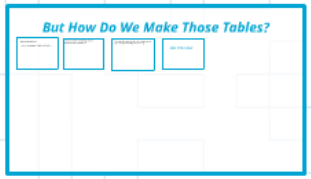
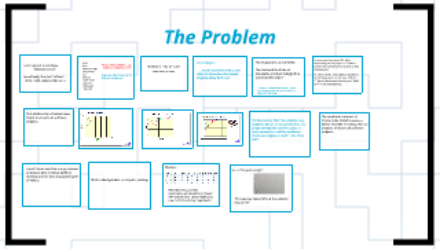
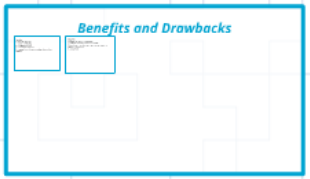
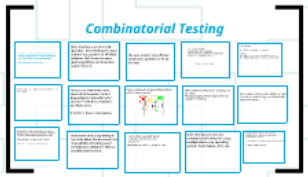
CS1699 - Lecture 17 - Pairwise & Combinatorial Testing

** ACTS and NIST imagery used with permission*



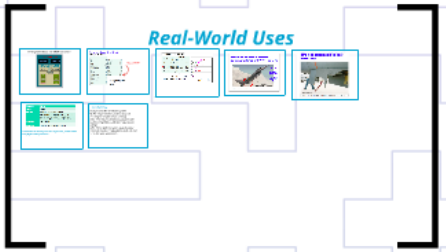


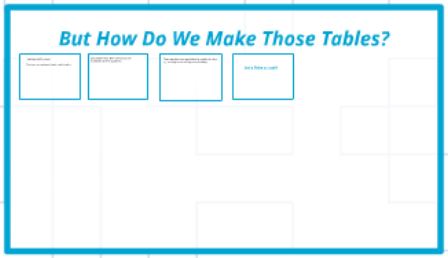


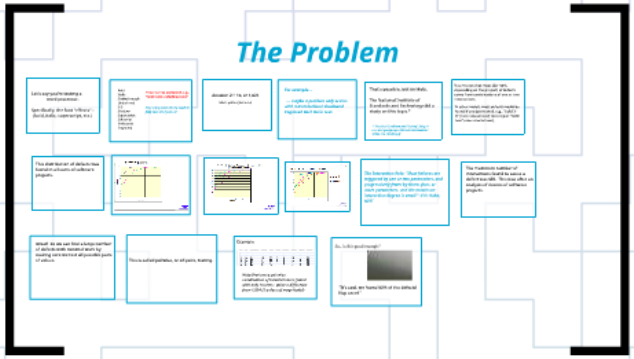


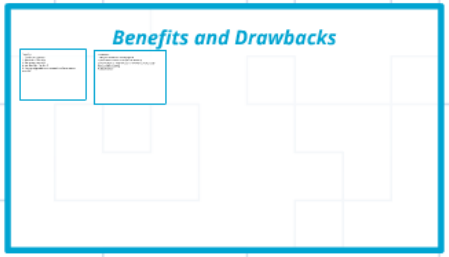
CS1699 - Lecture 17 - Pairwise & Combinatorial Testing

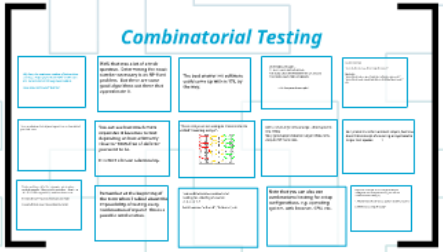
** ACTS and NIST imagery used with permission*











The Problem

Let's say you're testing a word processor.

Specifically, the font "effects" - (bold, italic, superscript, etc.)

Bold
Italic
Strikethrough
Underlined
3-D
Shadow
Superscript
Subscript
Embossed
Engraved

These can be combined, e.g., "bold italic underlined text"

How many tests do we need to fully test this feature?

Answer: 2 ^ 10, or 1,024

(that's quite a few tests!)

For example...

... maybe a problem only occurs with 3-D Underlined Shadowed Engraved Bold Italic text.

That's possible, but unlikely.

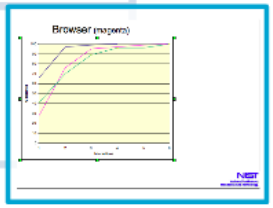
The National Institute of Standards and Technology did a study on this topic.*

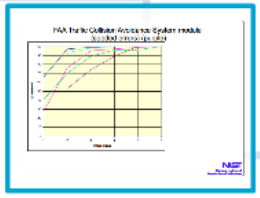
* "Practical Combinatorial Testing", http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf

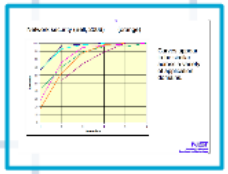
It turns out that most (50 - 90%, depending on the project) of defects come from combinations of one or two interactions.

In other words, most defects would be found if you just tested, e.g., "bold 3-D" (two interactions) text or just "bold text" (one interactions).

This distribution of defects was found in all sorts of software projects.







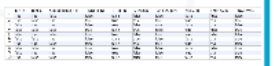
The Interaction Rule: "Most failures are triggered by one or two parameters, and progressively fewer by three, four, or more parameters, and the maximum interaction degree is small." -Eric Kuhn, NIST

The maximum number of interactions found to cause a defect was SIX. This was after an analysis of dozens of software projects.

Great! So we can find a large number of defects with minimal work by making sure we test all possible pairs of values.

This is called *pairwise*, or *all-pairs*, testing.

Example



Note that every pairwise combination of interactions is found with only 10 tests - quite a difference from 1,024 (2 orders of magnitude!)

So... is this good enough?

"It's cool, we found 90% of the defects! Hop on in!"

Let's say you're testing a word processor.

Specifically, the font "effects" - (bold, italic, superscript, etc.)

Bold
Italic
Strikethrough
Underlined
3-D
Shadow
Superscript
Subscript
Embossed
Engraved

These can be combined, e.g., "bold italic underlined text"

How many tests do we need to fully test this feature?

Answer: 2 ^ 10, or 1,024

(that's quite a few tests!)

For example...

... maybe a problem only occurs with 3-D Underlined Shadowed Engraved Bold Italic text.

That's possible, but unlikely.

The National Institute of Standards and Technology did a study on this topic.*

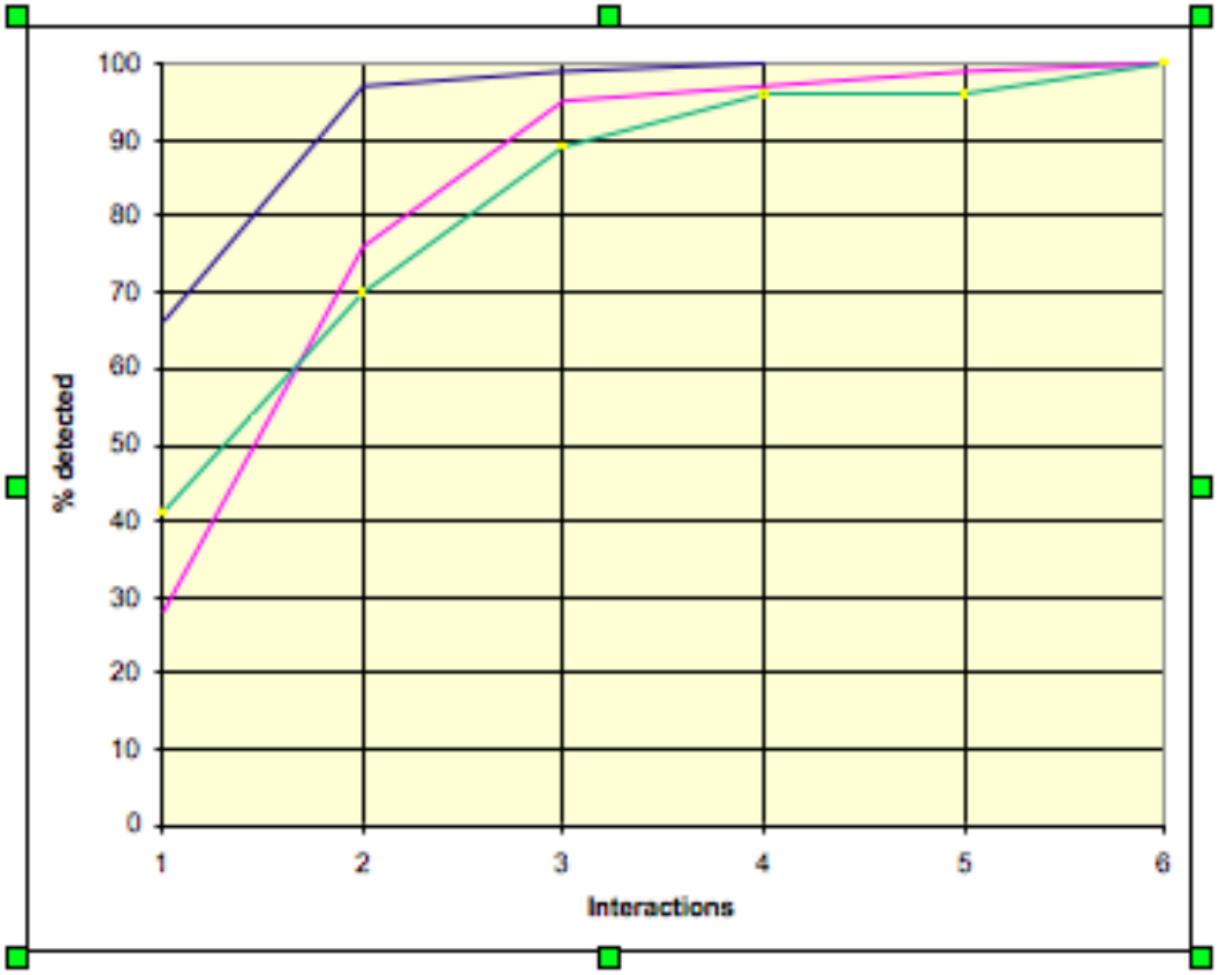
** "Practical Combinatorial Testing", http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf*

It turns out that most (50 - 90%, depending on the project) of defects come from combinations of one or two interactions.

In other words, most defects would be found if you just tested, e.g., "bold 3-D" (two interactions) text or just "bold text" (one interactions).

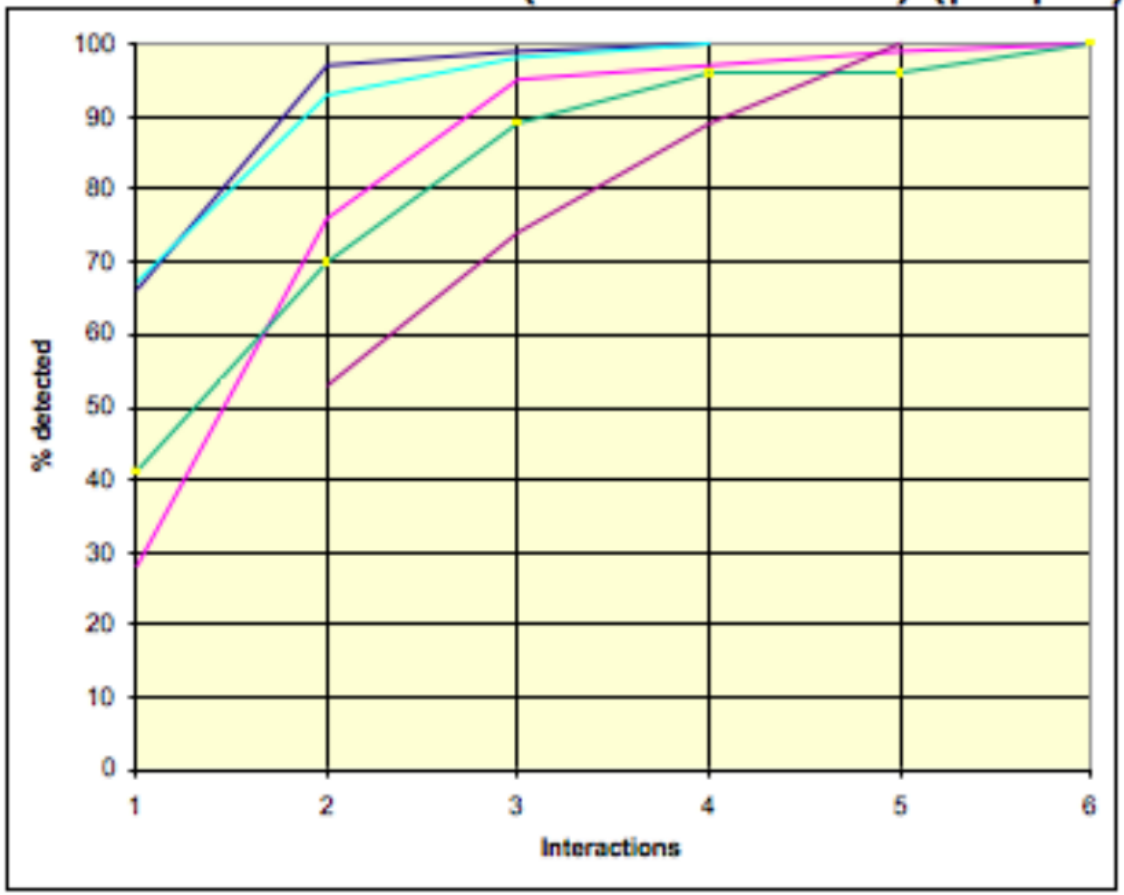
This distribution of defects was found in all sorts of software projects.

Browser (magenta)



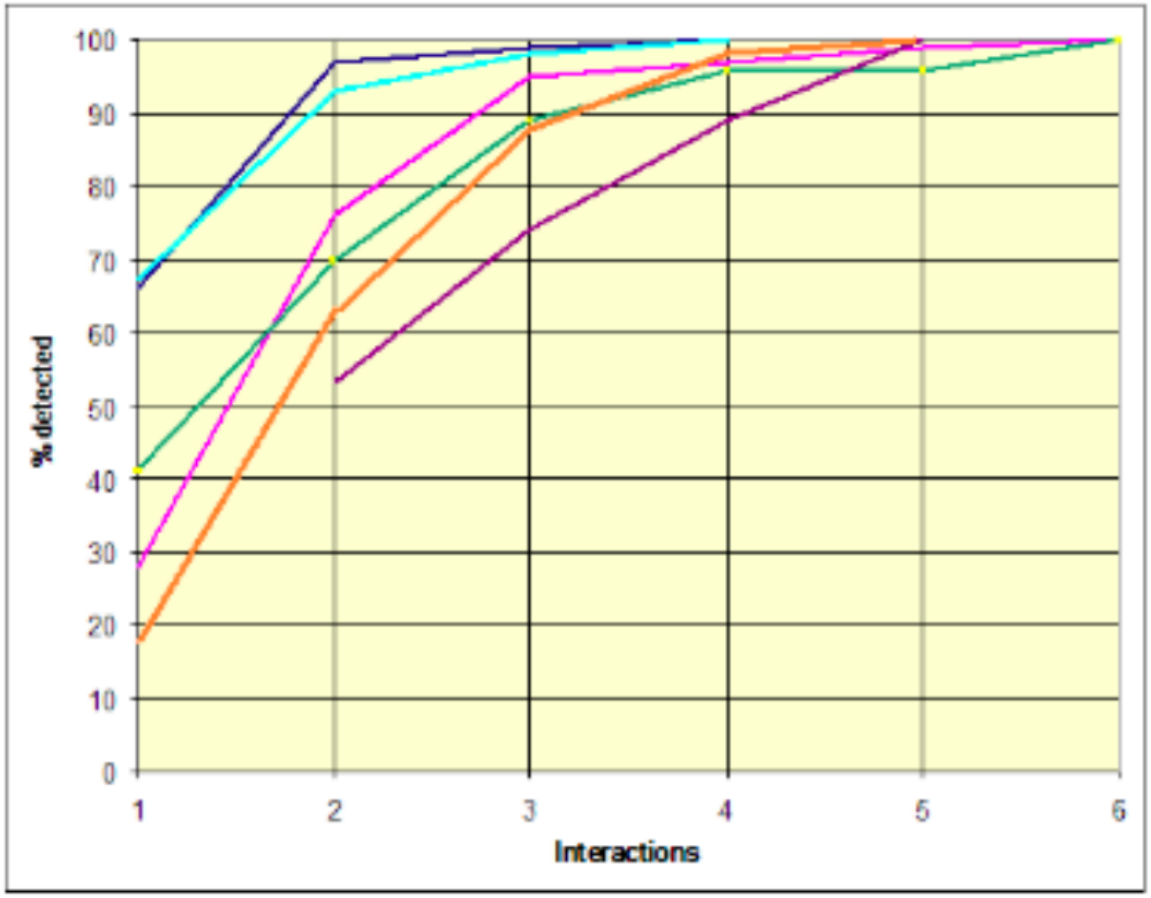
NIST
National Institute of Standards and Technology

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



NIST
National Institute of Standards and Technology

Network security (Bell, 2006) (orange)



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

NIST
National Institute of Standards and Technology

The Interaction Rule: "Most failures are triggered by one or two parameters, and progressively fewer by three, four, or more parameters, and the maximum interaction degree is small." -Eric Kuhn, NIST

The maximum number of interactions found to cause a defect was SIX. This was after an analysis of dozens of software projects.

Great! So we can find a large number of defects with minimal work by making sure we test all possible pairs of values.

This is called *pairwise*, or *all-pairs*, testing.

Example

	BOLD	ITALIC	STRIKETHROUGH	UNDERLINE	THREED	SHADOW	SUPERSCRIPT	SUBSCRIPT	EMBOSSED	ENGRAVED
1	true	true	false	false	false	false	false	false	false	false
2	true	false	true	true	true	true	true	true	true	true
3	false	true	true	false	true	false	true	false	true	false
4	false	false	false	true	false	true	false	true	false	true
5	false	true	false	true	true	false	true	true	false	false
6	false	false	true	false	false	true	false	false	true	true
7	true	true	false	false	false	true	true	true	true	false
8	false	false	true	true	true	false	false	false	false	true
9	false	true	true	false	true	false	false	true	true	true
10	true	false	false	false	false	false	true	false	true	false

Note that every pairwise combination of interactions is found with only 10 tests - quite a difference from 1,024 (2 orders of magnitude!)

So... is this good enough?

"It's cool, we found 90% of the defects! Hop on in!"

Combinatorial Testing

OK, then, the maximum number of interactions causing a defect found in the NIST studies was six. So let's test all six-way combinations.

How many tests would that be?

Well, that was a bit of a trick question. Determining the exact number necessary is an NP-Hard problem. But there are some good algorithms out there that approximate it.

The best answer my software could come up with is 178, by the way.

Interesting, though...
10 tests catch 90% of defects
178 tests catch 99.9999999%-ish of defects
1024 tests catch 100% of defects

... IF they are done right!

Pareto Principle:

"80% of effects come from 20% of causes."

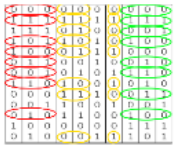
Examples:
"80% of your sales come from 20% of your customers."
"80% of your code execution time is in 20% of your code."
etc.

For a new feature, 80% of your bugs will come from 20% of your test cases.

You can see how much more expensive it becomes to test depending on how arbitrarily close to "100% free of defects" you want to be.

It is NOT a linear relationship.

These arrays we are using to make tests are called "covering arrays".



Still a relatively new concept - developed in the 1990s
Very good approximation algorithms now, despite NP hardness.

Our problem earlier seemed simple, but how does this concept of covering arrays handle larger test spaces ?

Pretty well, actually! Let's imagine an airplane cockpit console. There are 34 switches. Thus 1.7 x 10^10 (17 billion) possible combinations to test.

To test all three-way interactions: 33 tests!

To test all four-way interactions: 85 tests!

Remember at the beginning of the term when I talked about the impossibility of testing every combination of inputs? This is a possible amelioration.

You could also use combinatorial testing for ordering of events:
a, b, c, d, e, f

Pairs become "a then b", "b then a", etc.

Note that you can also use combinatorial testing for setup configurations, e.g. operating system, web browser, CPU, etc.

Any time when you have a large variety of configurations or inputs to test, you can use combinatorial testing to:

1. Find all combinations to capture n-way interactions
2. Maximize testing efficiency

OK, then, the maximum number of interactions causing a defect found in the NIST studies was six. So let's test all six-way combinations.

How many tests would that be?

Well, that was a bit of a trick question. Determining the exact number necessary is an NP-Hard problem. But there are some good algorithms out there that approximate it.

The best answer my software could come up with is 178, by the way.

Interesting, though...

10 tests catch 90% of defects

178 tests catch 99.9999999%-ish of defects

1024 tests catch 100% of defects

... IF they are done right!

Pareto Principle:

"80% of effects come from 20% of causes."

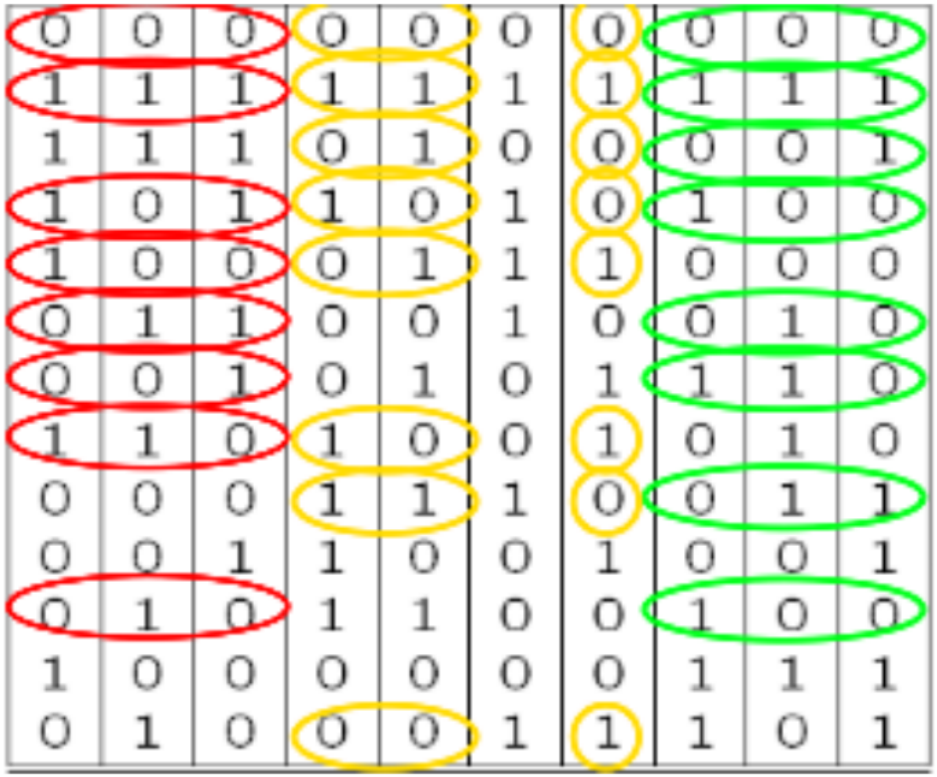
Examples:
"80% of your sales come from 20% of your customers."
"80% of your code execution time is in 20% of your code."
etc.

For a new feature, 80% of your bugs will come from 20% of your test cases.

You can see how much more expensive it becomes to test depending on how arbitrarily close to "100% free of defects" you want to be.

It is NOT a linear relationship.

These arrays we are using to make tests are called "covering arrays".



Still a relatively new concept - developed in the 1990s

Very good approximation algorithms now, despite NP hardness.

Our problem earlier seemed simple, but how does this concept of covering arrays handle larger test spaces ?

Pretty well, actually! Let's imagine an airplane cockpit console. There are 34 switches. Thus 1.7 x 10^10 (17 billion) possible combinations to test.

To test all three-way interactions: 33 tests!

To test all four-way interactions: 85 tests!

Remember at the beginning of the term when I talked about the impossibility of testing every combination of inputs? This is a possible amelioration.

Note that you can also use combinatorial testing for setup configurations, e.g. operating system, web browser, CPU, etc.

Any time when you have a large variety of configurations or inputs to test, you can use combinatorial testing to:

1. Find all combinations to capture n-way interactions

2. Maximize testing efficiency

But How Do We Make Those Tables?

Definitely NOT by hand.

These are not artisanal, hand-crafted tables.

One possible tool - NIST ACTS (Advanced Combinatorial Testing System)

There are other ones specialized for certain domains, e.g. security access settings or web testing.

Let's Take a Look!

Definitely NOT by hand.

These are not artisanal, hand-crafted tables.

One possible tool - NIST ACTS (Advanced Combinatorial Testing System)

There are other ones specialized for certain domains, e.g. security access settings or web testing.

Let's Take a Look!

Real-World Uses

Testing the DOM for the WWW Consortium

Document Object Model Events
Original test set:

36,626 tests!

Exhaustive testing of equivalence class values

Combinatorial test set:

All failures found using < 6% of original exhaustive test set

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

LANTIRN
Low Altitude Navigation & Targeting Infrared for Night

LANTIRN Pod Location

Ventral Fin

It's not supposed to look like this:

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

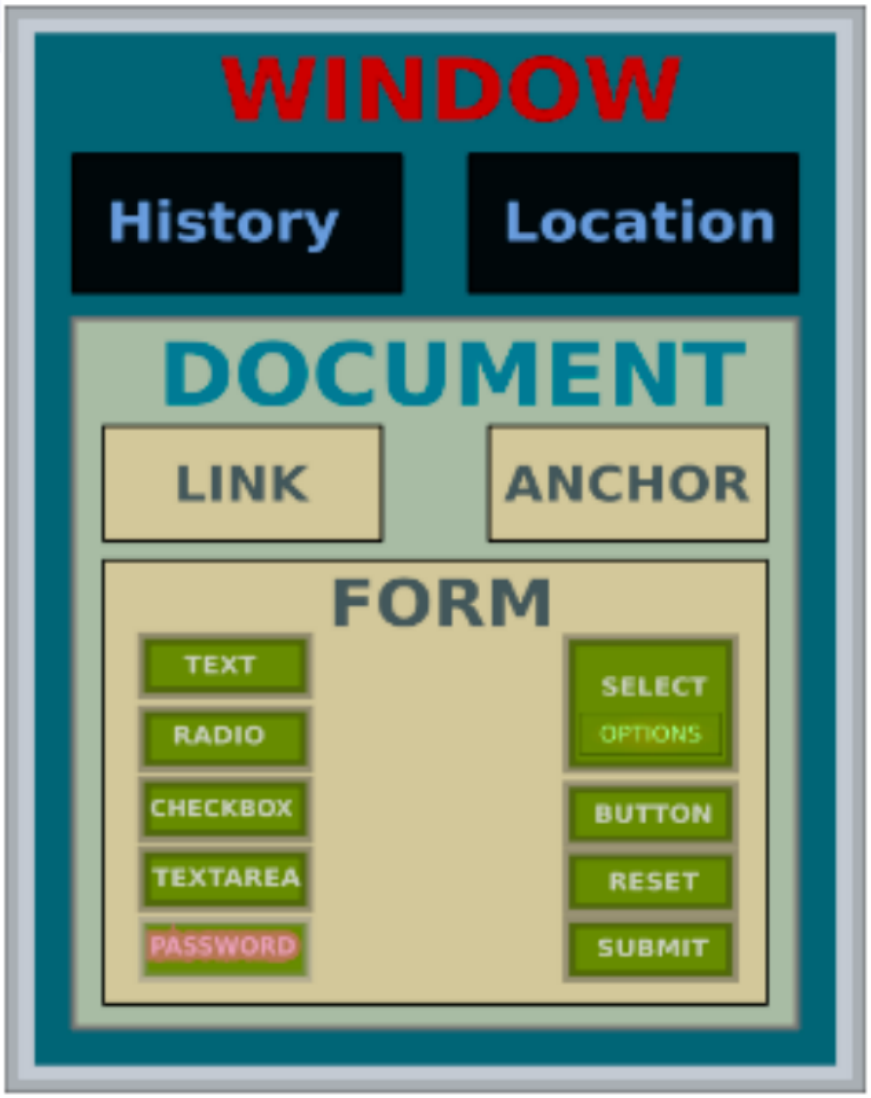
Parameter	Value
Aircraft	15, 40
Altitude	5k, 10k, 15k, 20k, 30k, 40k, 50k
Maneuver	In-speed throttle, slow accel/decel, L/R 5 deg side slip, L/R 360 roll, R/L 5 deg side slip, Mod accel/decel, R-L-R-L banking, Hi-speed to Low, 360 nose roll
Mach (100's)	40, 50, 60, 70, 80, 90, 100, 110, 120

Combinatorial testing was able to find the problem with less flight- and expert-time.

Security Testing

Analysis of buffer overflows by NIST:
94.7% - Single variable problem (e.g. not checking for length of ftps:// string)
4.9% - Two-way interaction (e.g. a particular search string with a particular replacement string)
0.4% - Three-way interaction (e.g. directory traversal enabled, magic_quotes enabled, and "../" in the page parameter)

Testing the DOM for the WWW Consortium



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

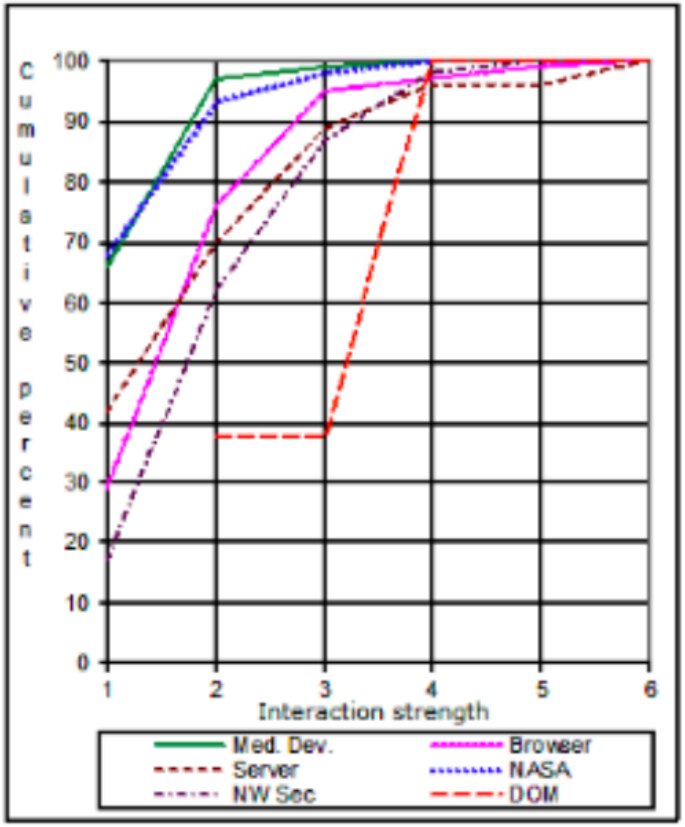
36,626 tests!

Exhaustive testing of equivalence class values

Combinatorial test set:

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 exhaustive test set



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

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

It's not supposed to look like this:

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

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

Combinatorial testing was able to find the problem with less flight- and expert-time.

Security Testing

Analysis of buffer overflows by NIST:
94.7% - Single variable problem (e.g. not checking for length of ftps:// string)
4.9% - Two-way interaction (e.g. a particular search string with a particular replacement string)
0.4% - Three-way interaction (e.g. directory traversal enabled, magic_quotes enabled, and ".." in the page parameter)

Benefits and Drawbacks

Benefits:
1. Great test coverage
2. Maximize efficiency
3. Can gauge coverage
4. Can turn dial "up to 11"
5. Very good growth rates as number of interactions increase

Drawbacks:
1. May be overkill for small projects
2. Extra time to make tests (albeit minimal)
3. New features - may have to re-run and re-create tests instead of just adding
4. Automation?

Benefits:

1. Great test coverage
2. Maximize efficiency
3. Can gauge coverage
4. Can turn dial "up to 11"
5. Very good growth rates as number of interactions increase

Drawbacks:

1. May be overkill for small projects
2. Extra time to make tests (albeit minimal)
3. New features - may have to re-run and re-create tests instead of just adding
4. Automation?

You could also use combinatorial testing for ordering of events:
a, b, c, d, e, f

Pairs become "a then b", "b then a", etc.

CS1699 - Lecture 17 - Pairwise & Combinatorial Testing

** ACTS and NIST imagery used with permission*