National Institute of
Standards and Technology

Combinatorial Methods in Software Testing

Rick Kuhn

National Institute of
Standards and Technology
Gaithersburg, MD

Conference on Applied Statistics in Defense
October 26, 2016

Overview

1. **Intro, empirical data and fault model**
2. How it works and coverage/cost considerations
3. Practical applications
4. Research topics

What is NIST and why are we doing this?

- US Government agency, whose mission is to support US industry through developing better measurement and test methods
- 3,000 scientists, engineers, and staff including 4 Nobel laureates
- **Project goal – <u>improve cost-benefit ratio for testing</u>**

Why combinatorial testing? - examples

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

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

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

Applications

Software testing – primary application of these methods

- functionality testing and security vulnerabilities
- approx 2/3 of vulnerabilities from implementation faults

Modeling and simulation – ensure coverage of complex cases

- measure coverage of traditional Monte Carlo sim
- faster coverage of input space than randomized input

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

>> systems with a large number of factors that interact <<

What is the empirical basis?

- NIST studied software failures in 15 years of FDA medical device recall data
- What causes software failures?
  - logic errors? calculation errors? inadequate input checking? interaction faults? Etc.

Interaction faults: e.g., failure occurs if

`altitude = 0 && volume < 2.2`

(interaction between 2 factors)

So this is a **2-way interaction**

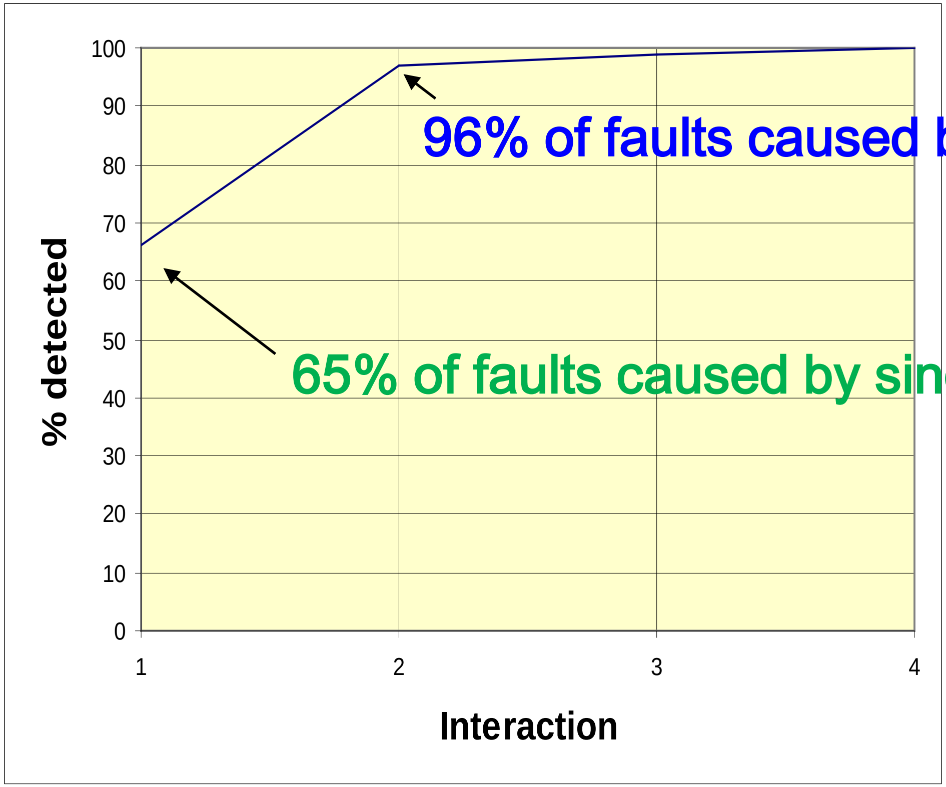
=> testing all pairs of values can find this fault

How are interaction faults distributed?

- Interactions e.g., failure occurs if

pressure < 10	(1-way interaction)
pressure < 10 & volume > 300	(2-way interaction)
pressure < 10 & volume > 300 & velocity = 5	(3-way interaction)

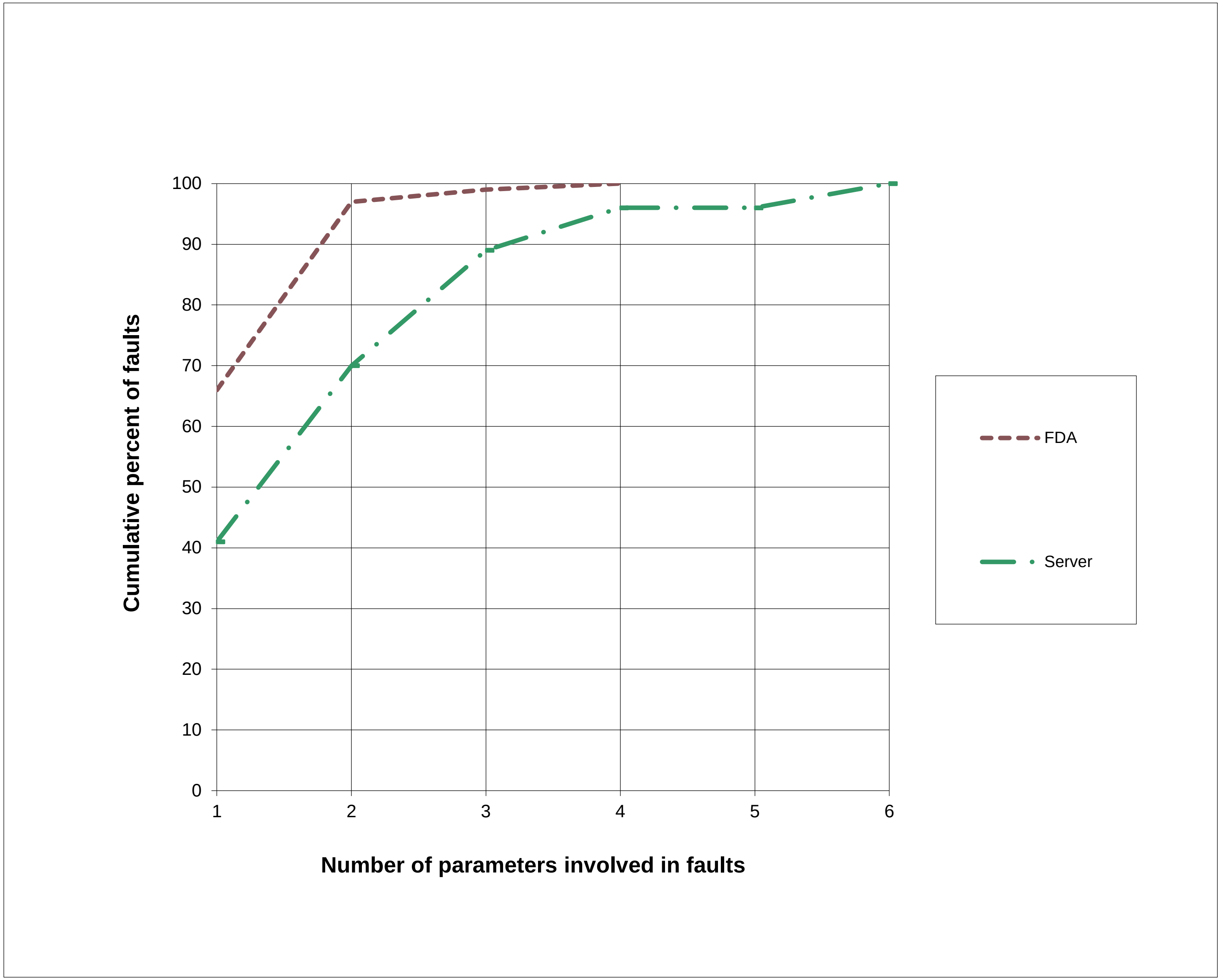
- Surprisingly, no one had looked at interactions > 2-way before



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

65% of faults caused by single factor

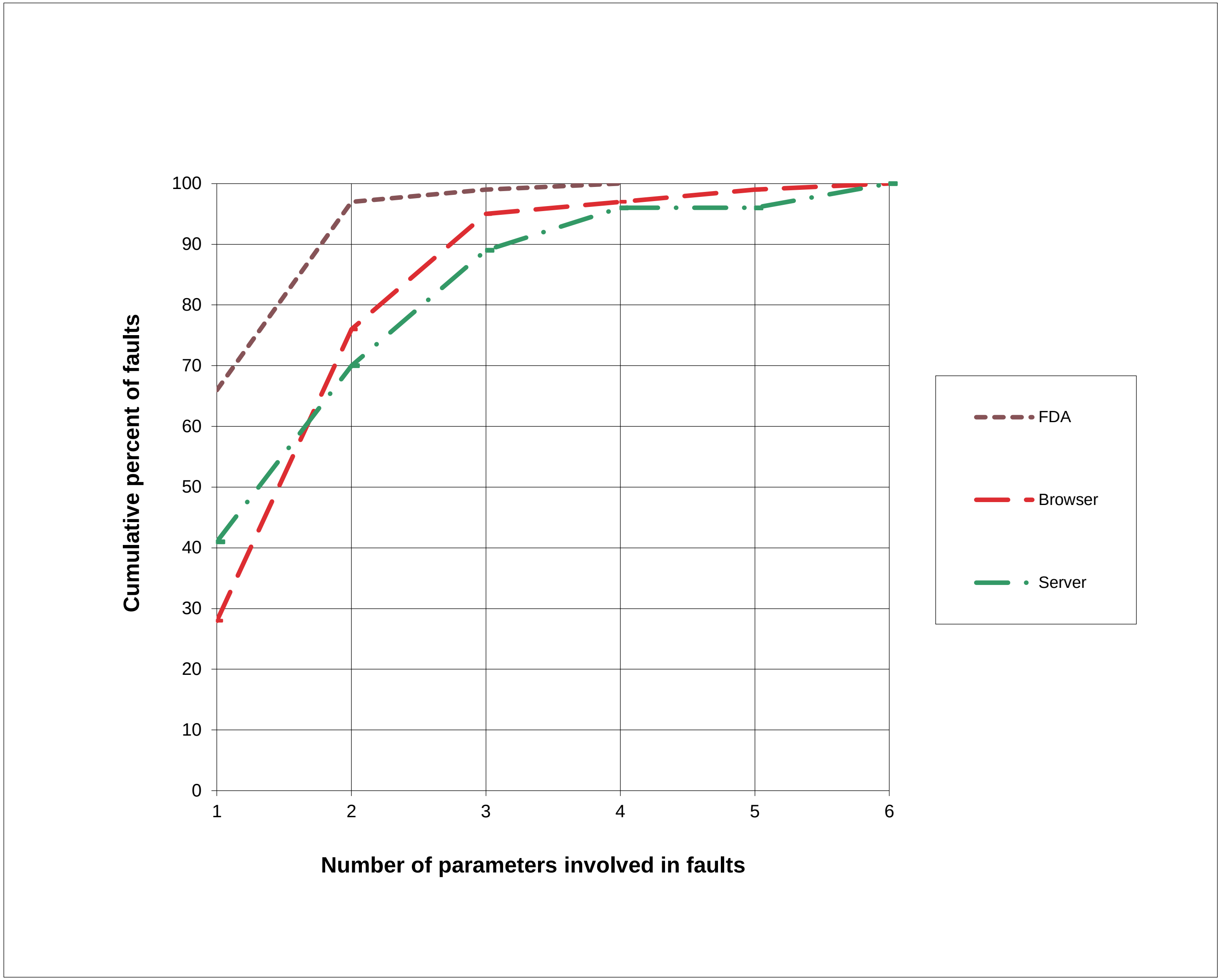
Server



These faults more complex than medical device software!!

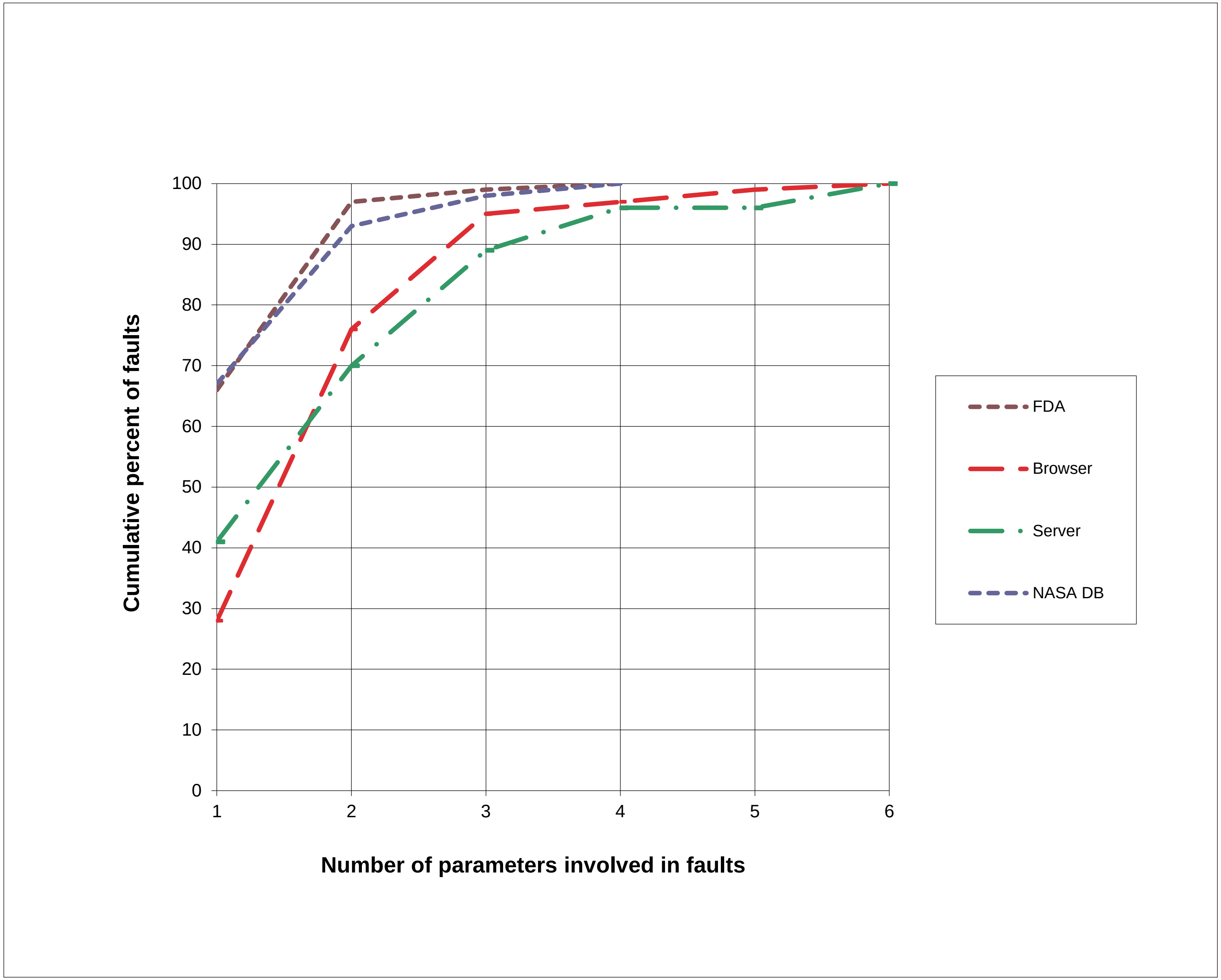
Why?

Browser



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

NASA distributed database

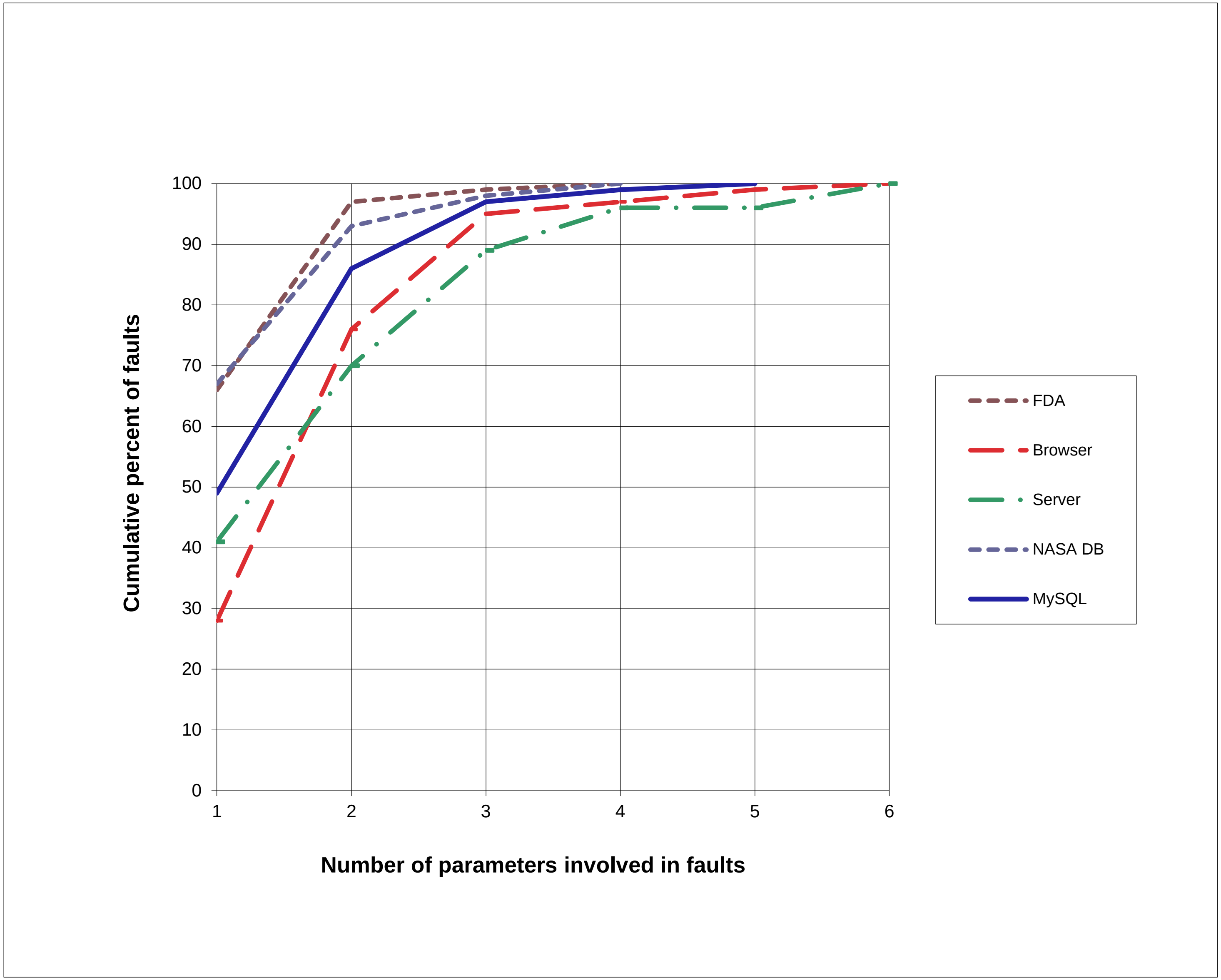


Note: initial testing

but ….

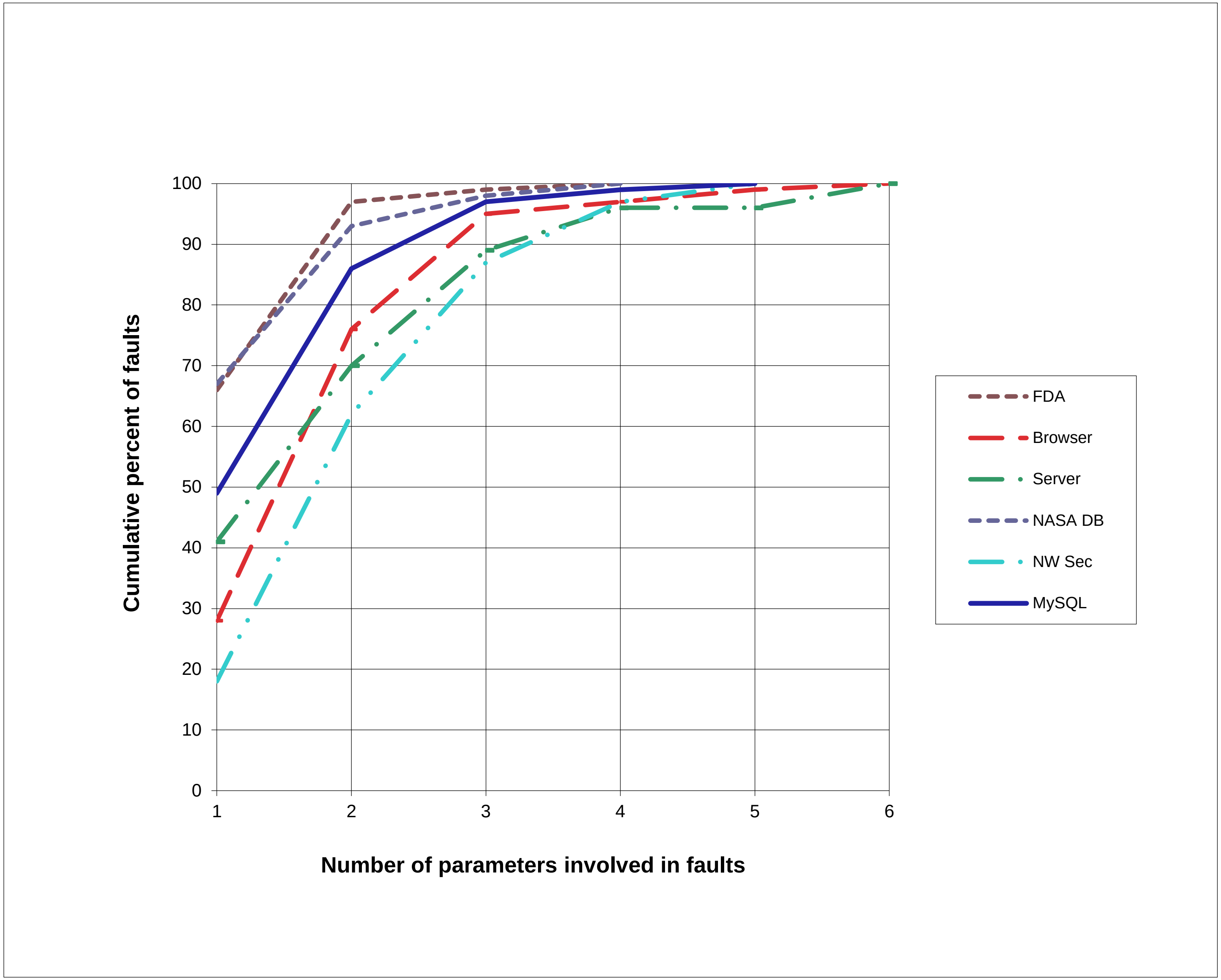
Fault profile better than medical devices!

MySQL



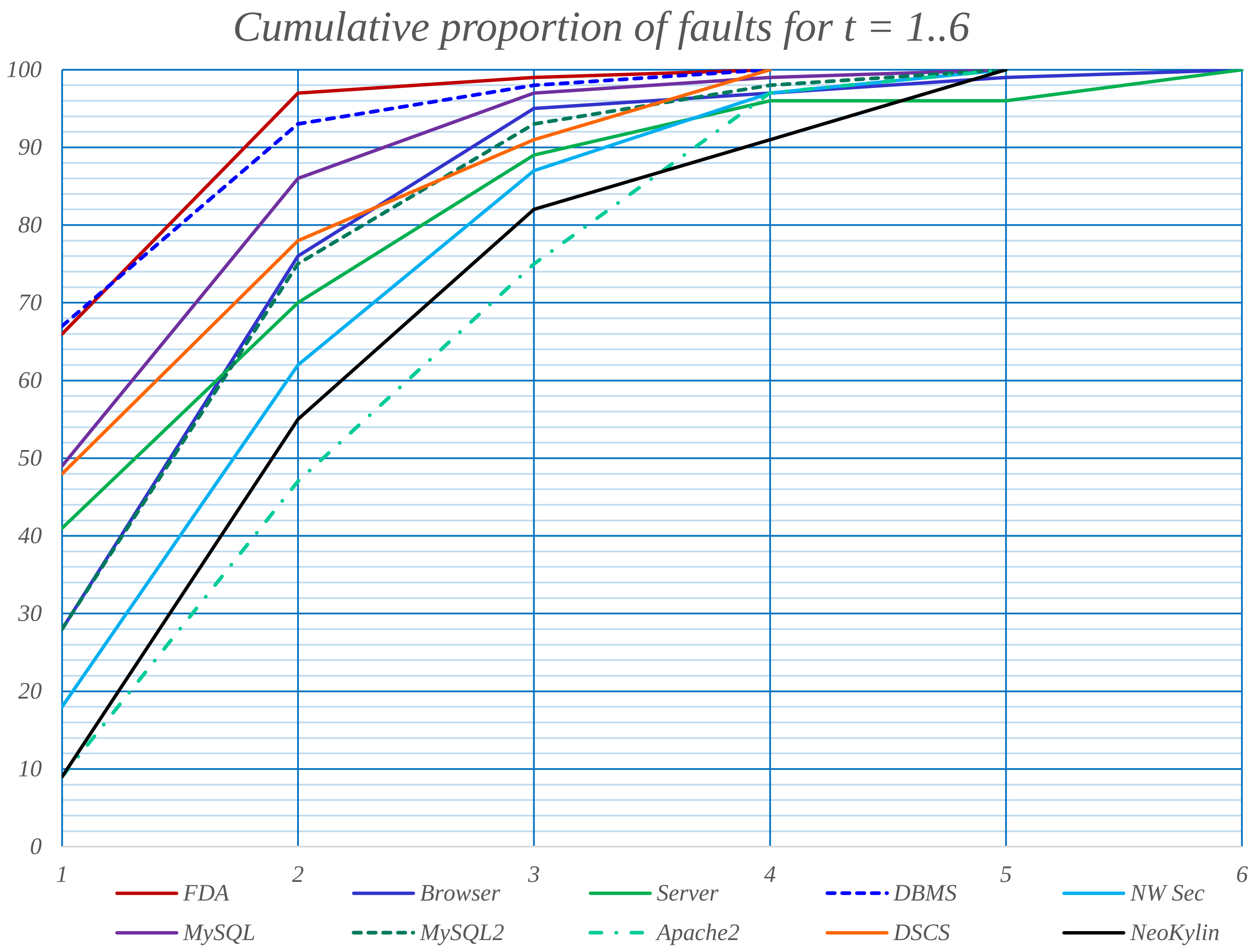
Cumulative percent of faults
100
90
80
70
60
50
40
30
20
10
0
1
2
3
4
5
6
Number of parameters involved in faults
FDA
Browser
Server
NASA DB
MySQL

TCP/IP



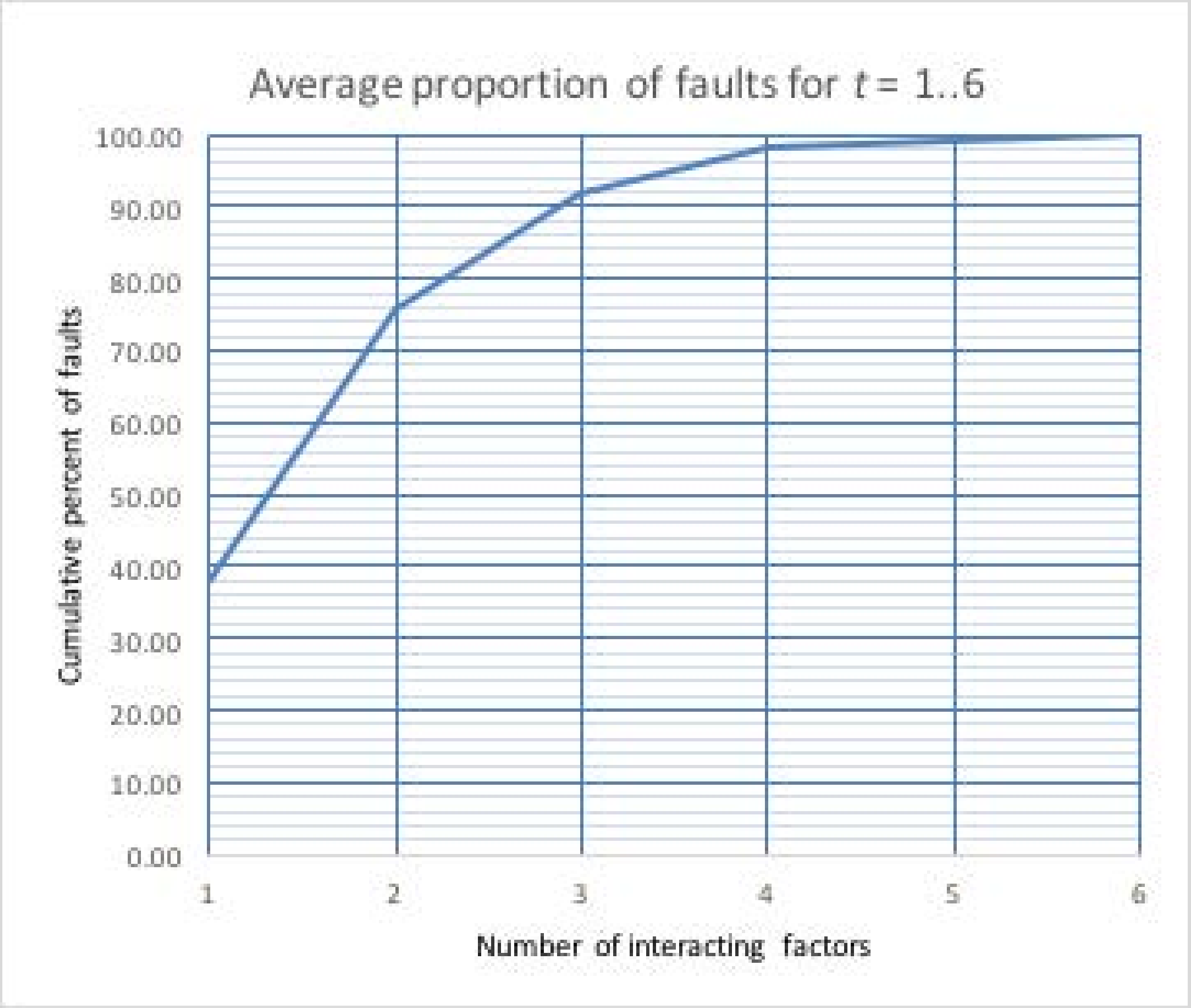
National Institute of Standards and Technology

Wait, there's more

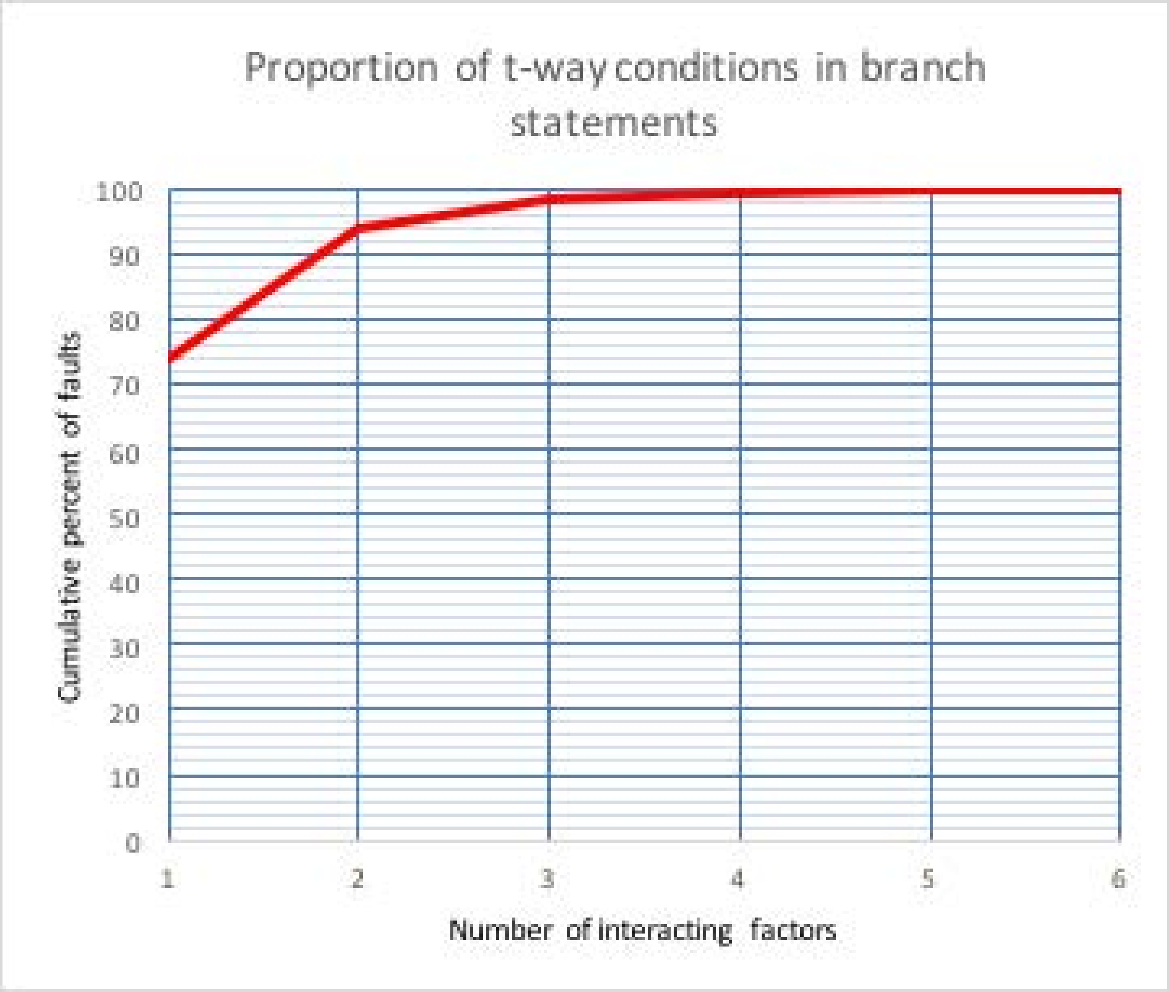


- Number of factors involved in failures is <u>small</u>
- No failure involving more than 6 variables has been seen

Average (unweighted)

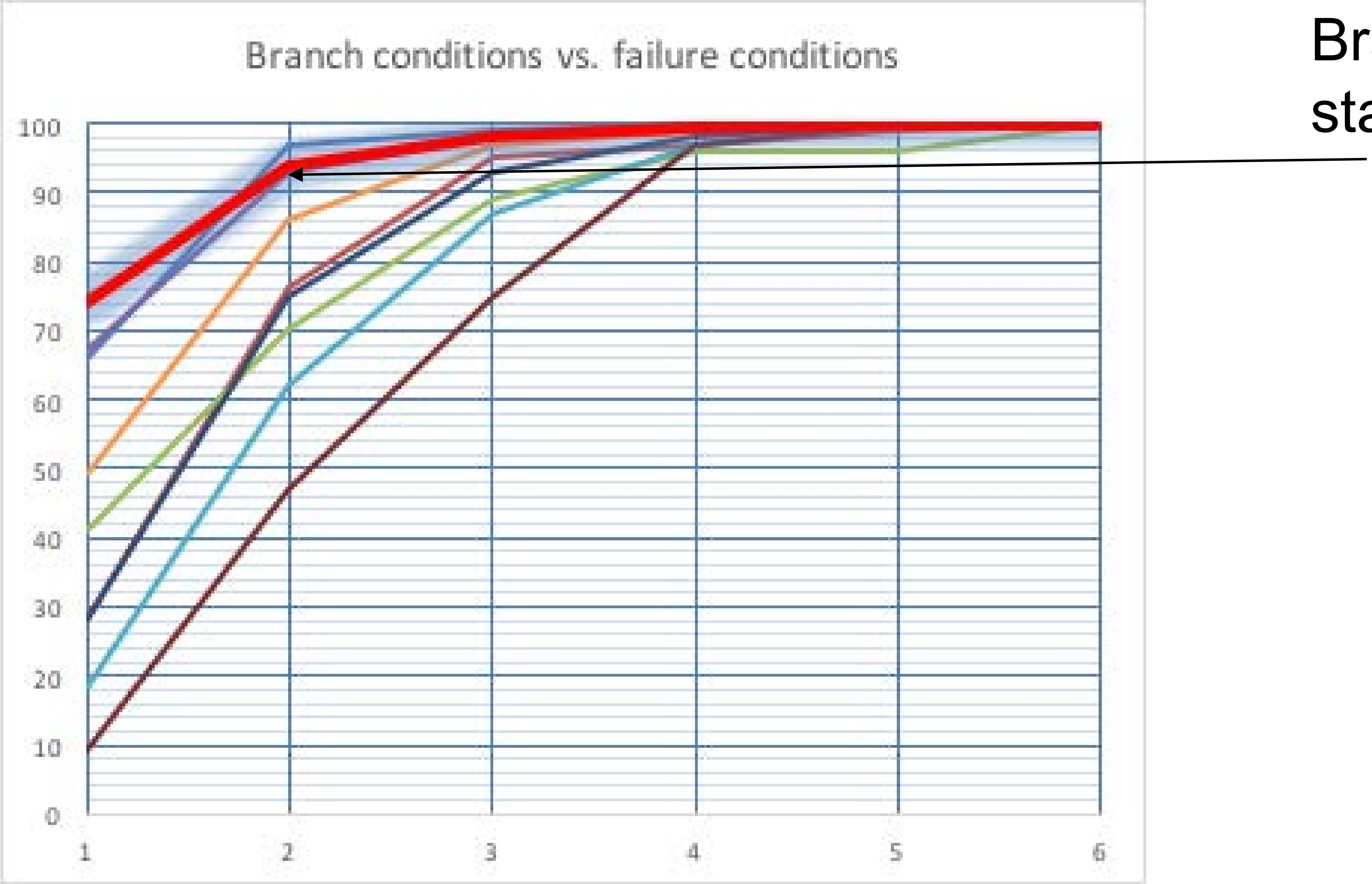


What causes this distribution?



One clue: branches in avionics software.
7,685 expressions from *if* and *while* statements

Comparing with Failure Data



Branch statements

- Distribution of t-way faults in untested software seems to be similar to distribution of t-way branches in code
- Testing and use push curve down as easy (1-way, 2-way) faults found

How does this knowledge help?

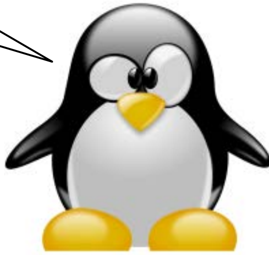
Interaction rule: When all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations is *pseudo-exhaustive* and can provide strong assurance.

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

The interaction rule says we don’t have to

(within reason; we still have value propagation issues, equivalence partitioning, timing issues, more complex interactions, . . .)

Still no silver bullet. Rats!



Overview

1. Intro, empirical data and fault model
2. **How it works and coverage/cost considerations**
3. Practical applications
4. Research topics

Design of Experiments - background

Key features of DoE

- Blocking
- Replication
- Randomization
- Orthogonal arrays to test interactions between factors

Test	P1	P2	P3
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

Each combination occurs same number of times, usually once.

Example: P1, P2 = 1,2

Orthogonal Arrays for Software Interaction 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)

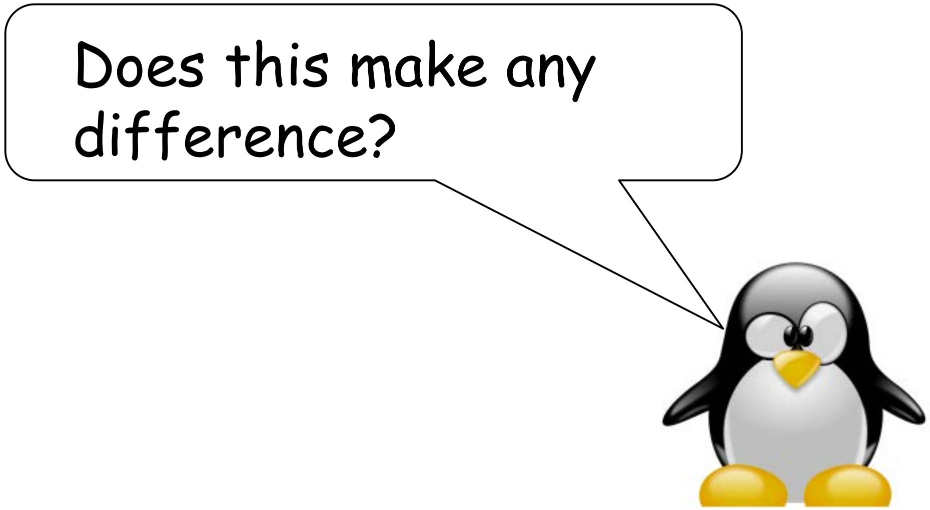
What’s different about software?

Traditional DoE

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

DoE for Software

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



How do these differences affect interaction testing for software?

Not orthogonal arrays, but Covering arrays: Fixed-value CA(N, v^k, t) has four parameters N, k, v, t : It is a matrix covers every t-way combination at least once

Key differences

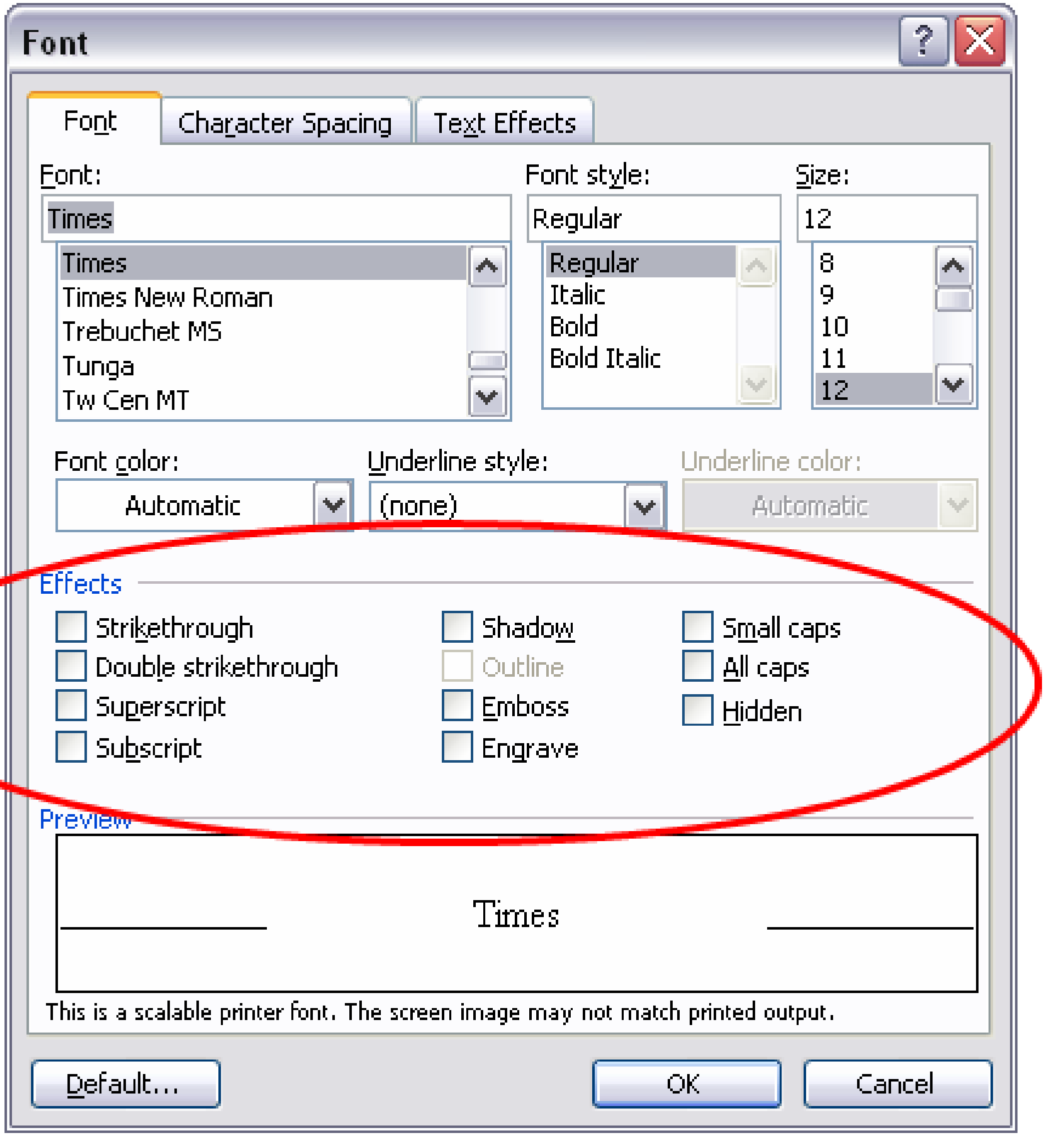
orthogonal arrays:

- Combinations occur same number of times
- Not always possible to find for a particular configuration

covering arrays:

- Combinations occur at least once
- Always possible to find for a particular configuration
- Size always ≤ orthogonal array

Let’s see how to use this in testing.
A simple example:



Font

Font | Character Spacing | Text Effects

Font: Times
Font style: Regular
Size: 12

Font color: Automatic
Underline style: (none)
Underline color: Automatic

Effects

☐ Strikethrough
☐ Double strikethrough
☐ Superscript
☐ Subscript
☐ Shadow
☐ Outline
☐ Emboss
☐ Engrave
☐ Small caps
☐ All caps
☐ Hidden

Preview

Times

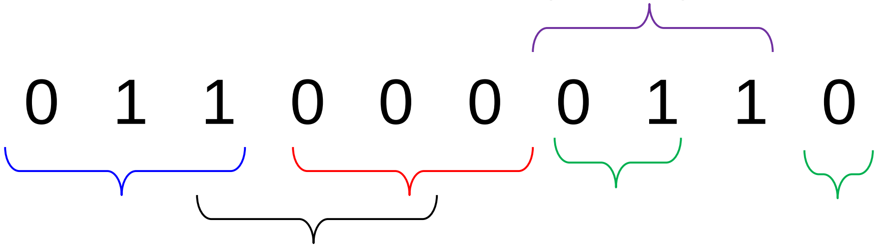
This is a scalable printer font. The screen image may not match printed output.

How Many Tests Would It Take?

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

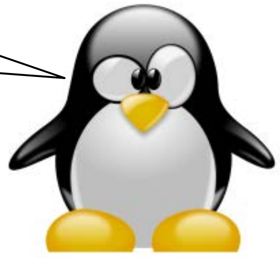
Now How Many Would It Take?

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



0 1 1 0 0 0 0 1 1 0

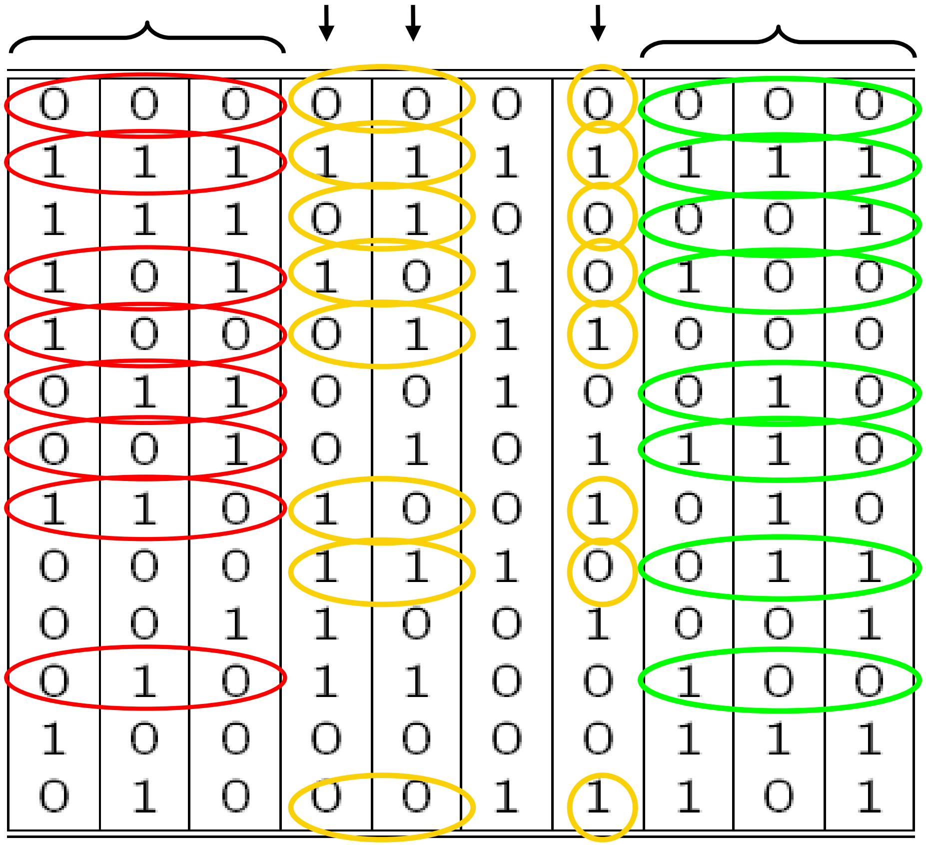
OK, OK, what's the smallest number of tests we need?



A covering array of 13 tests

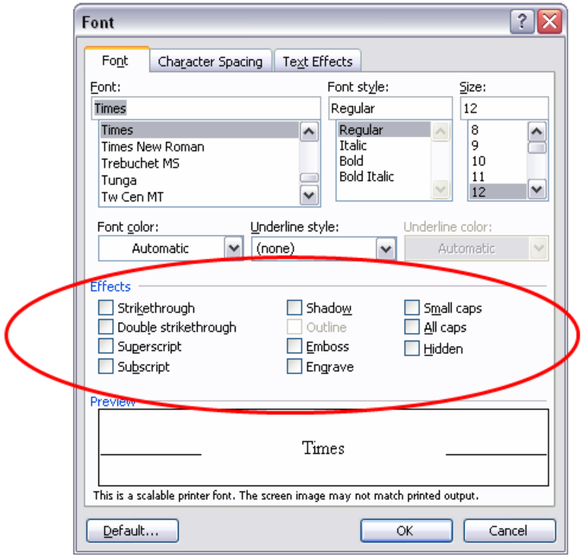
All triples in only 13 tests, covering $\binom{10}{3} 2^3 = 960$ combinations

Each row is a test:



0	0	0	0	0	0	0	0	0	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:



Font: Times

Font style: Regular

Size: 12

Font color: Automatic

Underline style: (none)

Underline color: Automatic

Effects

☐ Strikethrough
☐ Double strikethrough
☐ Superscript
☐ Subscript
☐ Shadow
☐ Outline
☐ Emboss
☐ Engrave
☐ Small caps
☐ All caps
☐ Hidden

Preview

Times

This is a scalable printer font. The screen image may not match printed output.

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

NIST
National Institute of Standards and Technology

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

How many tests are needed?

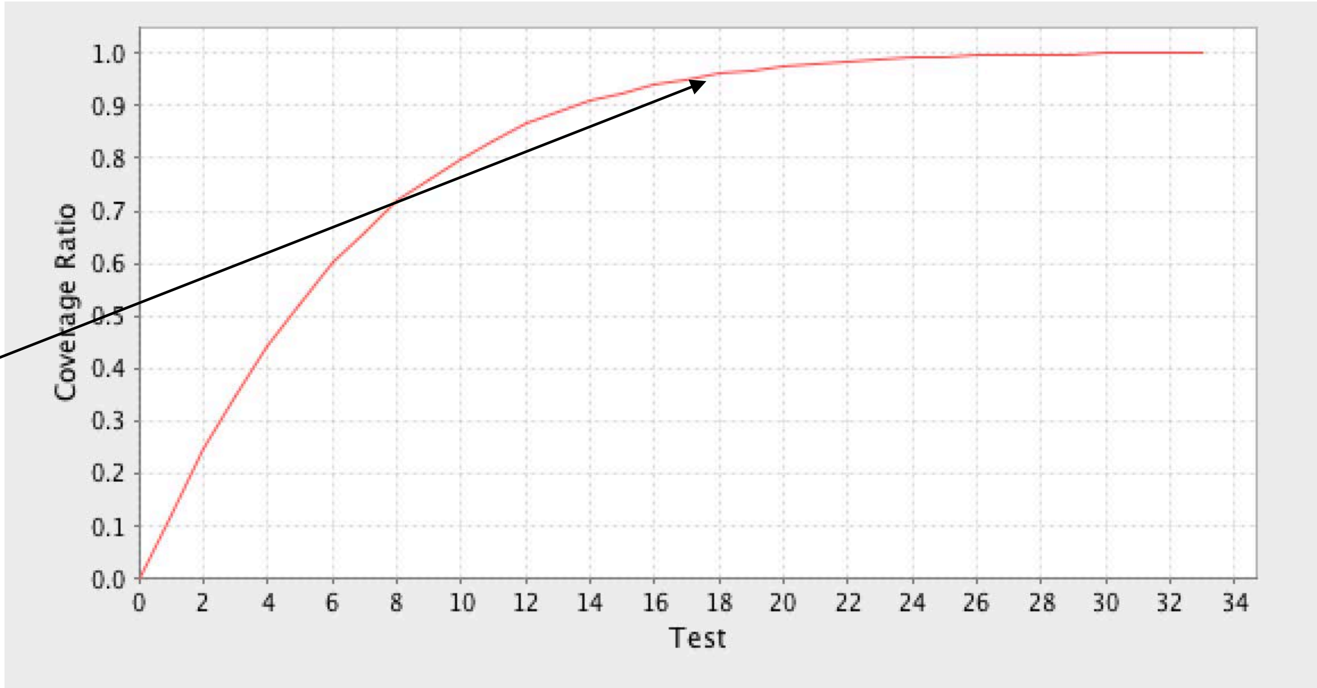
- Number of tests: proportional to $v^t \log n$ for *v* values, *n* variables, *t*-way interactions

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

- Bad news: increase <u>exponentially with interaction strength *t*</u>
=> select small number of representative values (but we always have to do this for any kind of testing)

However:

- coverage increases rapidly
- for 30 boolean variables
- 33 tests to cover all 3-way combinations
- but only 18 tests to cover about 95% of 3-way combinations



Testing inputs – combinations of variable values

Suppose we have a system with on-off switches.

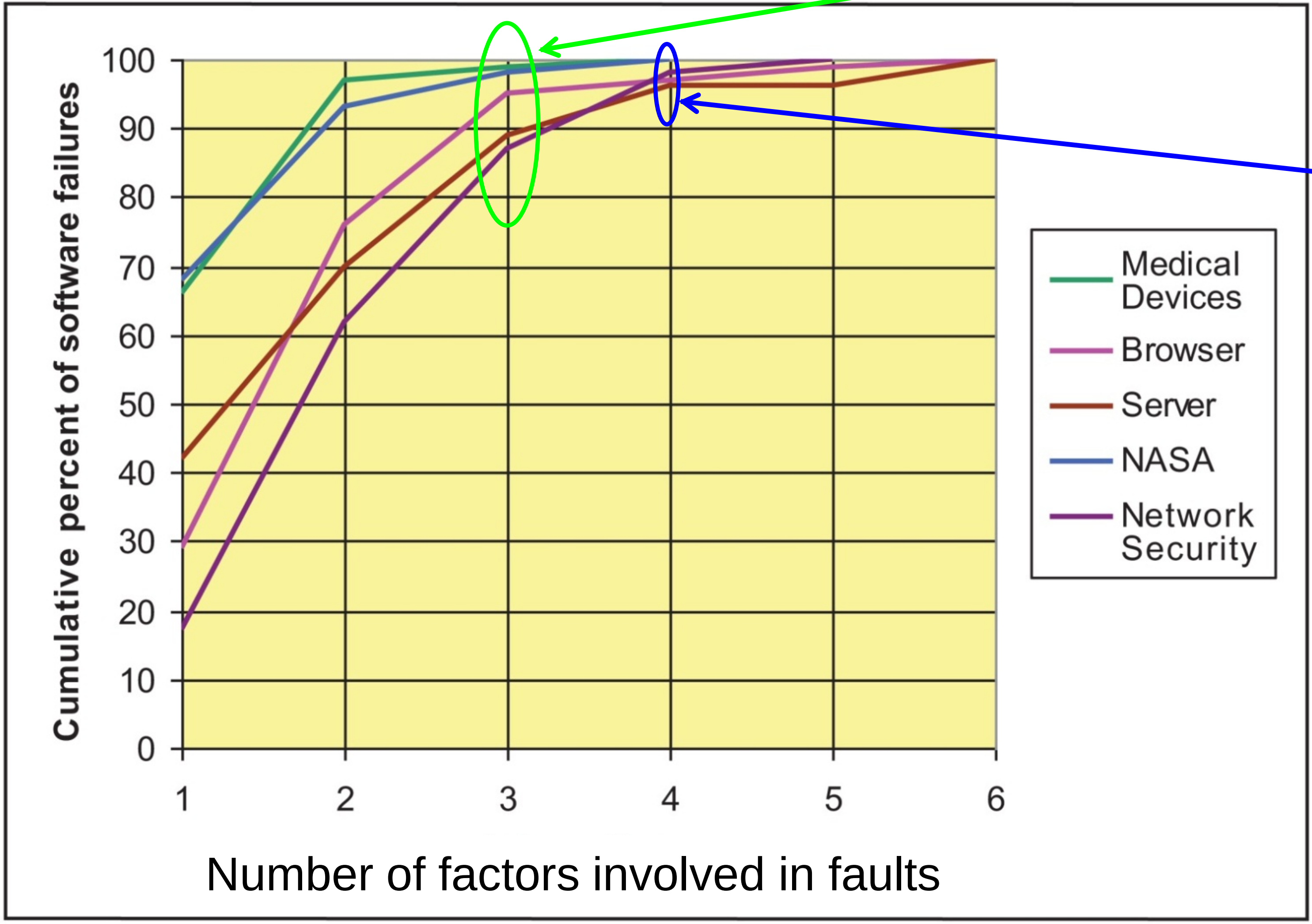
Software must produce the right response for any combination of switch settings

How do we test this?

34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = 17 billion tests

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

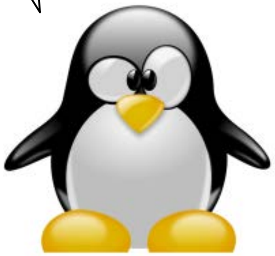
- 34 switches = 17 billion tests
- For 3-way interactions, need only **33** tests
- For 4-way interactions, need only **85** tests



33 tests for this (average) range of fault detection

85 tests for this (average) range of fault detection

That’s way better than 17 billion!



Testing inputs – combinations of property values

Suppose we want to test a **find-replace** function with only two inputs: search_string and replacement_string

How does combinatorial testing make sense in this case?

Problem example from Natl Vulnerability Database:
2-way interaction fault: *single character search string in conjunction with a single character replacement string, which causes an "off by one overflow"*

Approach: test properties of the inputs

Some properties for this test

String length: {0, 1, 1..*file_length*, >*file_length*}

Quotes: {yes, no, improperly formatted quotes}

Blanks: {0, 1, >1}

Embedded quotes: {0, 1, 1 escaped, 1 not escaped}

Filename: {valid, invalid}

Strings in command line: {0, 1, >1}

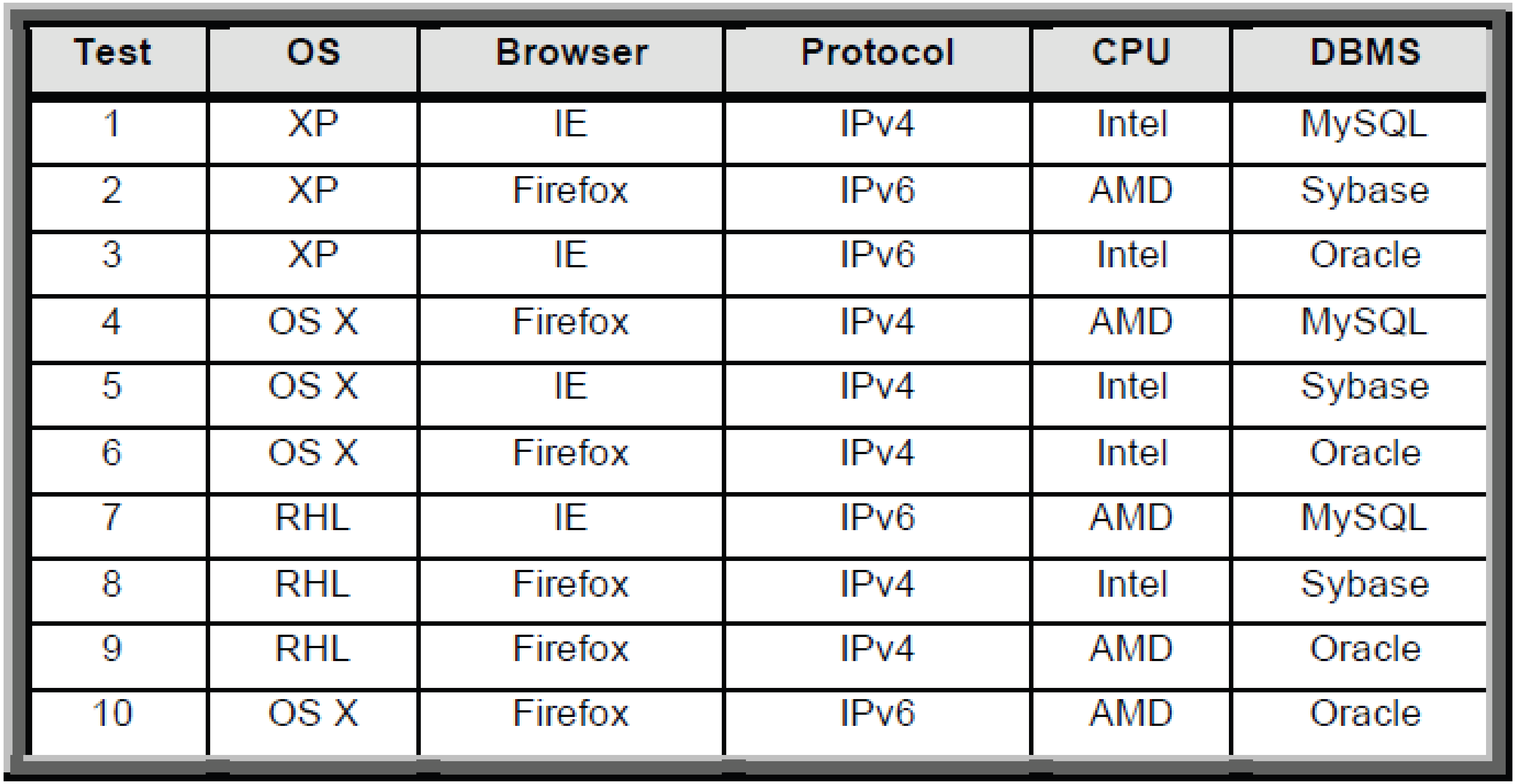
String presence in file: {0, 1, >1}

This is $2^1 3^4 4^2$= 2,592 possible combinations of parameter values. How many tests do we need for pairwise (2-way)?

We need only 19 tests for pairwise, 67 for 3-way, 218 for 4-way

Testing configurations – combinations of settings

- Example: application to 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

Testing Smartphone Configurations

Some 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

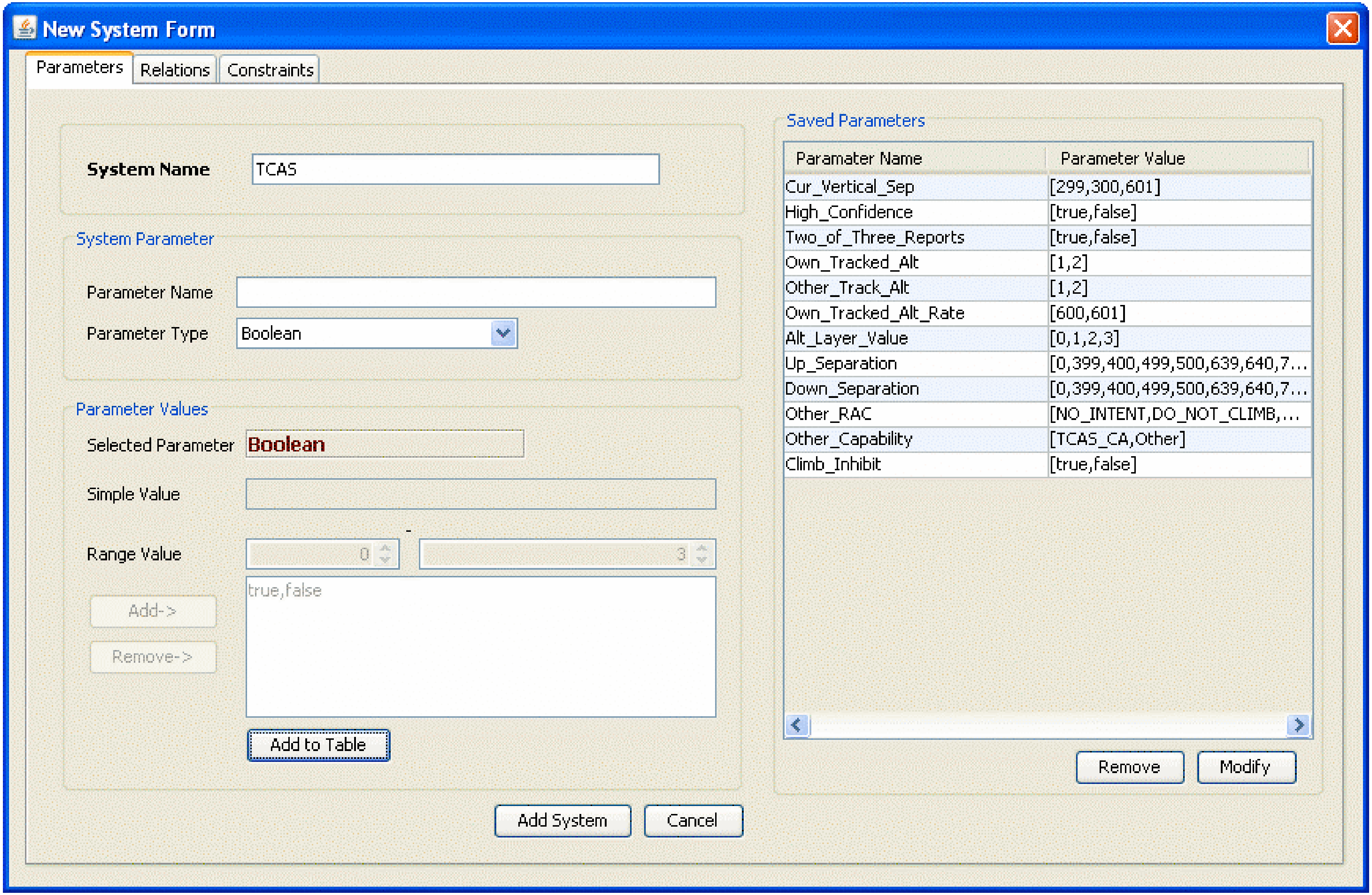
Total possible configurations:

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

Number of configurations generated for *t*-way interaction testing, *t* = 2..6

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

ACTS - Defining a new system



System Name TCAS

Parameter Name

Parameter Type Boolean

Selected Parameter **Boolean**

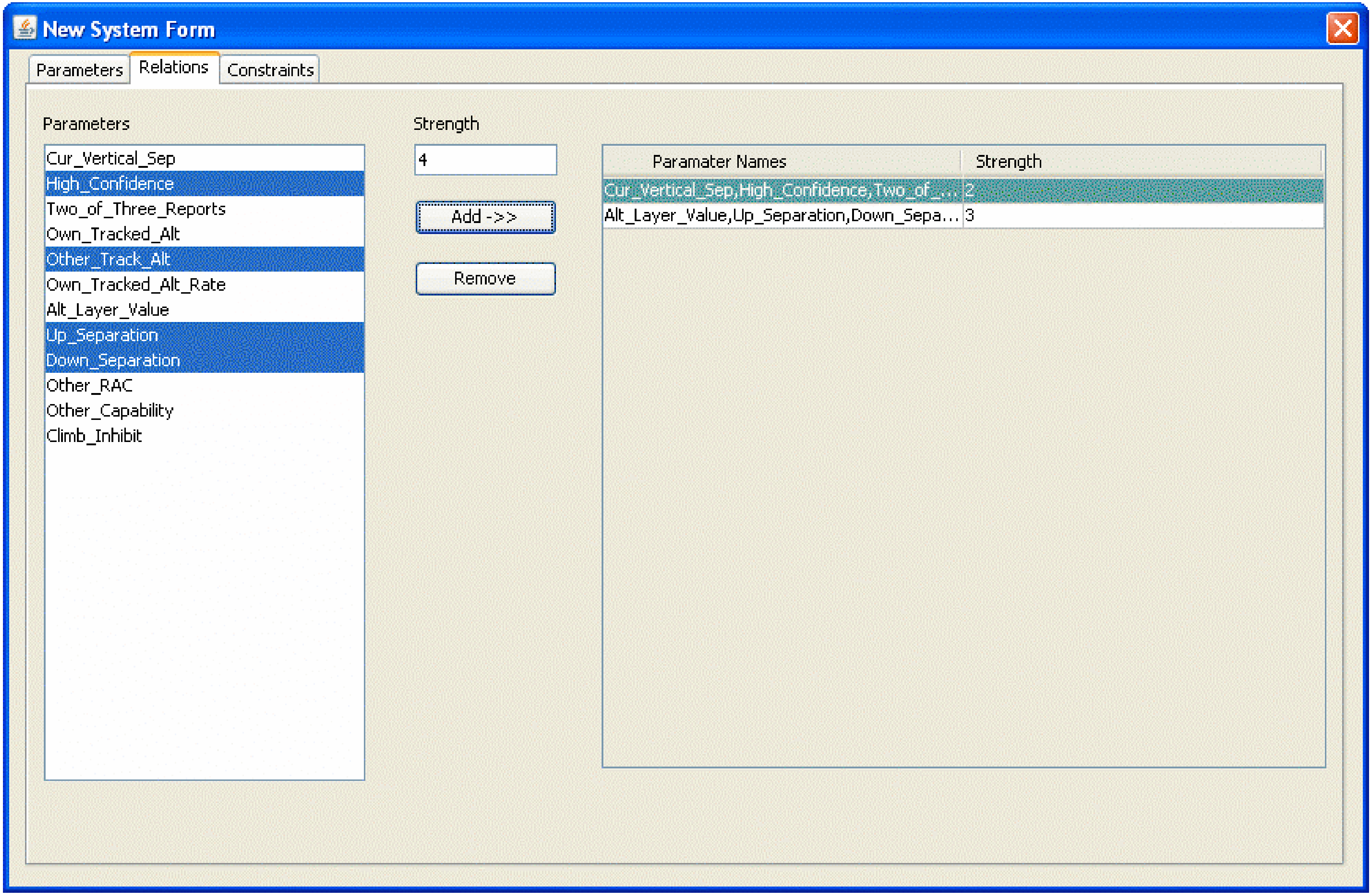
Simple Value

Range Value 0 - 3

true,false

Paramater Name	Parameter Value
Cur_Vertical_Sep	[299,300,601]
High_Confidence	[true,false]
Two_of_Three_Reports	[true,false]
Own_Tracked_Alt	[1,2]
Other_Track_Alt	[1,2]
Own_Tracked_Alt_Rate	[600,601]
Alt_Layer_Value	[0,1,2,3]
Up_Separation	[0,399,400,499,500,639,640,7...
Down_Separation	[0,399,400,499,500,639,640,7...
Other_RAC	[NO_INTENT,DO_NOT_CLIMB,...
Other_Capability	[TCAS_CA,Other]
Climb_Inhibit	[true,false]

Variable interaction strength



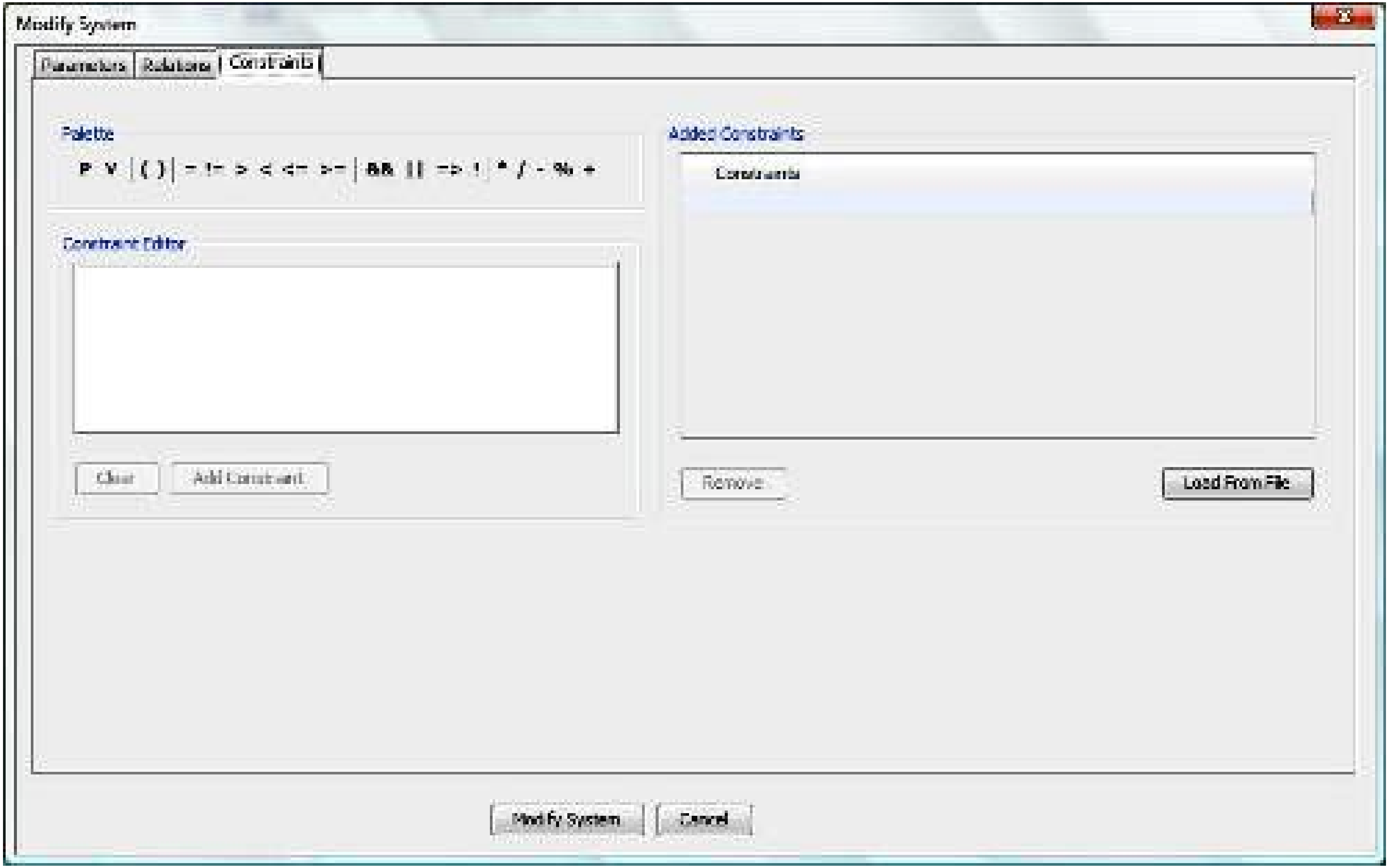
Parameters

- Cur_Vertical_Sep
- High_Confidence
- Two_of_Three_Reports
- Own_Tracked_Alt
- Other_Track_Alt
- Own_Tracked_Alt_Rate
- Alt_Layer_Value
- Up_Separation
- Down_Separation
- Other_RAC
- Other_Capability
- Climb_Inhibit

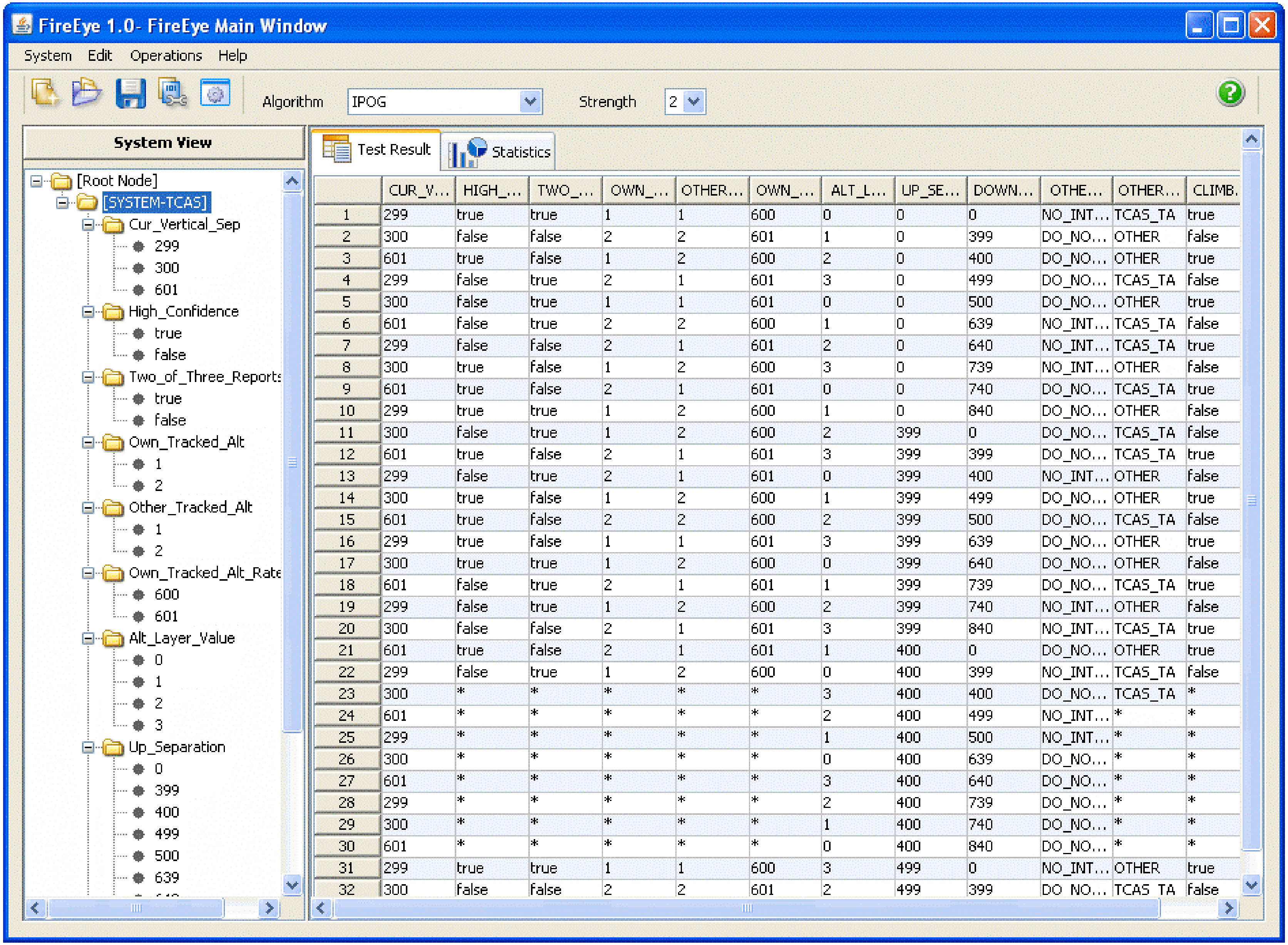
Strength: 4

Paramater Names	Strength
Cur_Vertical_Sep,High_Confidence,Two_of_...	2
Alt_Layer_Value,Up_Separation,Down_Sepa...	3

Constraints



Covering array output



Algorithm: IPOG

Strength: 2

- [Root Node]
  - [SYSTEM-TCAS]
    - Cur_Vertical_Sep
      - 299
      - 300
      - 601
    - High_Confidence
      - true
      - false
    - Two_of_Three_Reports
      - true
      - false
    - Own_Tracked_Alt
      - 1
      - 2
    - Other_Tracked_Alt
      - 1
      - 2
    - Own_Tracked_Alt_Rate
      - 600
      - 601
    - Alt_Layer_Value
      - 0
      - 1
      - 2
      - 3
    - Up_Separation
      - 0
      - 399
      - 400
      - 499
      - 500
      - 639

	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 options

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

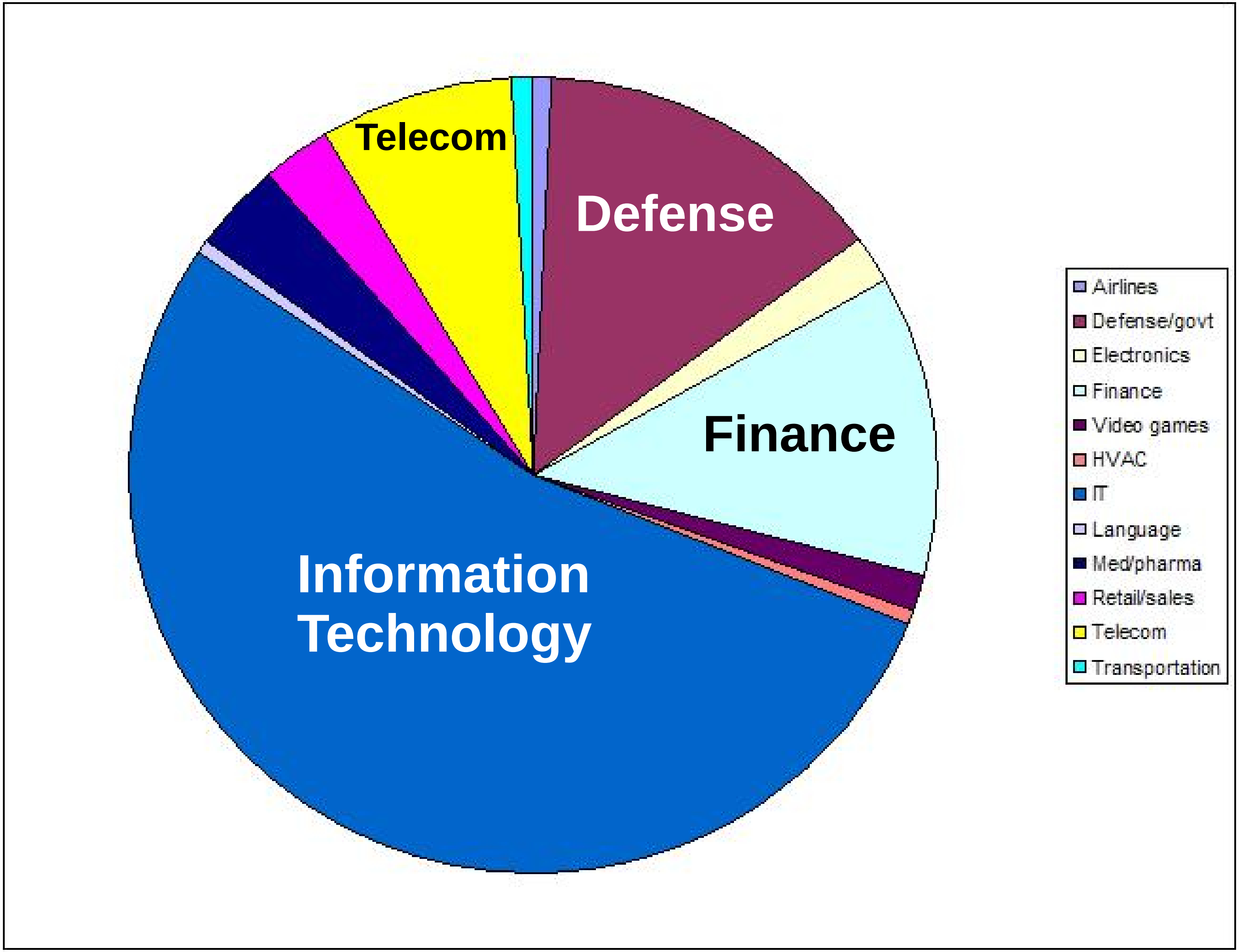
```
Degree of interaction coverage: 2
Number of parameters: 12
Maximum number of values per
parameter: 10
Number of configurations: 100
-----------------------------------
Configuration #1:

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
```

Available Tools

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

ACTS Users > 2,000 organizations



Overview

1. Intro, empirical data and fault model
2. How it works and coverage/cost considerations
3. **Practical applications**
4. Research topics

Case study example: Subway control system

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

Original testing by company: 2 months

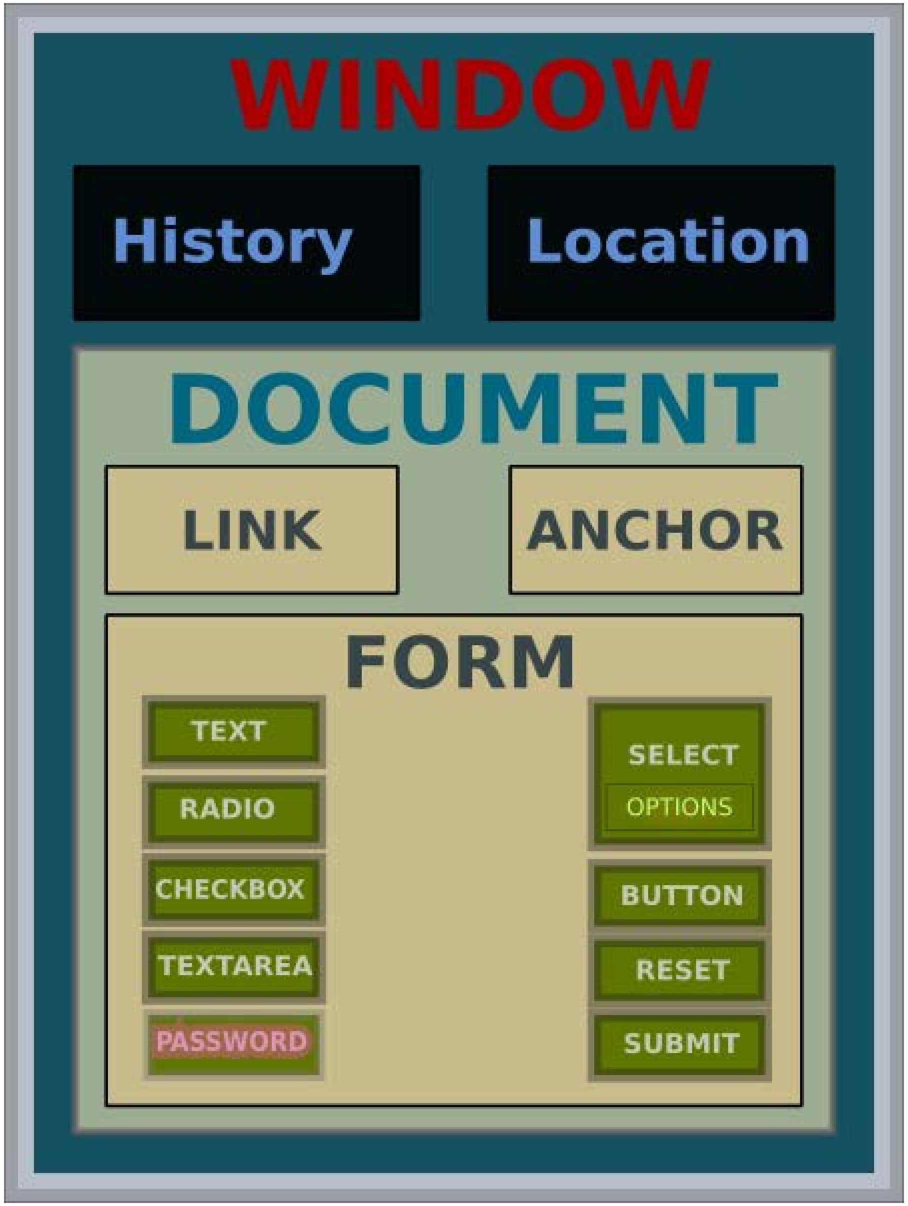
Combinatorial testing by U. Texas students: 2 weeks

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

Results

		Number of test cases	Number of bugs found	Did CT find all original bugs?
Package 1	Original	98	2	-
	CT	49	6	Yes
Package 2	Original	102	1	-
	CT	77	5	Yes
Package 3	Original	116	2	-
	CT	80	7	Miss 1
Package 4	Original	122	2	-
	CT	90	4	Yes

Research question – validate interaction rule?



- DOM is a World Wide Web Consortium standard for representing and interacting with browser objects
- NIST developed conformance tests for DOM
- Tests covered all possible combinations of discretized values, >36,000 tests

- Question: can we use the Interaction Rule to increase test effectiveness the way we claim?

Document Object Model Events

Original test set:

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

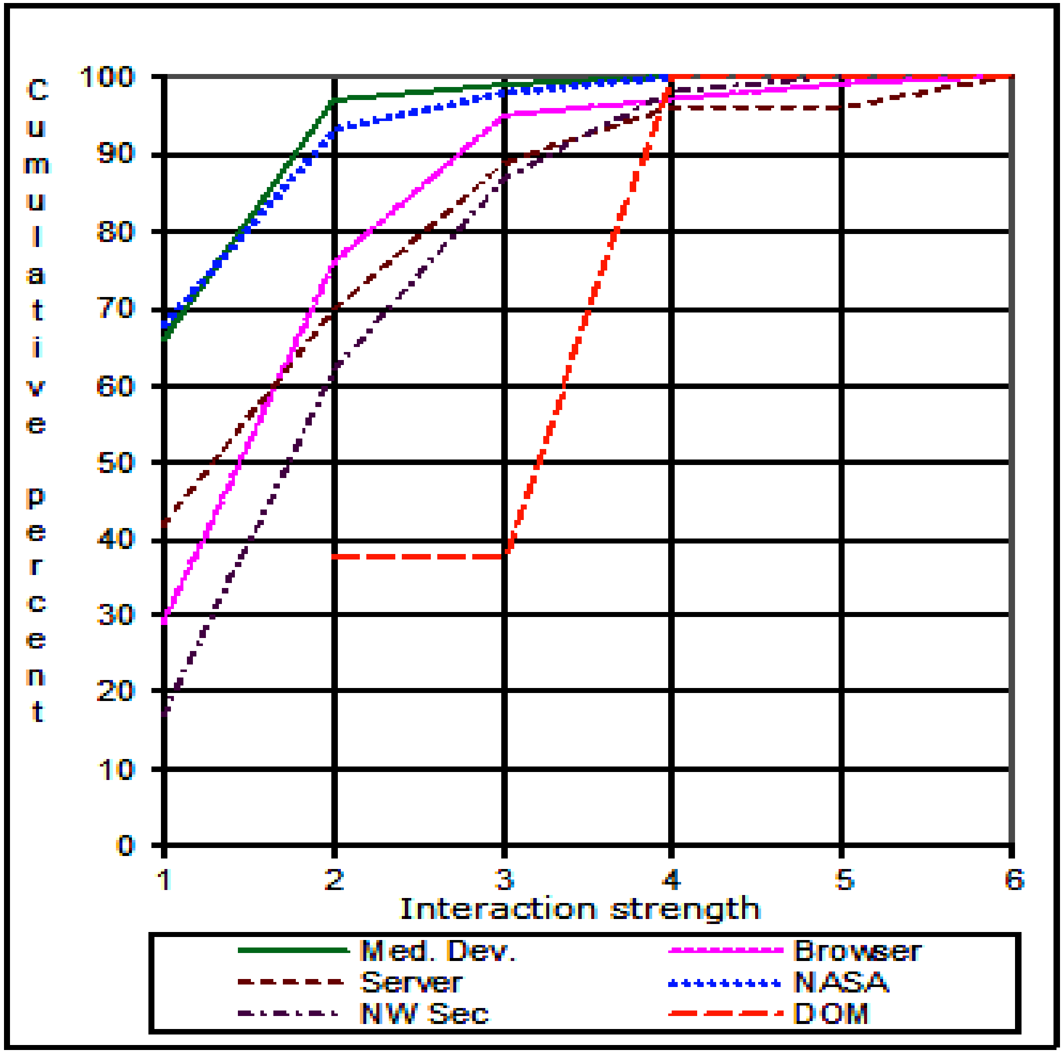
Exhaustive testing of equivalence class values

Document Object Model Events

Combinatorial test set:

t	Tests	% of Orig.	Test Results	
			Pass	Fail
2	702	1.92%	202	27
3	1342	3.67%	786	27
4	1818	4.96%	437	72
5	2742	7.49%	908	72
6	4227	11.54 %	1803	72

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



Modeling & Simulation

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

2. **Network defense/offense operations - NIST – analyze network configuration for vulnerability to deadlock**

Problem: unknown factors causing failures of F-16 ventral fin

Figure 1. LANTIRN pod carriage on the F-16.

LANTIRN = Low Altitude Navigation & Targeting Infrared for Night

It’s not supposed to look like this:

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

Can the problem factors be found efficiently?

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

Combinatorial testing solution: modeling and simulation using ACTS

Parameter	Values
Aircraft	15, 40
Altitude	5k, 10k, 15k, 20k, 30k, 40k, 50k
Maneuver	hi-speed throttle, slow accel/dwell, L/R 5 deg side slip, L/R 360 roll, R/L 5 deg side slip, Med accel/dwell, R-L-R-L banking, Hi-speed to Low, 360 nose roll
Mach (100th)	40, 50, 60, 70, 80, 90, 100, 110, 120

Results

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

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

Example: Network Simulation

- “Simured” network simulator
    - Kernel of ~ 5,000 lines of C++ (not including GUI)
- Objective: detect configurations that can produce deadlock:
    - Prevent connectivity loss when changing network
    - Attacks that could lock up network
- Compare effectiveness of random vs. combinatorial inputs
- Deadlock combinations discovered
- Crashes in >6% of tests w/ valid values (Win32 version only)

Simulation Input Parameters

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?

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

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)

Event Sequence Testing

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

Event	Description
a	connect range finder
b	connect telecom
c	connect satellite link
d	connect GPS
e	connect video
f	connect UAV

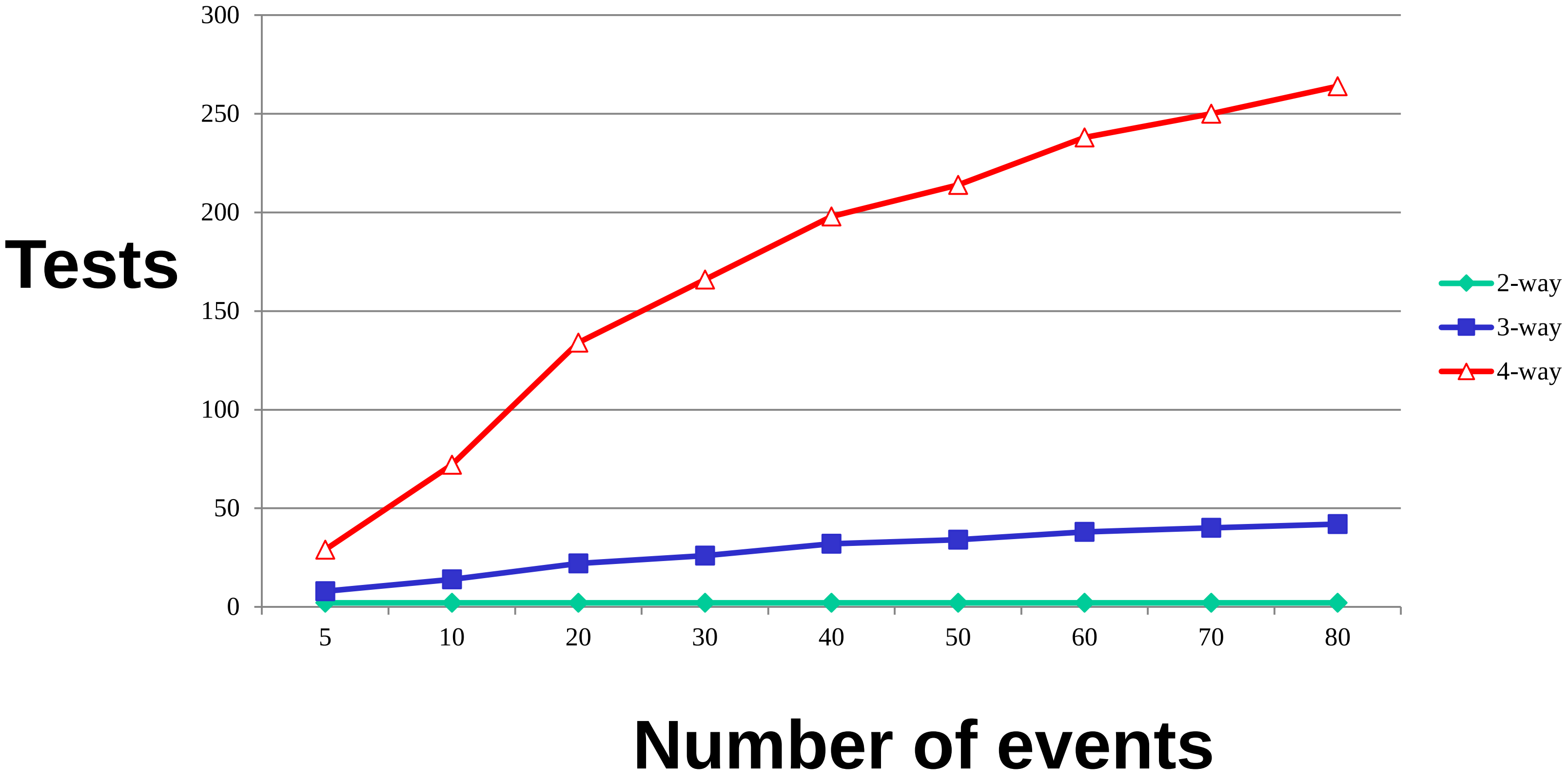
Sequence Covering Array

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



Combinatorial Coverage

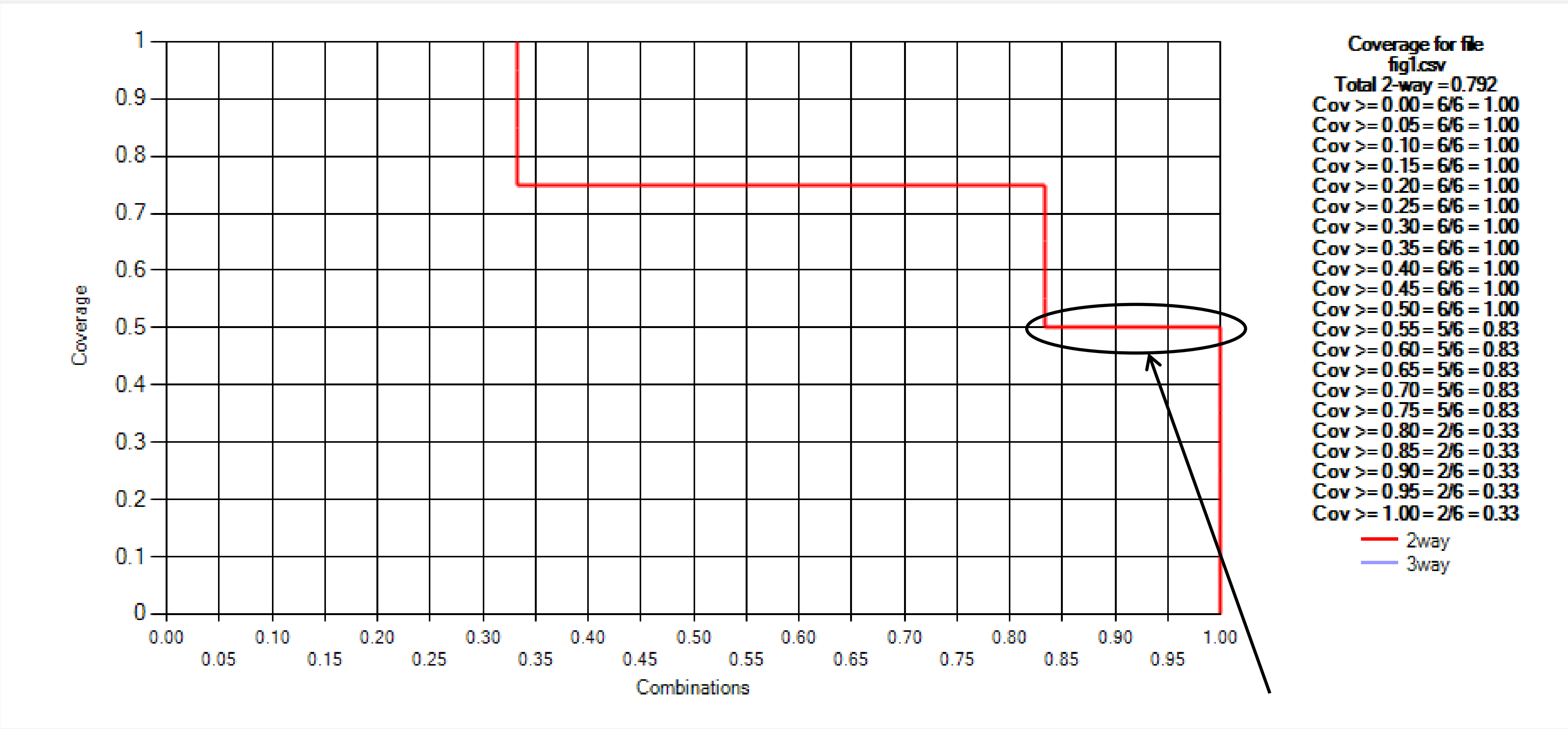
Tests	**Variables**			
	a	**b**	**c**	**d**
1	0	0	0	0
2	0	1	1	0
3	1	0	0	1
4	0	1	1	1

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

National Institute of
Standards and Technology

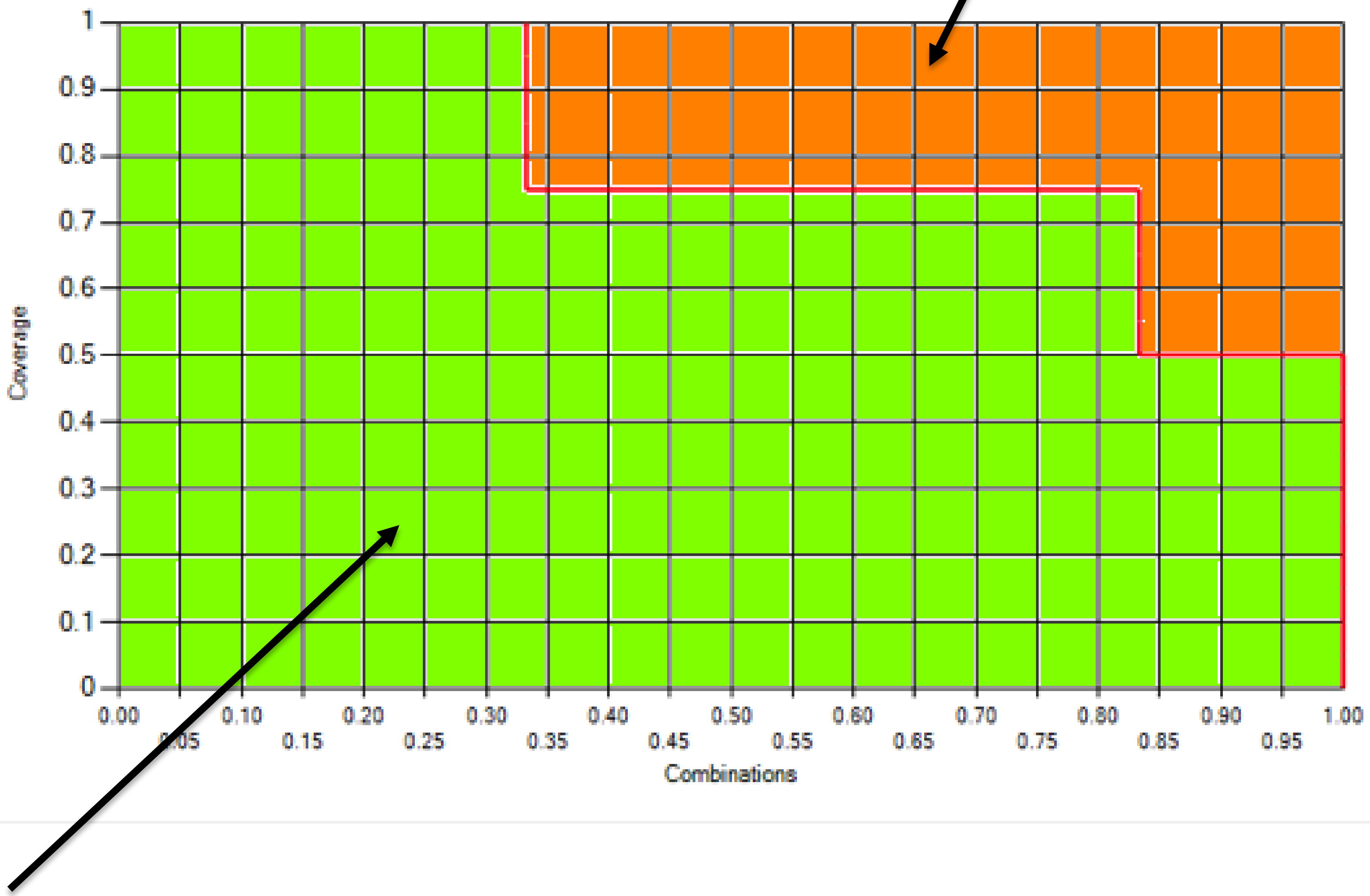
Graphing Coverage Measurement



Coverage for file
fig1.csv
Total 2-way = 0.792
Cov >= 0.00 = 6/6 = 1.00
Cov >= 0.05 = 6/6 = 1.00
Cov >= 0.10 = 6/6 = 1.00
Cov >= 0.15 = 6/6 = 1.00
Cov >= 0.20 = 6/6 = 1.00
Cov >= 0.25 = 6/6 = 1.00
Cov >= 0.30 = 6/6 = 1.00
Cov >= 0.35 = 6/6 = 1.00
Cov >= 0.40 = 6/6 = 1.00
Cov >= 0.45 = 6/6 = 1.00
Cov >= 0.50 = 6/6 = 1.00
Cov >= 0.55 = 5/6 = 0.83
Cov >= 0.60 = 5/6 = 0.83
Cov >= 0.65 = 5/6 = 0.83
Cov >= 0.70 = 5/6 = 0.83
Cov >= 0.75 = 5/6 = 0.83
Cov >= 0.80 = 2/6 = 0.33
Cov >= 0.85 = 2/6 = 0.33
Cov >= 0.90 = 2/6 = 0.33
Cov >= 0.95 = 2/6 = 0.33
Cov >= 1.00 = 2/6 = 0.33

Bottom line:
All combinations covered to at least 50%

What else does this chart show?

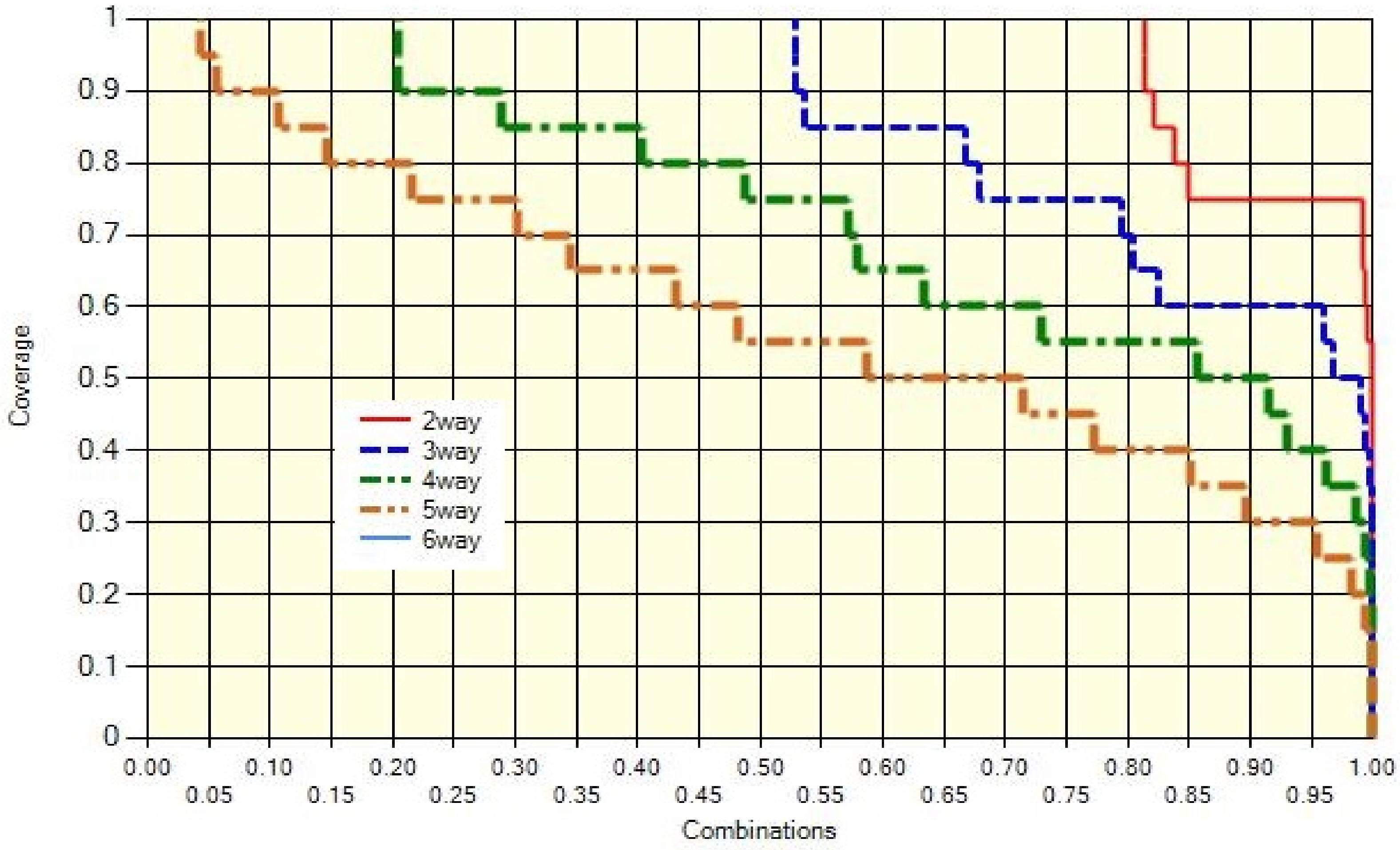


Untested combinations

(look for problems here)

Tested combinations

Spacecraft software example
82 variables, 7,489 tests, conventional test design (not covering arrays)



National Institute of Standards and Technology

Application to testing and assurance

- Useful for providing a measurable value with direct relevance to assurance
- To answer the question:

  How thorough is this test set?

  We can provide a defensible answer

Examples:

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

Overview

1. Intro, empirical data and fault model
2. How it works and coverage/cost considerations
3. Practical applications
4. **Research topics**

How do we automate checking correctness of output?

- **Creating test data is the easy part!**
- How do we check that the code worked correctly on the test input?
  - **Crash testing** server or other code to ensure it does not crash for any test input (like 'fuzz testing')
    - Easy but limited value
  - **Built-in self test with embedded assertions** – incorporate assertions in code to check critical states at different points in the code, or print out important values during execution
  - **Full scale model-checking** using mathematical model of system and model checker to generate expected results for each input - expensive but tractable

Crash Testing

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

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

- buffer overflows
- server crashes

Embedded Assertions

Assertions check properties of expected result:

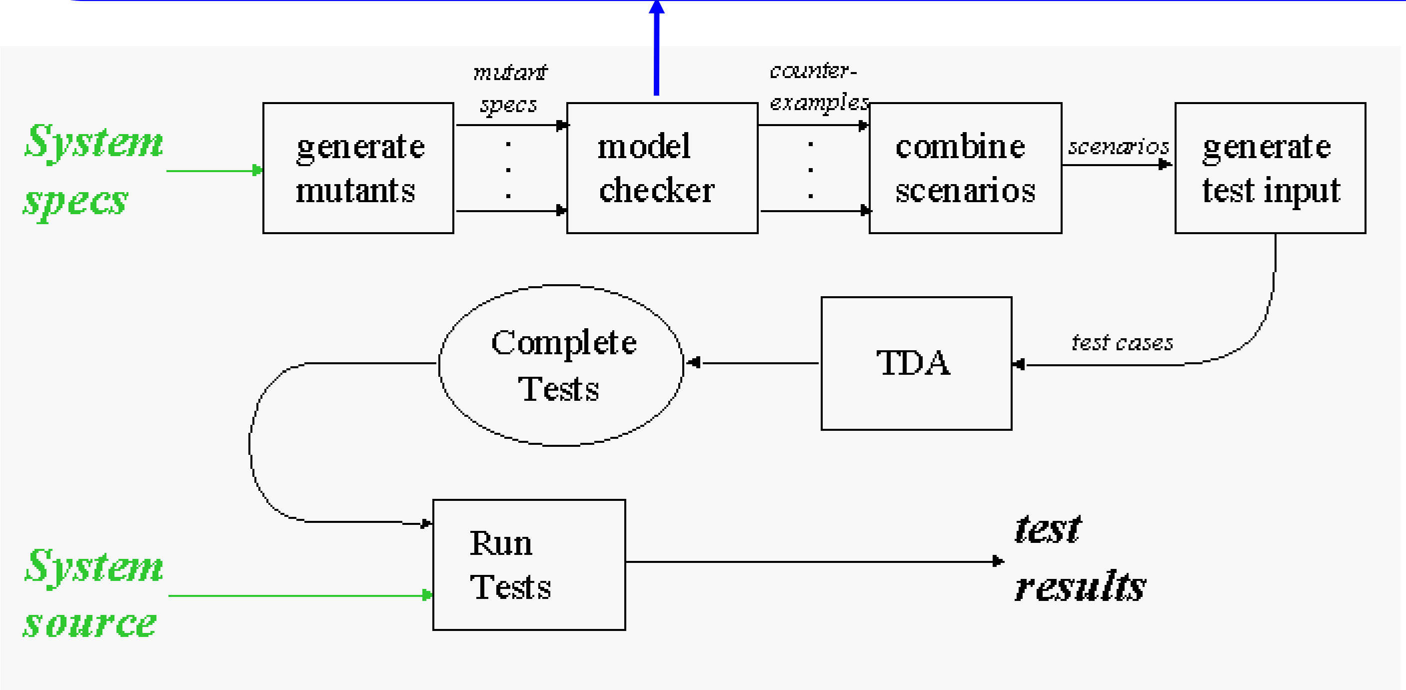
```
ensures balance  == \old(balance) - amount
 &&  \result == balance;
```

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

•May identify problems with handling unanticipated inputs

•Example: Smart card testing
- Used Java Modeling Language (JML) assertions
- Detected 80% to 90% of flaws

Using model checking to produce tests



Black & Ammann, 1999

- Model-checker test production:
  if assertion is not true, then a counterexample is generated.
- This can be converted to a test case.

Testing inputs

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

Model checking example

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

Computation Tree Logic

The usual logic operators,plus temporal:

A φ - All: φ holds on all paths starting from the current state.

E φ - Exists: φ holds on some paths starting from the current state.

G φ - Globally: φ has to hold on the entire subsequent path.

F φ - Finally: φ eventually has to hold

X φ - Next: φ has to hold at the next state

[others not listed]

execution paths

states on the execution paths

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

“FOR ALL executions,
IF enabled & (intent_not_known ....
THEN in the next state alt_sep = UPWARD_RA”

What is the most effective way to integrate combinatorial testing with model checking?

- Given `AG(P -> AX(R))`
  "for all paths, in every state,
  if P then in the next state, R holds"
- For k-way variable combinations, `v1 & v2 & ... & vk`
- vi abbreviates "var1 = val1"
- Now combine this constraint with assertion to produce counterexamples. Some possibilities:

1. `AG(v1 & v2 & ... & vk & P -> AX !(R))`
2. `AG(v1 & v2 & ... & vk -> AX !(1))`
3. `AG(v1 & v2 & ... & vk -> AX !(R))`

What happens with these assertions?

1. `AG(v1 & v2 & ... & vk & P -> AX !(R))`

   P may have a negation of one of the v_i, so we get

   `0 -> AX !(R))`

   always true, so no counterexample, no test.

   This is too restrictive!

2. `AG(v1 & v2 & ... & vk -> AX !(1))`

   The model checker makes non-deterministic choices for variables not in v1..vk, so all R values may not be covered by a counterexample.

   This is too loose!

3. `AG(v1 & v2 & ... & vk -> AX !(R))`

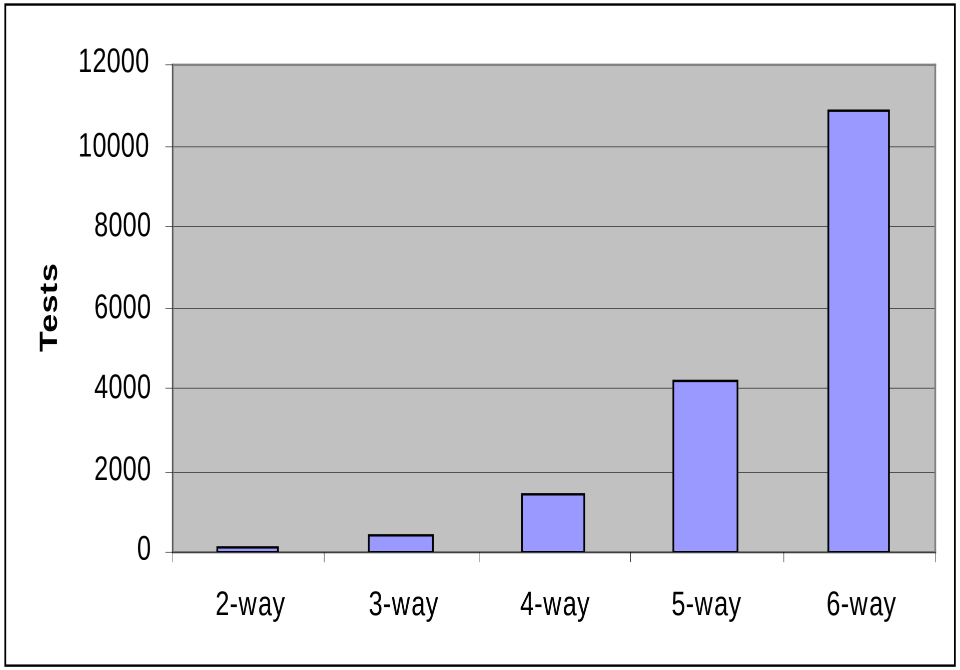
   Forces production of a counterexample for each R.

   This is just right!

NIST
National Institute of
Standards and Technology

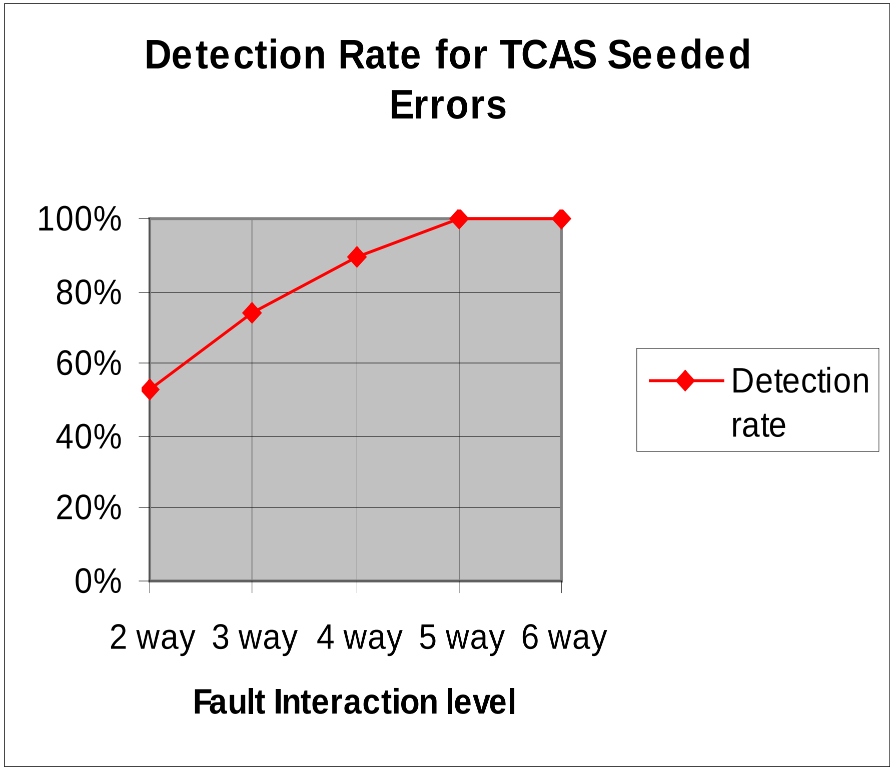
Tests generated

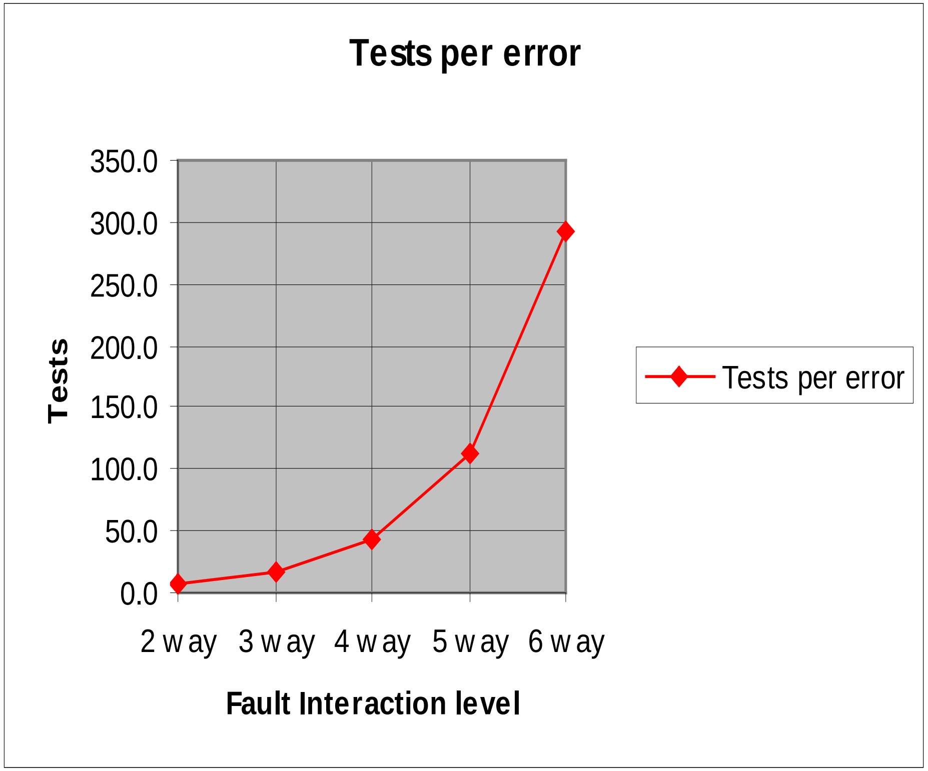
t	Test cases
2-way:	156
3-way:	461
4-way:	1,450
5-way:	4,309
6-way:	11,094



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

Oracle-free testing

Some current approaches:

Fuzz testing – send random values until system fails, then analyze memory dump, execution traces

Metamorphic testing – e.g. *cos*(*x*) = *cos*(*x*+360), so compare outputs for both, with a difference indicating an error.

Partial test oracle – e.g., insert element x in data structure *S*, check $x \in S$

New method using two-layer covering arrays

Consider equivalence classes

Example: shipping cost based on distance d and weight w, with packages < 1 pound are in one class, 1..10 pounds in another, > 10 in a third class.

Then for cost function $f(d,w)$,

$$f(d, 0.2) = f(d, 0.9),$$

for equal values of d.

But

$$f(d, 0.2) \neq f(d, 5.0),$$

because two different weight classes are involved.

Using the basic property of equivalence classes

when a_1 and a_2 are in the same equivalence class,

$$f(a_1,b,c,d,\ldots) \approx f(a_2,b,c,d,\ldots),$$

where $\approx$ is equivalence with respect to some predicate.

If not, then
- either the code is wrong,
- or equivalence classes are not defined correctly.

Can we use this property for testing?

Let’s do an example: access control. access is allowed if
(1) subject is employee & time is in working hours on a weekday; or
(2) subject is an employee with administrative privileges; or
(3) subject is an auditor and it is a weekday.

Equivalence classes for time of day and day of the week

time = minutes past midnight (0..0539), (0540..1020), (1021..1439).

Days of the week = weekend and weekdays,
designated as (1,7) and (2..6) respectively.

Code we want to test

```
int access_chk() {
   if (emp && t >= START && t <= END &&
           d >= MON && d <= FRI) return 1;
   else
   if (emp && p) return 2;
   else
   if (aud && d >= MON && d <= FRI)
          return 3;
   else
   return 0;
}
```

Establish equivalence classes

emp: boolean

day: (1,7), (2,6)
A1 A2

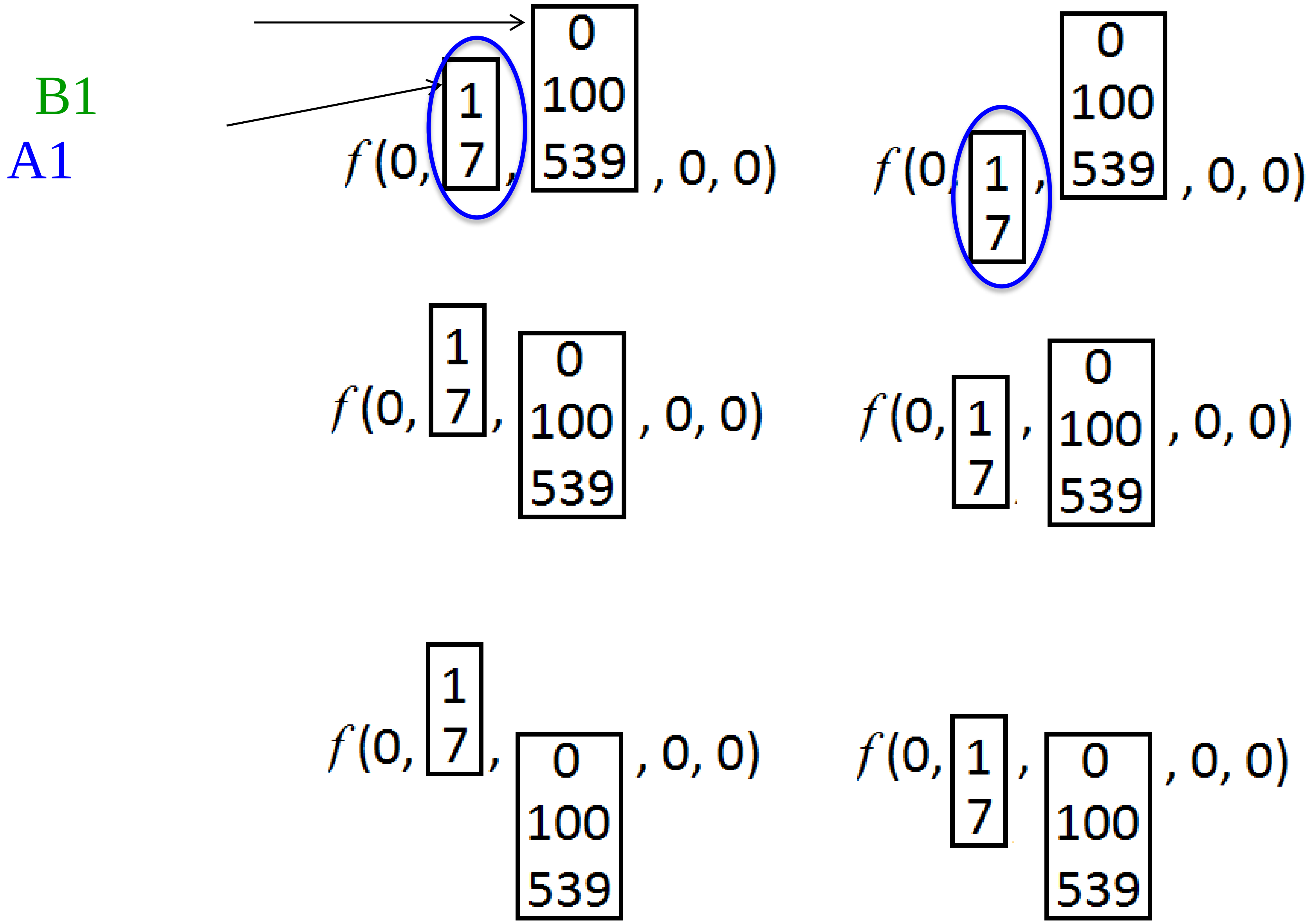
time:(0,100,539),(540,1020),(1021,1439)
B1 B2 B3

priv: boolean

aud: boolean

day (enum) : A1,A2
time (enum): B1,B2,B3

All of these should be equal



B1

A1

$f(0, \boxed{\begin{array}{c}1\\7\end{array}}, \boxed{\begin{array}{c}0\\100\\539\end{array}}, 0, 0)$

$f(0, \boxed{\begin{array}{c}1\\7\end{array}}, \boxed{\begin{array}{c}0\\100\\539\end{array}}, 0, 0)$

$f(0, \boxed{\begin{array}{c}1\\7\end{array}}, \boxed{\begin{array}{c}0\\100\\539\end{array}}, 0, 0)$

$f(0, \boxed{\begin{array}{c}1\\7\end{array}}, \boxed{\begin{array}{c}0\\100\\539\end{array}}, 0, 0)$

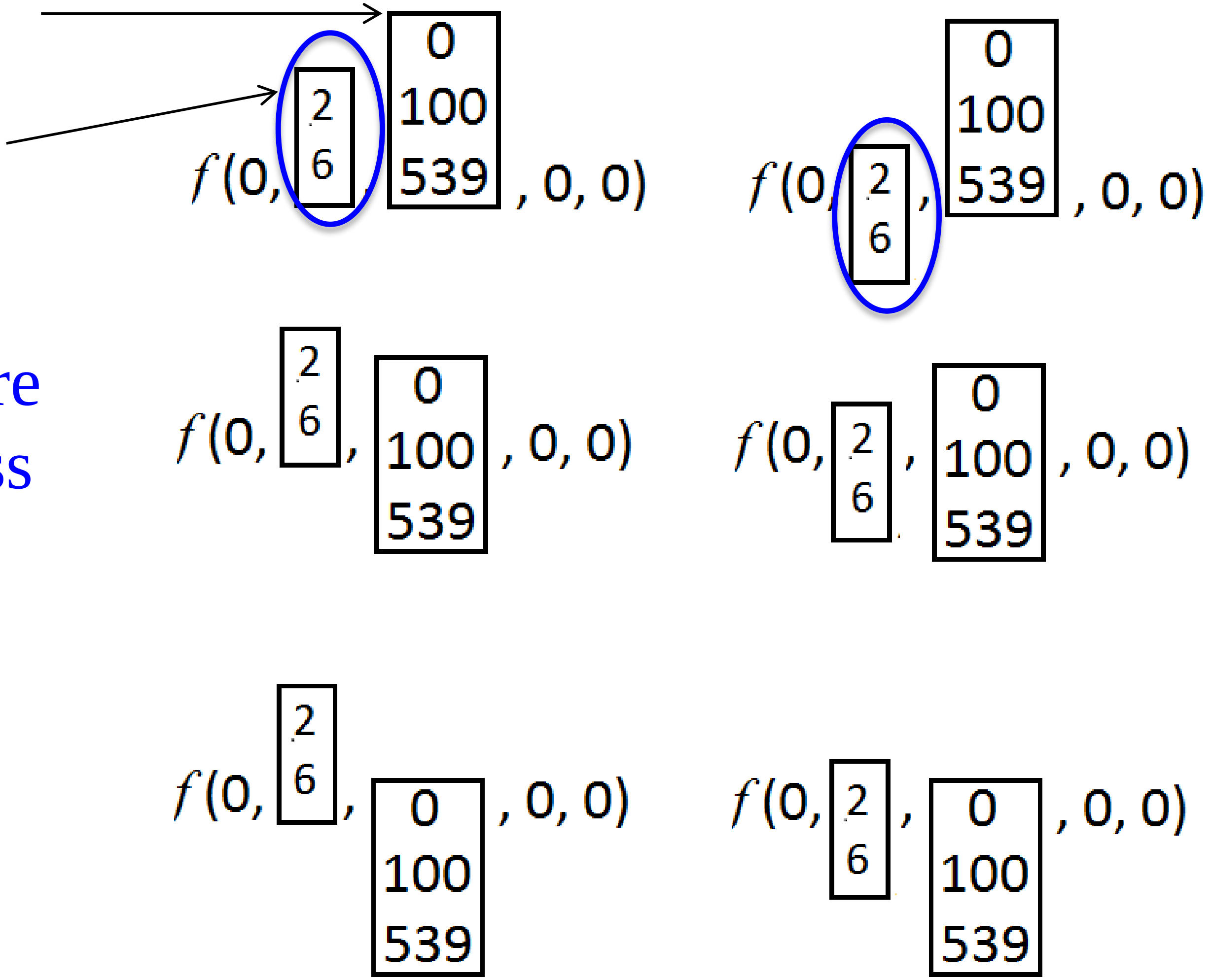
$f(0, \boxed{\begin{array}{c}1\\7\end{array}}, \boxed{\begin{array}{c}0\\100\\539\end{array}}, 0, 0)$

$f(0, \boxed{\begin{array}{c}1\\7\end{array}}, \boxed{\begin{array}{c}0\\100\\539\end{array}}, 0, 0)$

These should also be equal

B1

A2



$f\left(0, \boxed{\begin{matrix}2\\6\end{matrix}}, \boxed{\begin{matrix}0\\100\\539\end{matrix}}, 0, 0\right)$ $f\left(0, \boxed{\begin{matrix}2\\6\end{matrix}}, \boxed{\begin{matrix}0\\100\\539\end{matrix}}, 0, 0\right)$

Now we're using class A2

$f\left(0, \boxed{\begin{matrix}2\\6\end{matrix}}, \boxed{\begin{matrix}0\\100\\539\end{matrix}}, 0, 0\right)$ $f\left(0, \boxed{\begin{matrix}2\\6\end{matrix}}, \boxed{\begin{matrix}0\\100\\539\end{matrix}}, 0, 0\right)$

$f\left(0, \boxed{\begin{matrix}2\\6\end{matrix}}, \boxed{\begin{matrix}0\\100\\539\end{matrix}}, 0, 0\right)$ $f\left(0, \boxed{\begin{matrix}2\\6\end{matrix}}, \boxed{\begin{matrix}0\\100\\539\end{matrix}}, 0, 0\right)$

Covering array

Primary array:

0,A2,B1,1,1
1,A1,B1,0,0
0,A1,B2,1,0
1,A2,B2,0,1
0,A1,B3,0,1
1,A2,B3,1,0

emp: boolean
day: (1,7), (2,6)
A1 A2
time: (0,539),(540,1020),(1021, 1439)
B1 B2 B3
priv: boolean
aud: boolean

Class A2 = (2,6)
Class B1 = (0,539)

0 2 0 1 1
0 6 0 1 1
0 2 539 1 1
0 6 539 1 1

Run the tests

Correct code output:

```
3333
0000
0000
1111
0000
2222
```

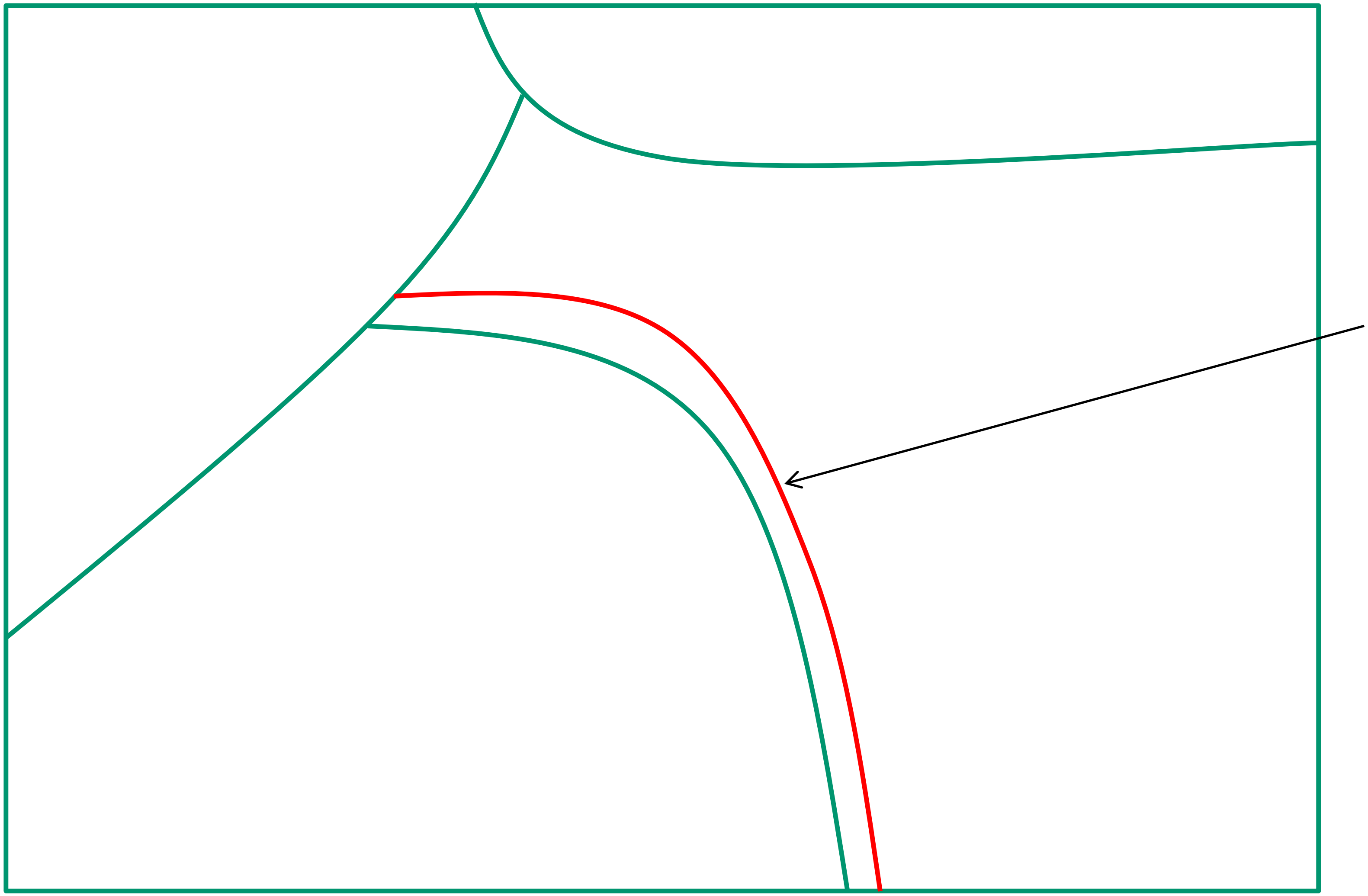
Faulty code:

```
if (emp && t>=START && t==END && d>=MON && d<=FRI) return 1;
```

Faulty code output:

```
3333
0000
0000
3311
0000
2222
```

What’s happening here?



Can this really work on practical code?

Experiment: TCAS code (same used in earlier model checking tests)

- Small C module, 12 variables
- Seeded faults in 41 variants

- Results:

Primary x secondary	#tests	total	faults detected
3-way x 3-way	285x8	2280	6
4-way x 3-way	970x8	7760	22

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

Next Steps

Realistic trial use

Different constructions for secondary array, e.g., random values

Formal analysis of applicability – range of applicability/effectiveness, limitations, special cases

Determine how many faults can be detected this way

Develop tools to incorporate into build process

Another approach to oracle problem

conventional solution:

- “use cases” verifying important or common situations
- ad hoc
- often not very thorough

model-based testing solution:

- rules → formal model → model checker/sim → test cases
- may be based on fault model; mutation testing

Pseudo-exhaustive testing solution using covering arrays:

- determine dependencies
- partition according to these dependencies
- exhaustively test the inputs on which an output is dependent
- example: for access control:
  - convert rule antecedents to *k*-DNF form, producing sets of *k* or fewer attributes that will produce a “grant” decision
  - generate separate *k*-way covering arrays for combinations that should produce “grant” and “deny”

Example: where covering arrays come in

attributes: *employee* , *age*, *first_aid_training*, *EMT_cert*, *med_degree*

rule: “If subject is an employee AND 18 or older AND: (has first aid training OR an EMT certification OR a medical degree), then authorize”

policy:

emp && age > 18 && (fa || emt || med) → grant
else → deny

(emp && age > 18 && fa) ||
(emp && age > 18 && emt) ||
(emp && age > 18 && med)

3-DNF so a 3-way covering array will include combinations that instantiate all of these terms to true

Rule structure

attributes: *employment_status* and *time_of_day*

rule: “If subject is an employee and the hour is between 9 am and 5 pm, then allow entry.”

policy structure:

$R_1 \rightarrow grant$

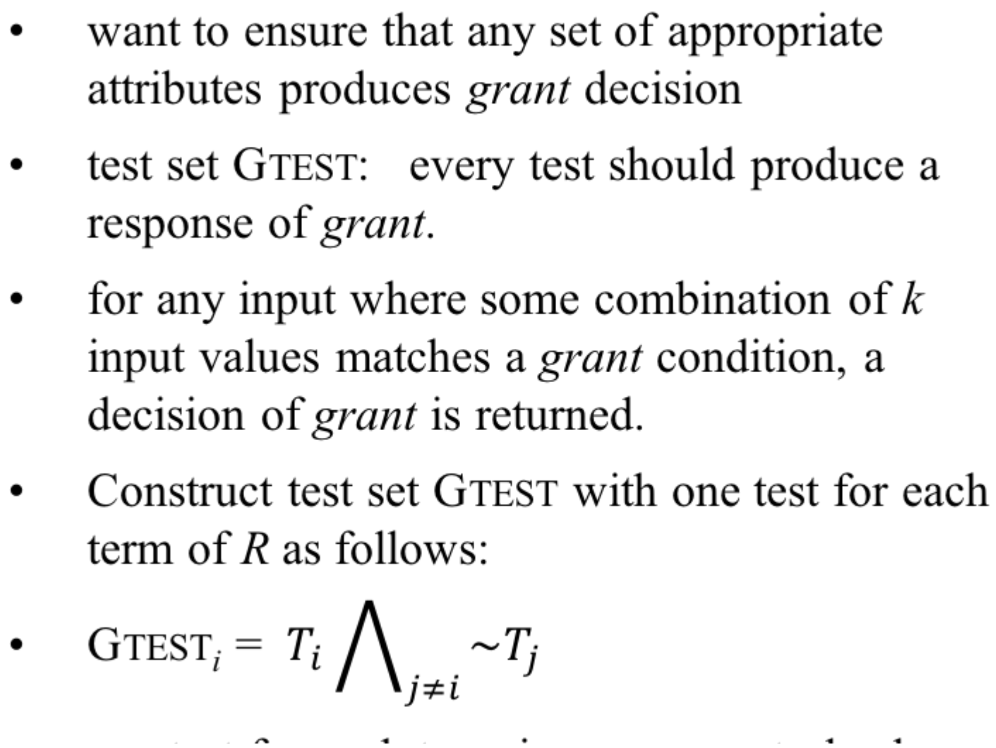
$R_2 \rightarrow grant$

…

$R_m \rightarrow grant$

$else \rightarrow deny$

Positive testing (easy)



- want to ensure that any set of appropriate attributes produces *grant* decision
- test set GTEST: every test should produce a response of *grant*.
- for any input where some combination of k input values matches a *grant* condition, a decision of *grant* is returned.
- Construct test set GTEST with one test for each term of R as follows:
- $\text{GTEST}_i = T_i \bigwedge_{j \neq i} {\sim} T_j$

Negative testing (hard)

- test set DTEST = covering array of strength k, for the set of attributes included in R
- constraints specified by ~R
- ensures that all deny-producing conjunctions of attributes tested
- masking is not a consideration – because of problem structure
  - *deny* is issued only after all *grant* conditions have been evaluated
  - masking of one combination by another can only occur for DTEST when a test produces a response of *grant*
  - if so, an error has been discovered; repair and run test set again

Generating test array for all 3-way negative cases

!*((emp && age > 18 && fa) ||*
(emp && age > 18 && emt) ||
(emp && age > 18 && med))

constraint

Covering array generator

output

All 3-way combinations of these variables except for positive cases

emp	age	fa	emt	med
TRUE	TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	FALSE	TRUE	FALSE	FALSE

Number of tests

for positive tests, Gtest: one test for each term in the rule set, for for *m* rules with *p* terms each , *mp*

for negative tests, Dtest: one covering array per rule, where each attribute in the rule is a factor

easily practical for huge numbers of tests when evaluation is fast - access control systems have to be

k	v	n	m	N tests	#GTEST	#DTEST
3	2	50	20	36	80	720
			50		200	1800
		100	20	45	80	900
			50		200	2250
	4	50	20	306	80	6120
			50		200	15300
		100	20	378	80	7560
			50		200	18900
	6	50	20	1041	80	20820
			50		200	52050
		100	20	1298	80	25960
			50		200	64900
4	2	50	20	98	80	1960
			50		200	4900
		100	20	125	80	2500
			50		200	6250
	4	50	20	1821	80	36420
			50		200	91050
		100	20	2337	80	46740
			50		200	116850
	6	50	20	9393	80	187860
			50		200	469650
		100	20	12085	80	241700
			50		200	604250

Fault detection properties

tests from GTEST and DTEST will detect added, deleted, or altered faults with up to *k* attributes

if more than *k* attributes are included in faulty term *F*, some faults are still detected, for number of attributes $j > k$

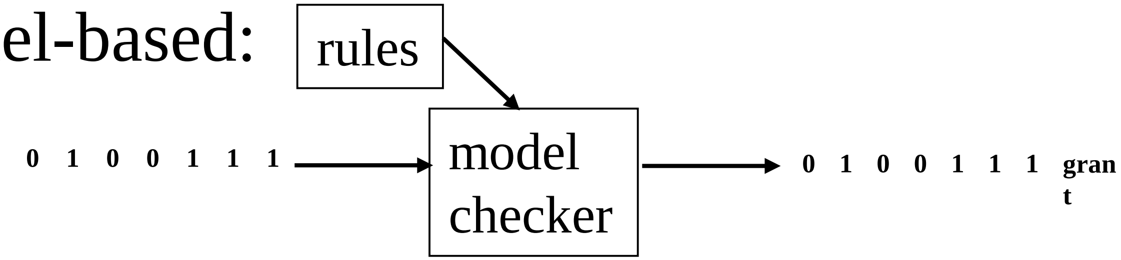
$j > k$ and correct term *C* is not a subset of *F*: detected by GTEST

$j > k$ and *C* is a subset of *F*: not detected by DTEST; possibly detected by GTEST; higher strength covering arrays for DTEST can detect

generalized to cases with more than grant/deny outputs; suitable for small number of outputs which can be distinguished
(in principle can be applied with large number of outputs)

Summarizing: Comparison with Model-based Testing

model-based:



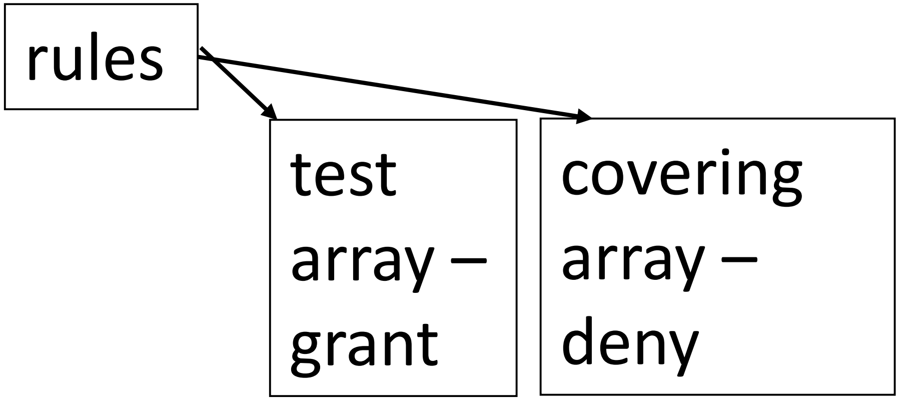
rules

0 1 0 0 1 1 1

model checker

0 1 0 0 1 1 1 grant

pseudo-exhaustive:



rules

test array – grant

covering array – deny

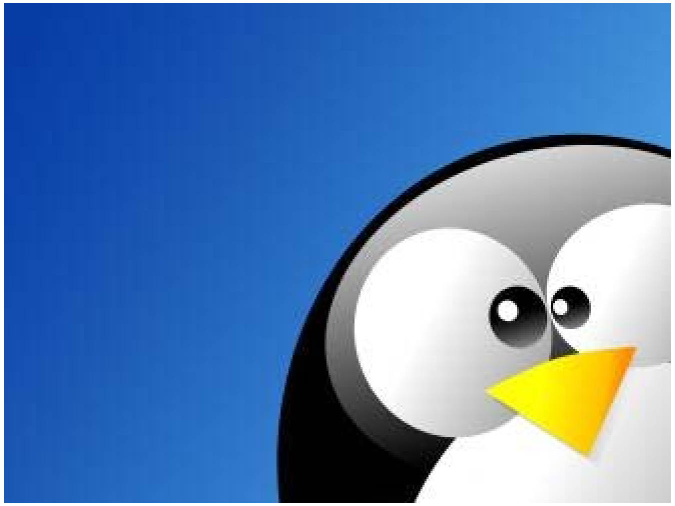
Learning and Applying Combinatorial Testing

Web sites:

- csrc.nist.gov/acts – tutorials, technical papers, free and open source tools
- pairwise.org - tutorials, links to free and open source tools
- Air Force Institute of Technology – statistical testing for systems and software http://www.afit.edu/STAT/page.cfm?page=713

Summary

- Software failures are triggered by a small number of factors interacting – 1 to 6 in known cases
- Therefore covering all t-way combinations, for small t, is pseudo-exhaustive and provides strong assurance
- Strong *t*-way interaction coverage can be provided using covering arrays
- Combinatorial testing is practical today using existing tools for real-world software
- Combinatorial methods have been shown to provide significant cost savings with improved test coverage, and proportional cost savings increases with the size and complexity of problem



Please contact us if you're interested!

Rick Kuhn
kuhn@nist.gov

Raghu Kacker
raghu.kacker@nist.gov

http://csrc.nist.gov/acts