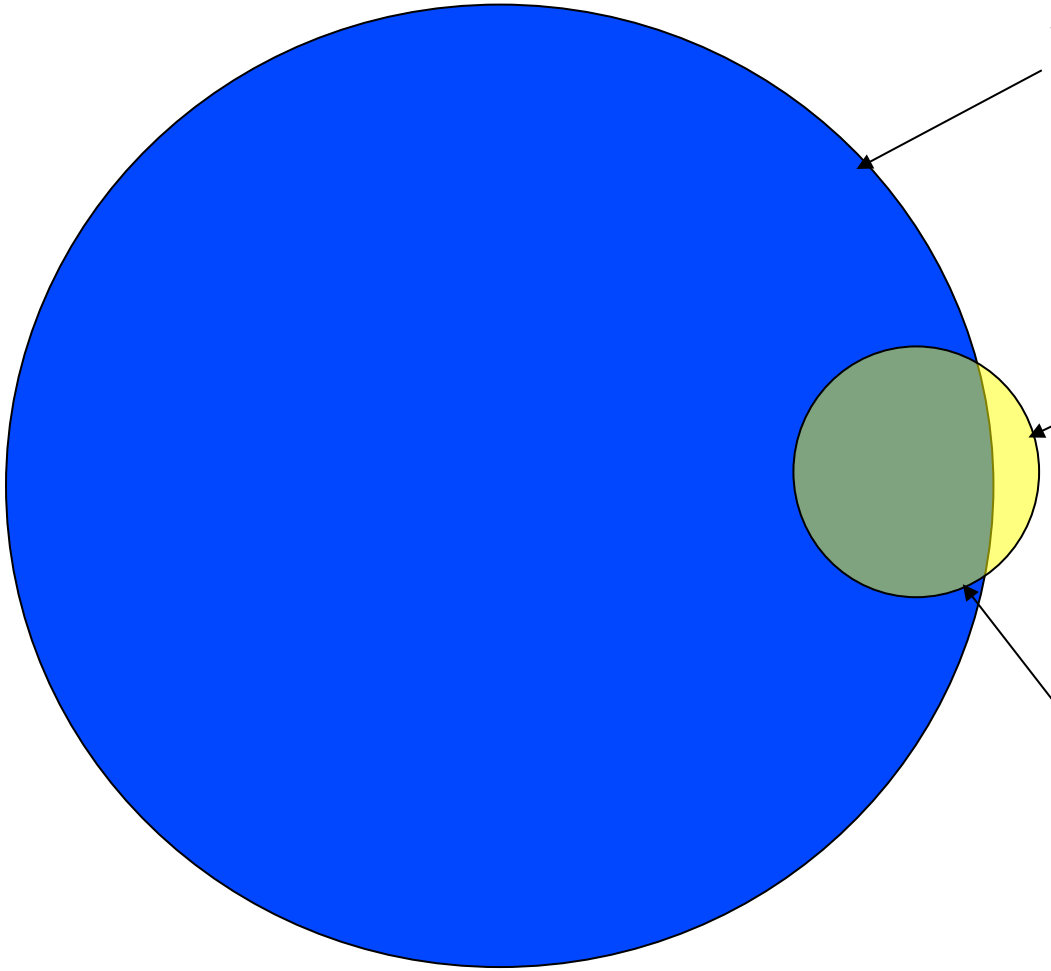
(because they are looking for it)

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?

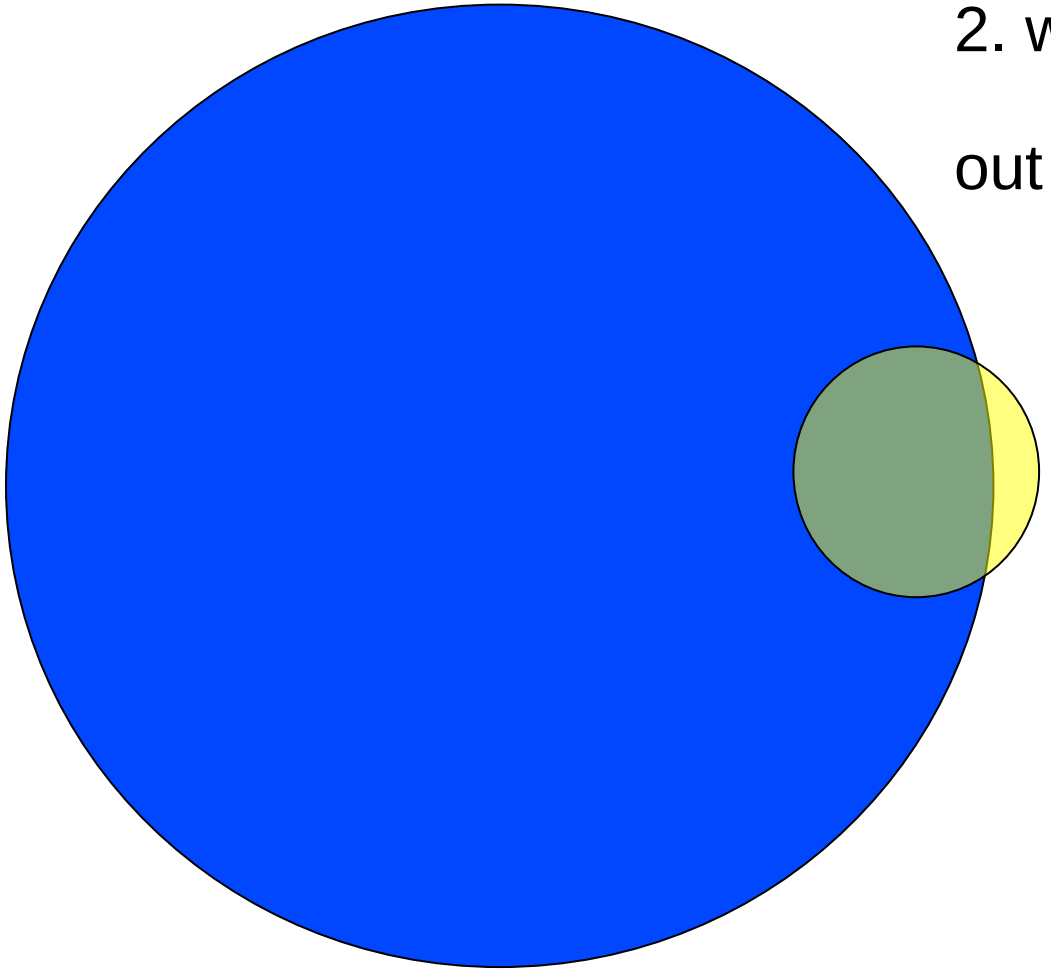


variable/value combinations in passing tests

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 $v^t\binom{n}{t}$ 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

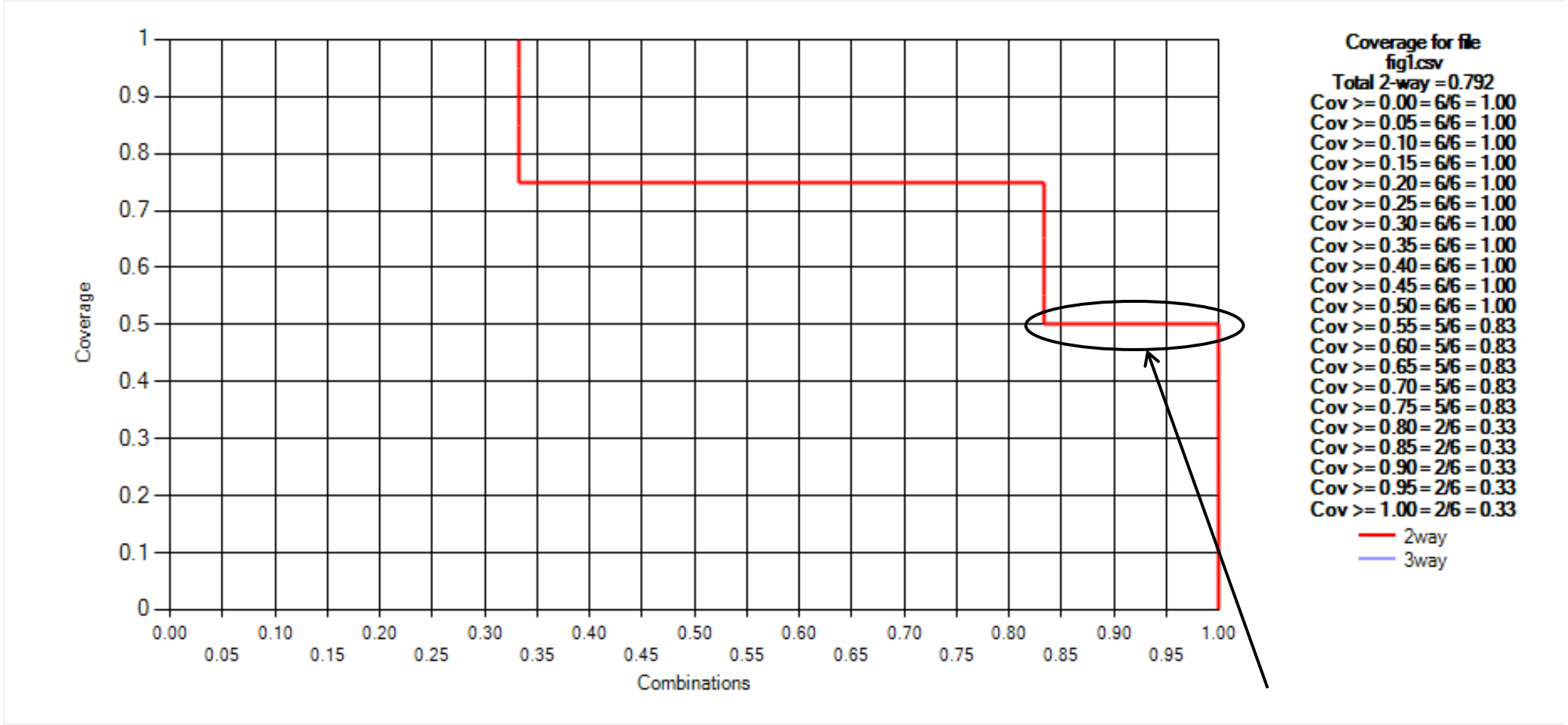
Tests	Variables			
	a	b	c	d
1	0	0	0	0
2	0	1	1	0
3	1	0	0	1
4	0	1	1	1
5	0	1	0	1
6	1	0	1	1
7	1	0	1	0
8	0	1	0	0

Variable pairs	Variable-value combinations covered	Coverage
ab	00, 01, 10	.75
ac	00, 01, 10	.75
ad	00, 01, 11	.75
bc	00, 11	.50
bd	00, 01, 10, 11	1.0
cd	00, 01, 10, 11	1.0

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

NIST
National Institute of Standards and Technology

Graphing Coverage Measurement

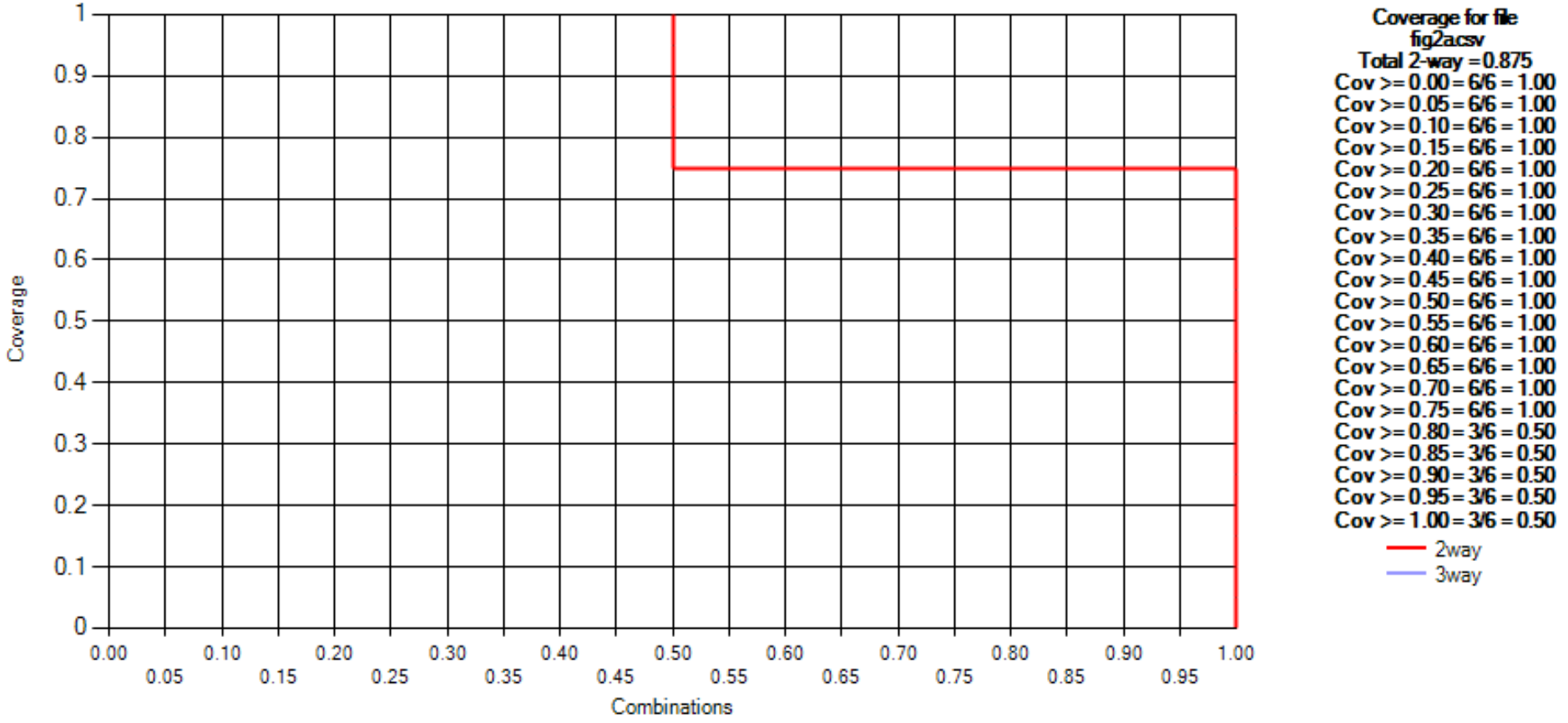


100% coverage of 33% of combinations
75% coverage of half of combinations
50% coverage of 16% of combinations

Bottom line:
All combinations
covered to at least 50%

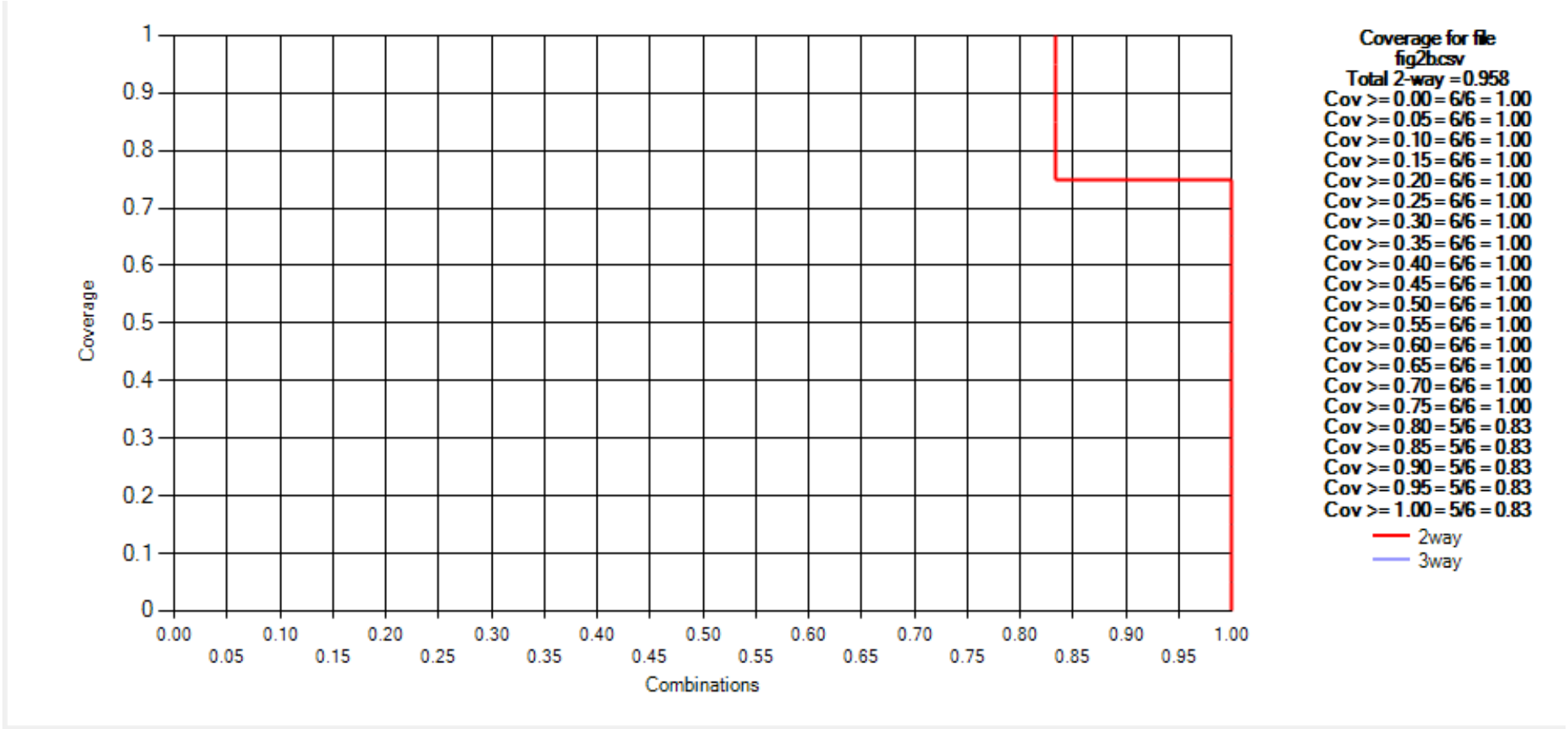
NIST
National Institute of
Standards and Technology

Adding a test



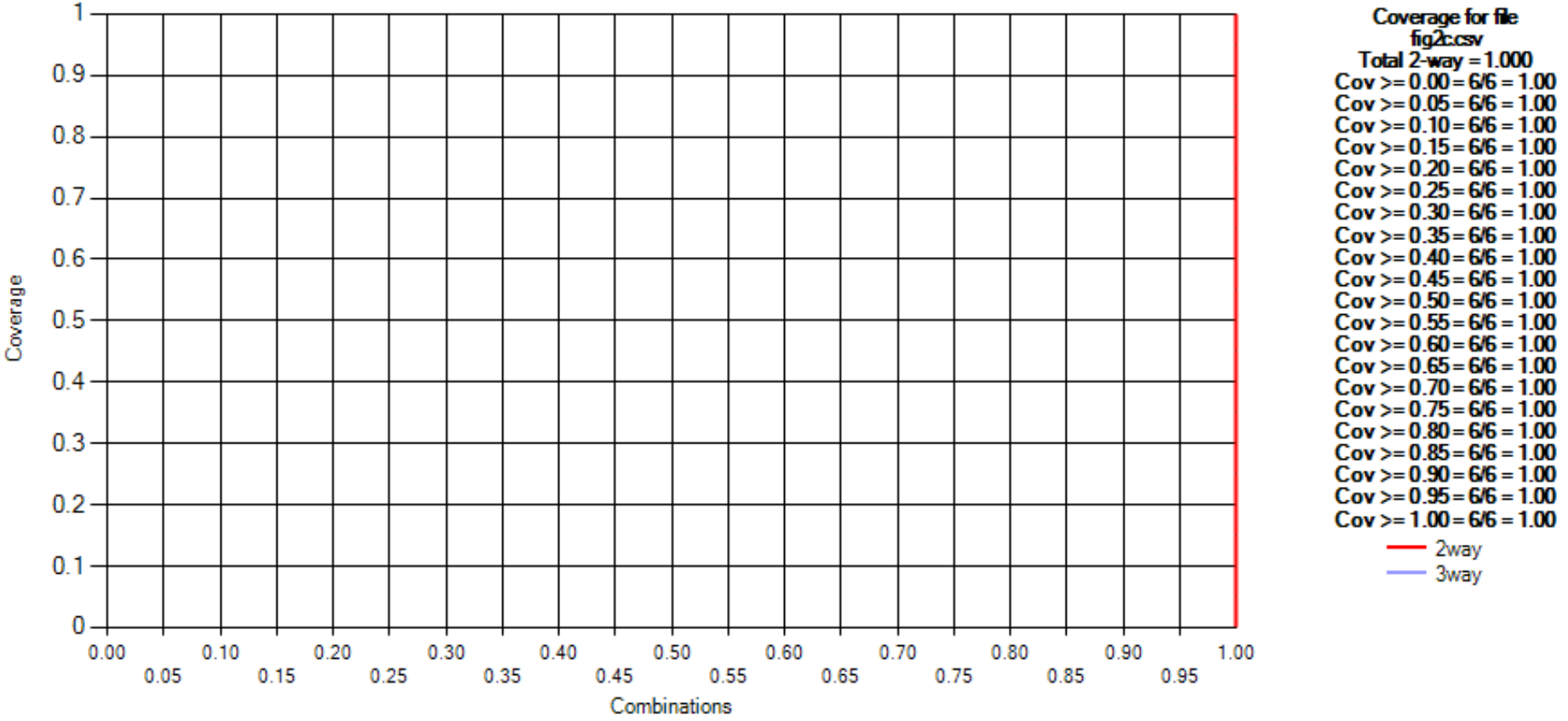
Coverage after adding test [1,1,0,1]

Adding another test



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

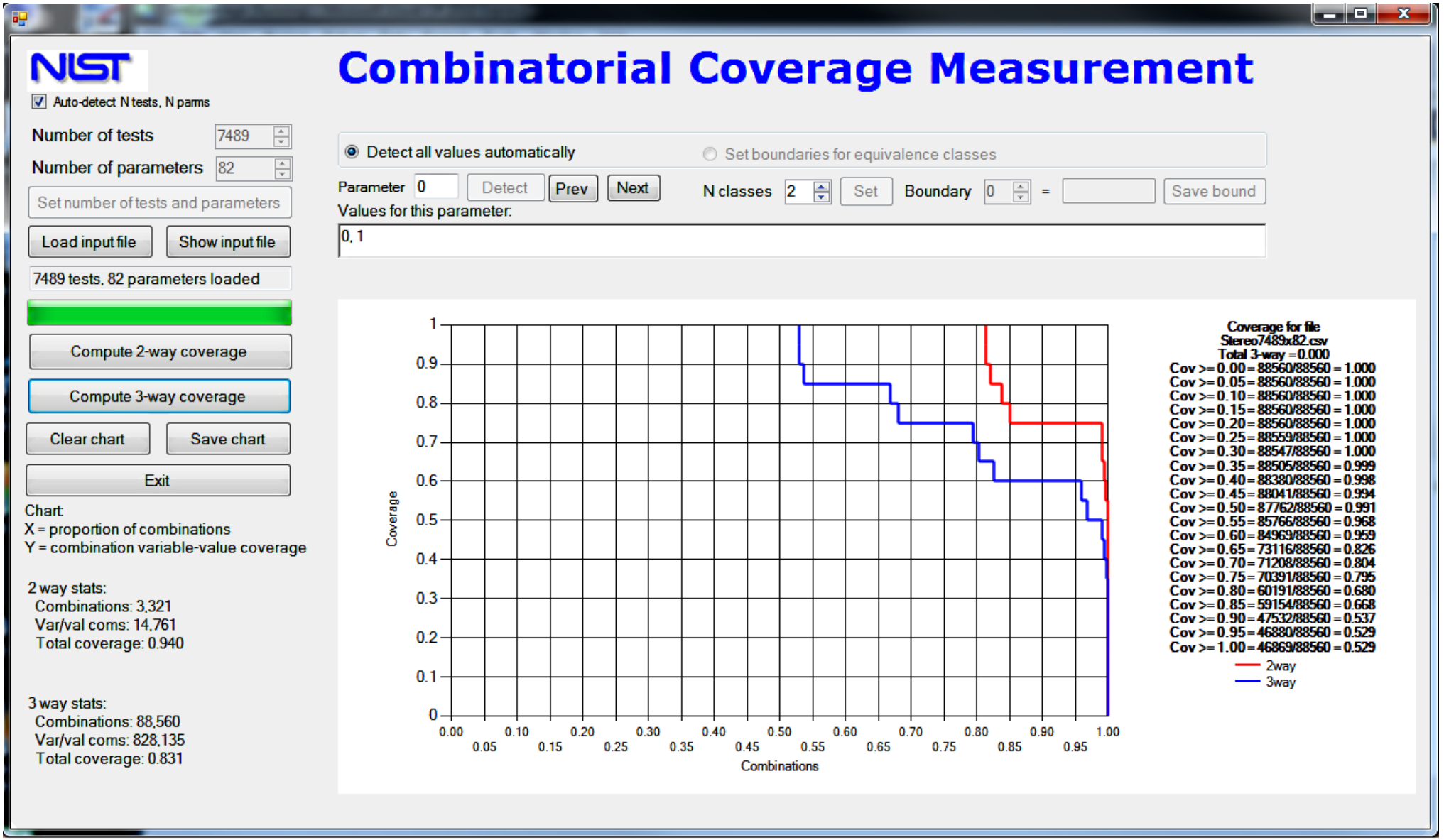
Additional test completes coverage



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

NIST
National Institute of Standards and Technology

Combinatorial Coverage Measurement



NIST
National Institute of Standards and Technology

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)?

Event	Description
a	connect flow meter
b	connect pressure gauge
c	connect satellite link
d	connect pressure readout
e	start comm link
f	boot system

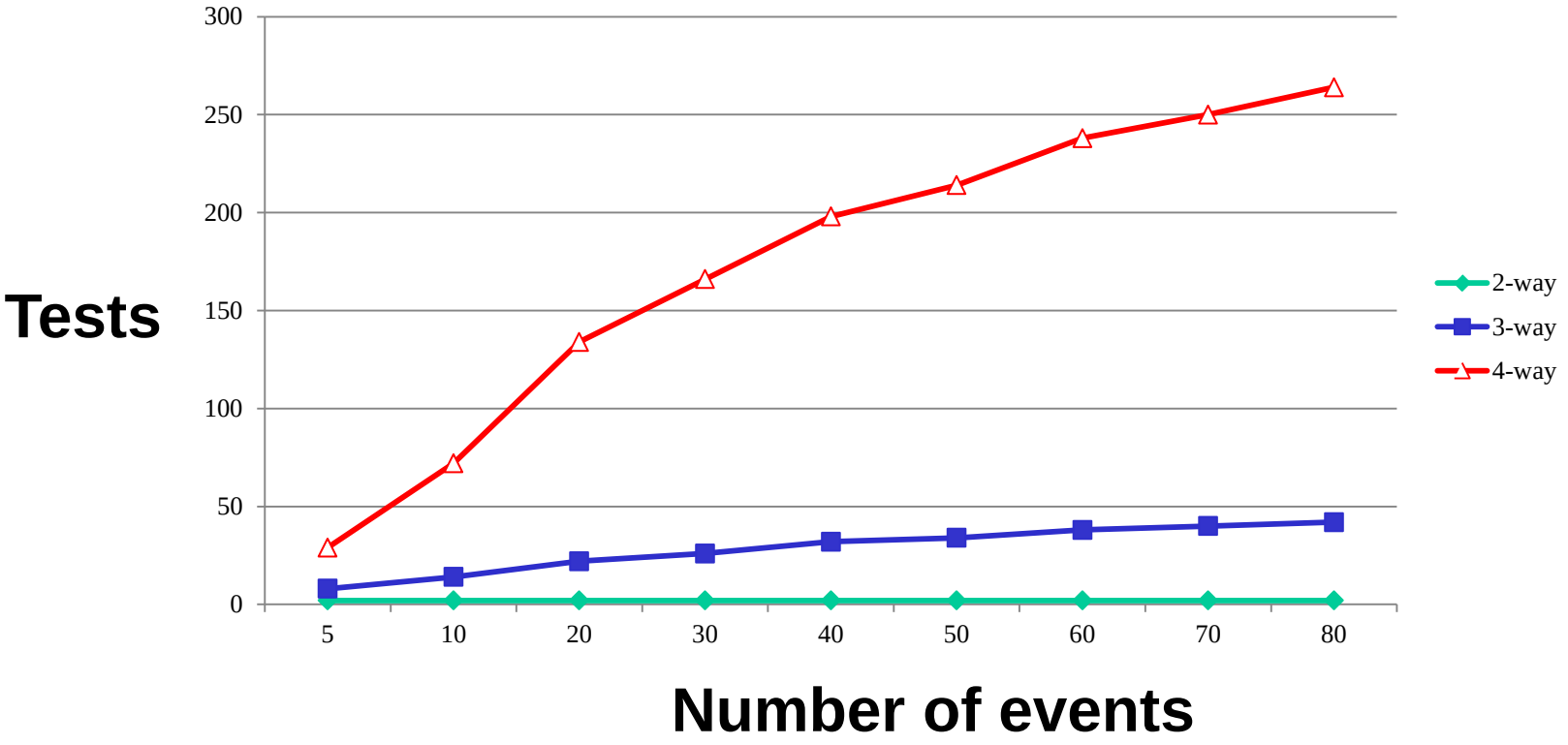
Sequence Covering Array

- 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.

Test	Sequence					
1	a	b	c	d	e	f
2	f	e	d	c	b	a
3	d	e	f	a	b	c
4	c	b	a	f	e	d
5	b	f	a	d	c	e
6	e	c	d	a	f	b
7	a	e	f	c	b	d
8	d	b	c	f	e	a
9	c	e	a	d	b	f
10	f	b	d	a	e	c

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

NIST
National Institute of Standards and Technology

Technology Transfer

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

Collaborations: USAF 46th Test Wing, Lockheed Martin, George Mason Univ., UMBC, JHU/APL, Carnegie Mellon Univ.

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!)