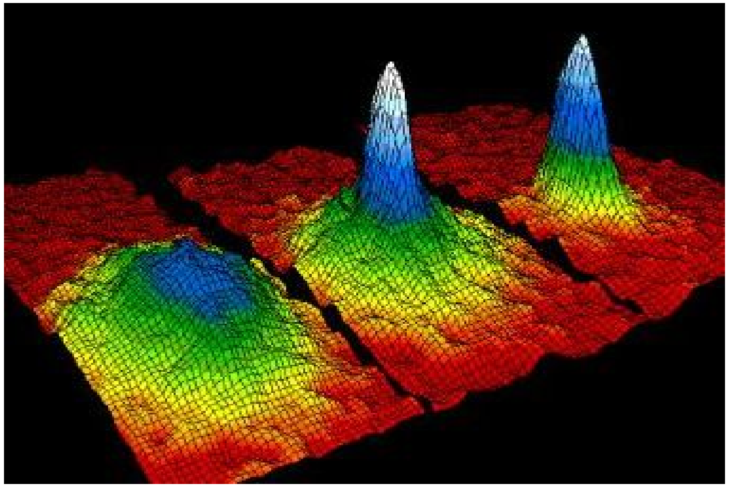
Combinatorial Testing

Rick Kuhn (NIST)
Raghu Kacker (NIST)
Sreedevi Sampath (UMBC)

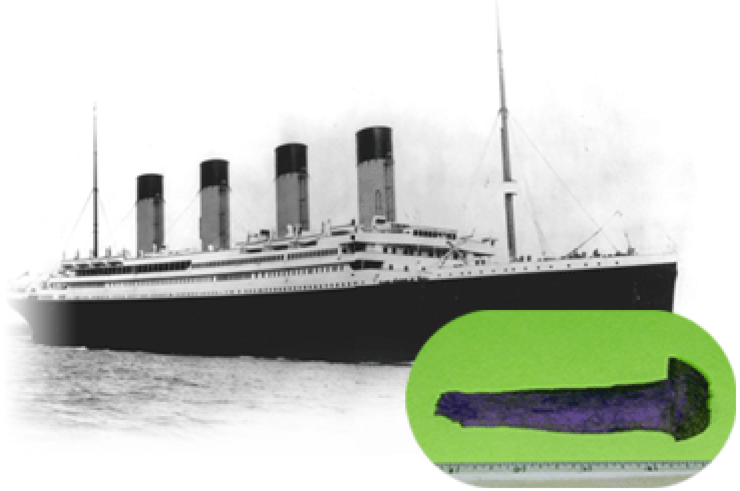
Aberdeen Proving Grounds Tutorial May 17, 2010

What is NIST and why are we doing this?

- A US Government agency
- The nation's measurement and testing laboratory – 3,000 scientists, engineers, and support staff including 3 Nobel laureates



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



Analysis of engineering failures, including buildings, materials, and software. This work applies knowledge gained to improve testing.

NIST
National Institute of
Standards and Technology

Tutorial Overview

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

What is combinatorial testing?

THIS PART IS BASED ON PROF ADITYA P. MATHUR’S SLIDES BASED ON HIS BOOK

Foundations of Software Testing

Chapter 4: Test Generation: Combinatorial Designs

Test configuration

- Software applications are often designed to work in a variety of environments. Combinations of factors such as the operating system, network connection, and hardware platform, lead to a variety of environments.

- An environment is characterized by combination of hardware and software.

- Each environment corresponds to a given set of values for each factor, known as a test configuration.

Test configuration: Example

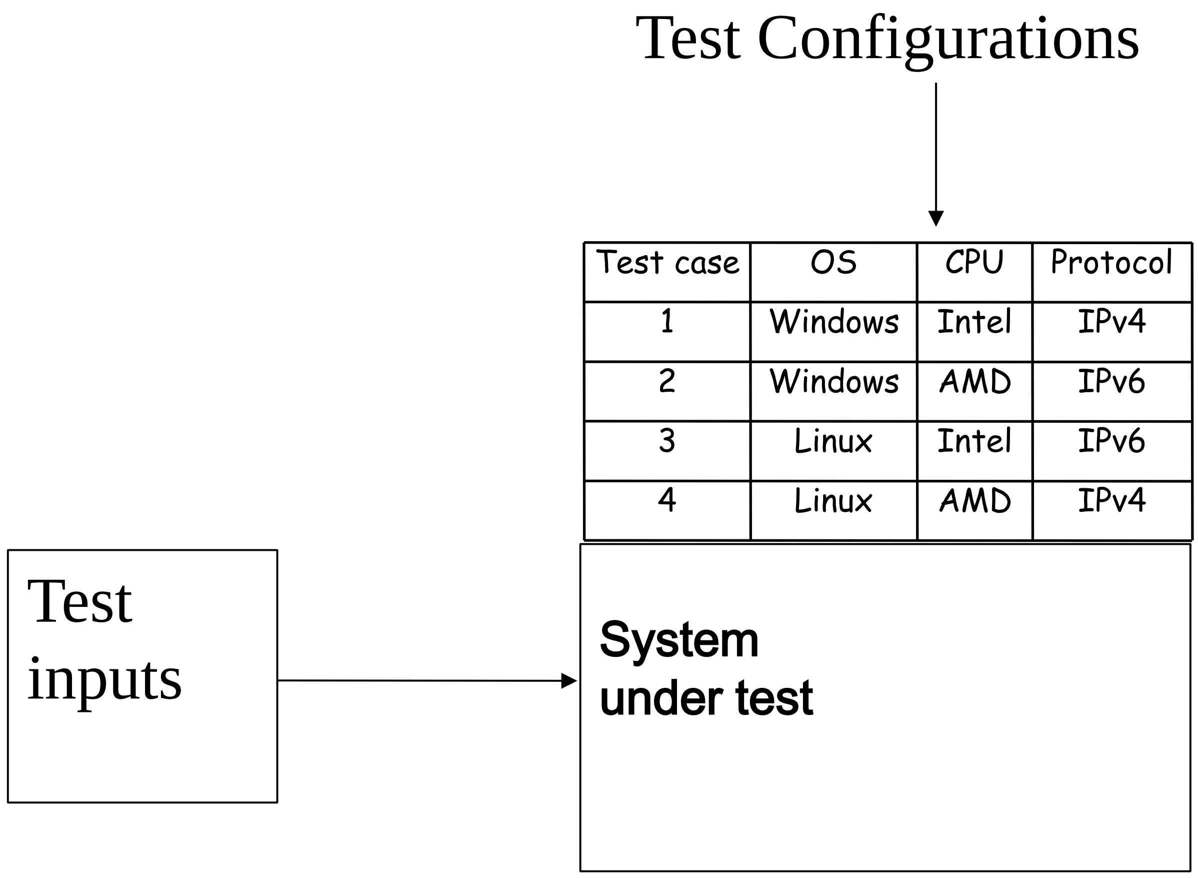
- Windows XP, Dial-up connection, and a PC with 512MB of main memory, is one possible configuration.

- Different versions of operating systems and printer drivers, can be combined to create several test configurations for a printer.

- To ensure high reliability across the intended environments, the application must be tested under as many test configurations, or environments, as possible.

The number of such test configurations could be exorbitantly large making it impossible to test the application exhaustively.

Two scopes of combinatorial testing



Test Configurations

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

Test inputs

System under test

Modeling: Input and configuration space [1]

The configuration space of P consists of all possible settings of the environment variables under which P could be used.

Similarly, the input space of a program P consists of *k*-tuples of values that could be input to P during execution.

Example: Consider program P that takes two integers x>0 and y>0 as inputs. The input space of P is the set of all pairs of positive non-zero integers.

Modeling: Input and configuration space [2]

Now suppose that this program is intended to be executed under the Windows and the MacOS operating system, through the Netscape or Safari browsers, and must be able to print to a local or a networked printer.

The configuration space of P consists of triples (X, Y, Z) where X represents an operating system, Y a browser, and Z a local or a networked printer.

Factors (Parameteres)
Levels (Values)

Consider a program P that takes *k* inputs or has *k* configurations corresponding to variables X_1, X_2, ..X_k. We refer to the inputs or configuration variables as factors or parameters.

Let us assume that each factor may be set at any one from a total of c_i, $1 \leq i \leq v$ values. Each value assignable to a factor is known as a level (value).

|F| refers to the number of levels for factor F.

Factor (parameter) combinations

A set of values, one for each factor, is known as a factor combination.

For example, suppose that program P has two input variables X and Y. Let us say that during an execution of P, X and Y may each assume a value from the set {a, b, c} and {d, e, f}, respectively.

Thus we have 2 factors and 3 levels for each factor. This leads to a total of $3^2 = 9$ factor combinations, namely (a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e), and (c, f).

Factor combinations: Too large?

In general, for $\underline{k}$ factors with each factor assuming a value from a set of $\underline{v}$ values, the total number of factor combinations is $\underline{v^k}$.

Suppose now that each factor combination yields one test case. For many programs, the number of tests generated for exhaustive testing could be exorbitantly large.

For example, if a program has 15 factors with 4 levels each, the total number of tests is $4^{15} \sim 10^9$. *Executing a billion tests might be impractical for many software applications*.

Example: Pizza Delivery Service (PDS) [1]

A PDS takes orders online, checks for their validity, and schedules Pizza for delivery.

A customer is required to specify the following four items as part of the online order: Pizza size, Toppings list, Delivery address and a home phone number. Let us denote these four factors by S, T, A, and P, respectively.

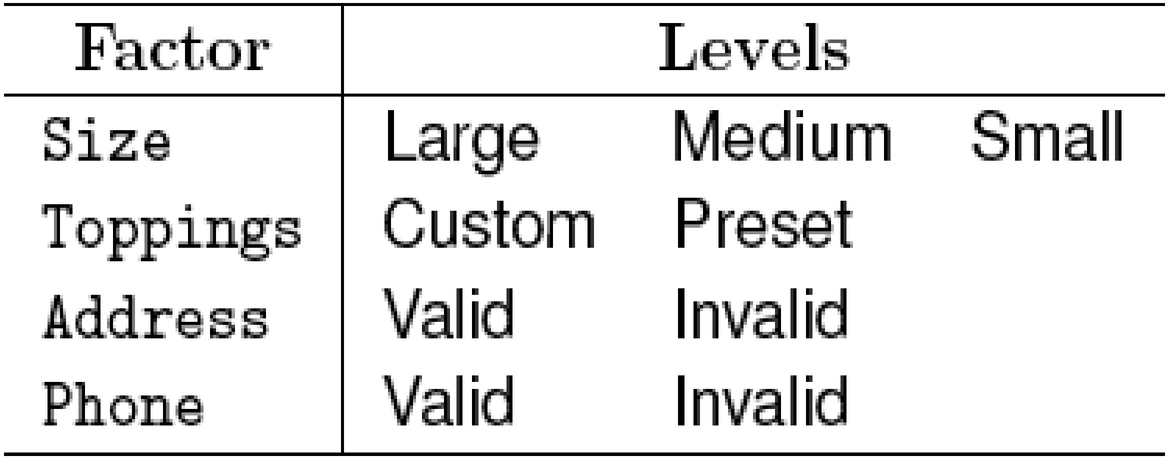
Pizza Delivery Service (PDS): Specs

Suppose now that there are three varieties for size: Large, Medium, and Small.

There is a list of 6 toppings from which to select. In addition, the customer can customize the toppings.

The delivery address consists of customer name, one line of address, city, and the zip code. The phone number is a numeric string possibly containing the dash (``--") separator.

PDS: Input space model



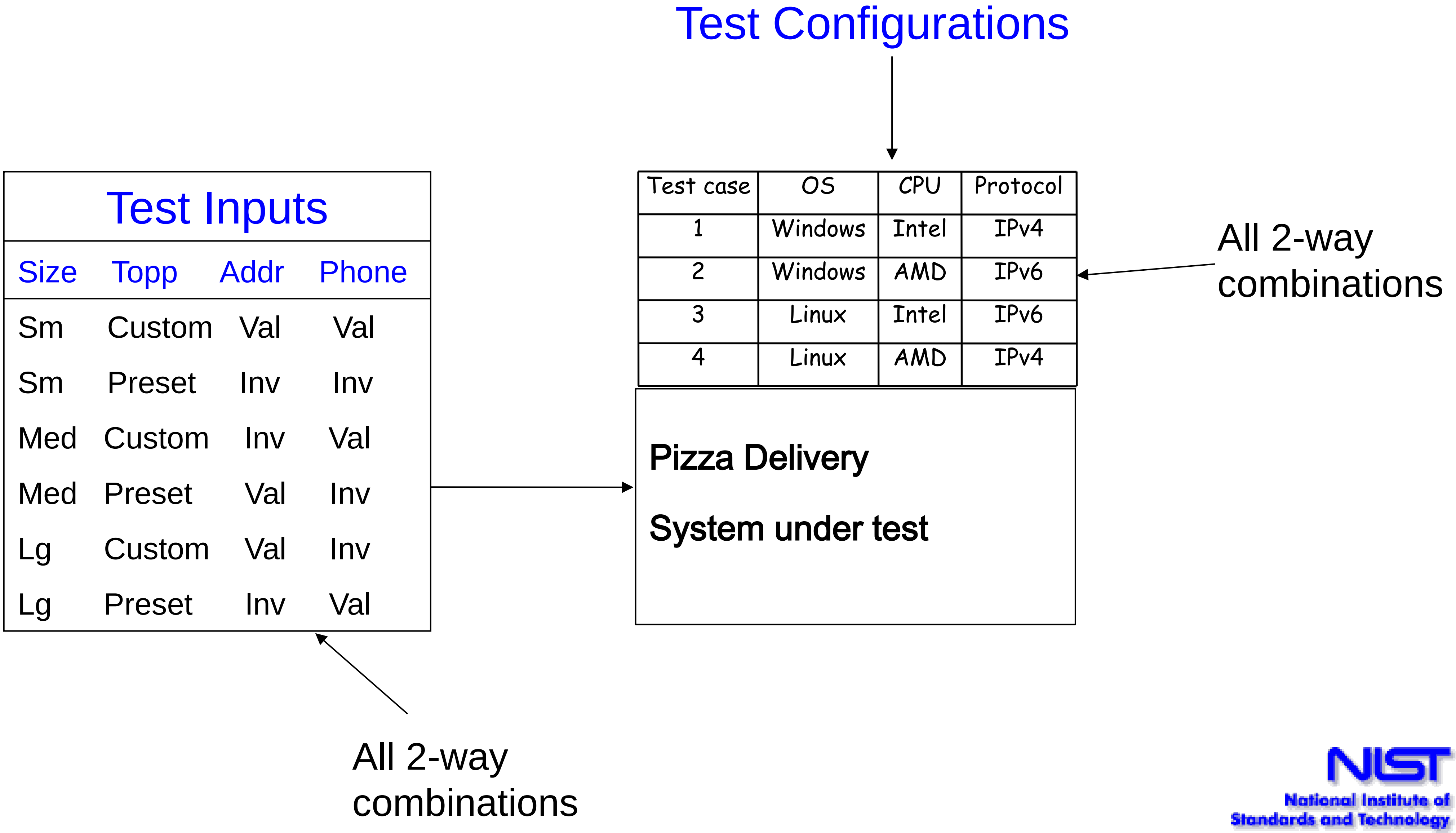
Factor	Levels		
Size	Large	Medium	Small
Toppings	Custom	Preset	
Address	Valid	Invalid	
Phone	Valid	Invalid	

The total number of factor combinations is $3^1 x 2^3 = 24$.

Suppose we consider 6+1 = 7 levels for Toppings. Number of combinations = $7^1 x 3^1 x 2^2 = 84$.

Different types of values for Address and Phone number will further increase the combinations

Two scopes of combinatorial testing



Test Configurations

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

All 2-way combinations

Test Inputs

Size	Topp	Addr	Phone
Sm	Custom	Val	Val
Sm	Preset	Inv	Inv
Med	Custom	Inv	Val
Med	Preset	Val	Inv
Lg	Custom	Val	Inv
Lg	Preset	Inv	Val

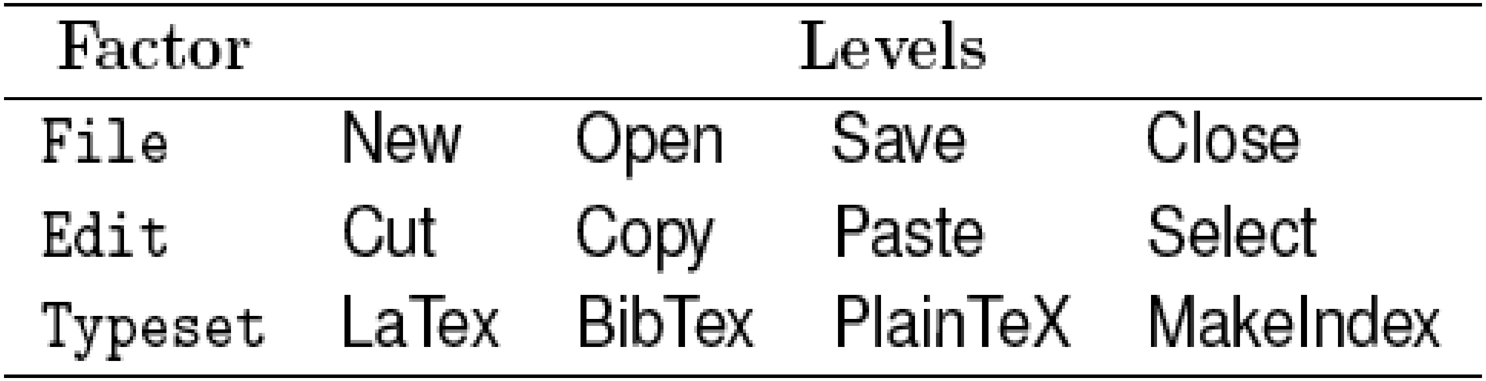
Pizza Delivery

System under test

All 2-way combinations

Example: Testing a GUI

The Graphical User Interface of application T consists of three menus labeled File, Edit, and Format.



Factor	Levels			
File	New	Open	Save	Close
Edit	Cut	Copy	Paste	Select
Typeset	LaTex	BibTex	PlainTeX	MakeIndex

We have three factors in T. Each of these three factors can be set to any of four levels. Thus we have a total 4^3=64 factor combinations.

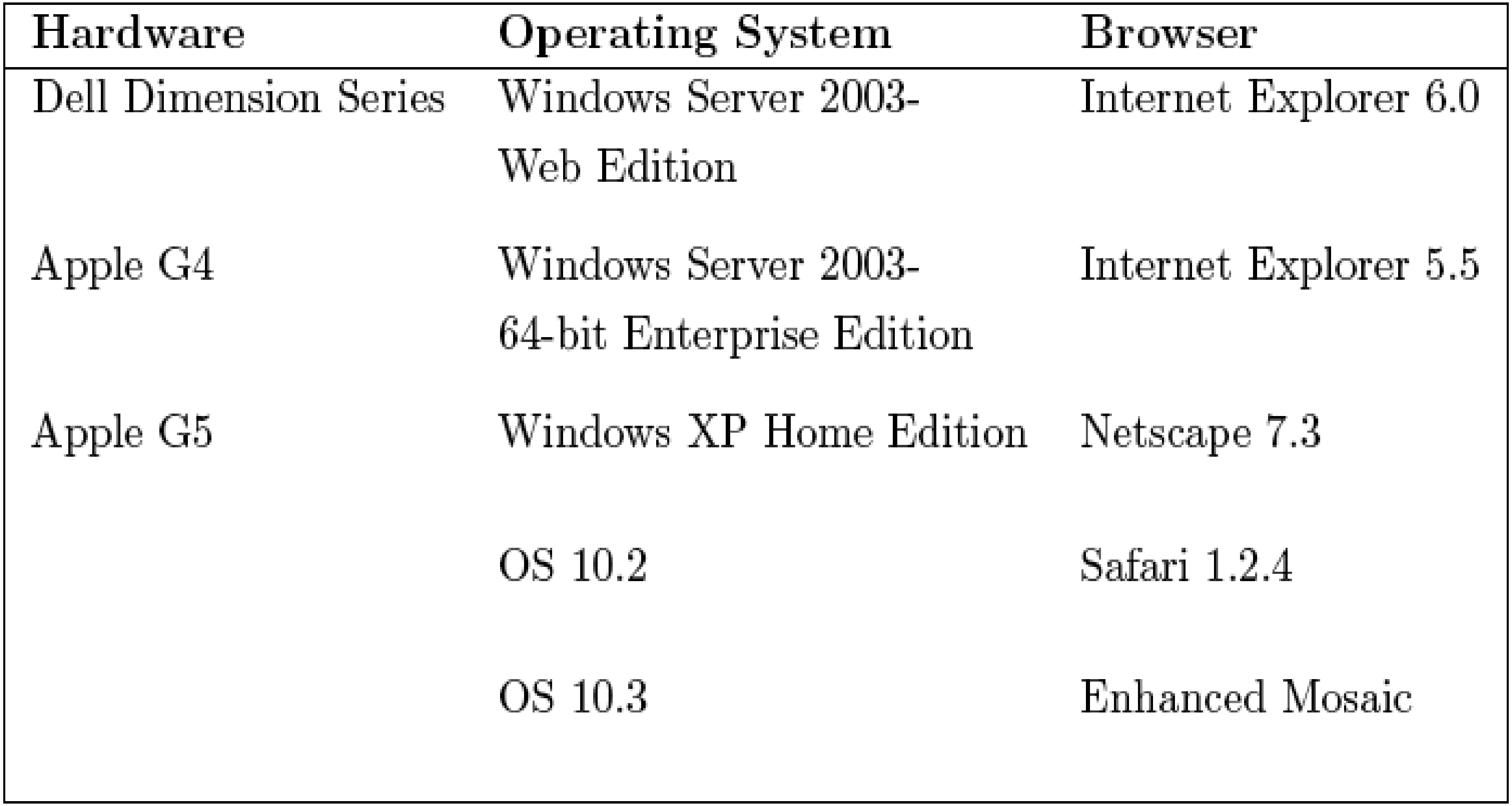
Example: Compatibility testing

There is often a need to test a web application on different platforms to ensure that any claim such as “Application X can be used under Windows and Mac OS X” are valid.

Here we consider a combination of hardware, operating system, and a browser as a platform. Let X denote a Web application to be tested for compatibility.

Given that we want X to work on a variety of hardware, OS, and browser combinations, it is easy to obtain three factors, i.e. hardware, OS, and browser.

Compatibility testing: Factor levels



Hardware	Operating System	Browser
Dell Dimension Series	Windows Server 2003-Web Edition	Internet Explorer 6.0
Apple G4	Windows Server 2003-64-bit Enterprise Edition	Internet Explorer 5.5
Apple G5	Windows XP Home Edition	Netscape 7.3
	OS 10.2	Safari 1.2.4
	OS 10.3	Enhanced Mosaic

Compatibility testing: Combinations

There are 75 factor combinations. However, some of these combinations are infeasible.

For example, Mac OS10.2 is an OS for the Apple computers and not for the Dell Dimension series PCs. Similarly, the Safari browser is used on Apple computers and not on the PC in the Dell Series.

While various editions of the Windows OS can be used on an Apple computer using an OS bridge such as the Virtual PC, we assume that this is not the case for testing application X.

Compatibility testing: Reduced combinations

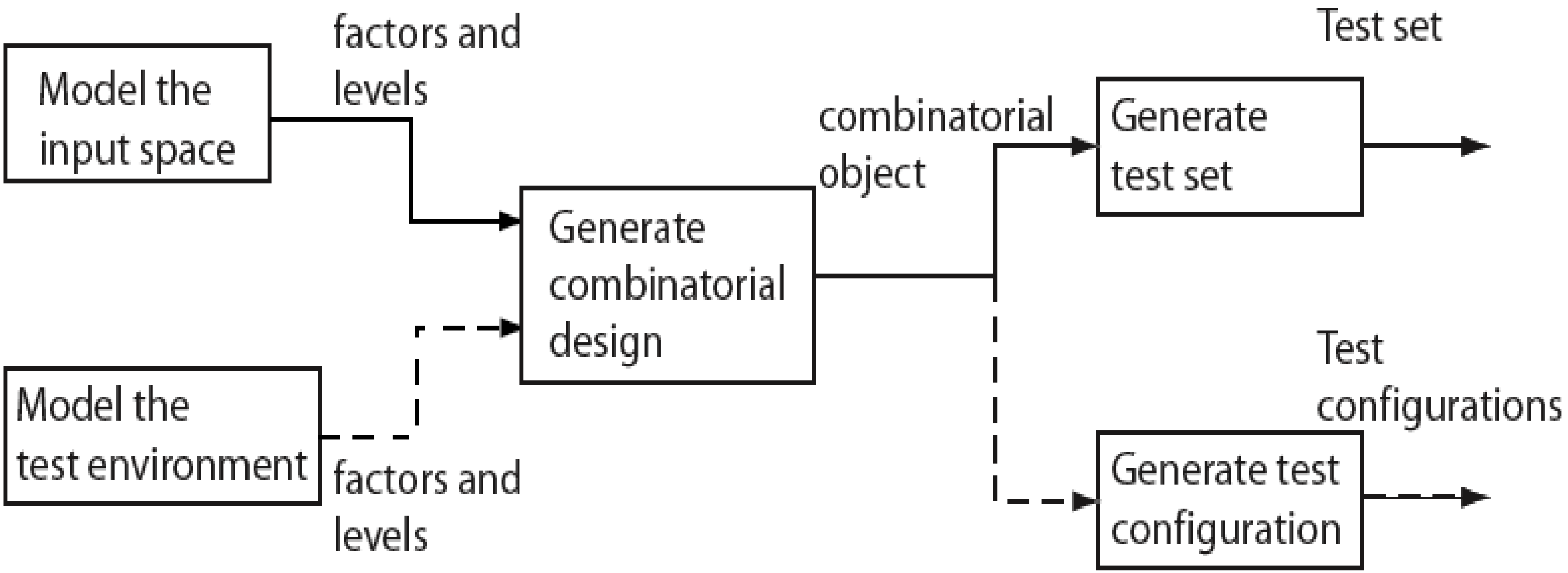
The discussion above leads to a total of 40 infeasible factor combinations corresponding to the hardware-OS combination and the hardware-browser combination. Thus in all we are left with 35 platforms on which to test X.

Note that there is a large number of hardware configurations under the Dell Dimension Series. These configurations are obtained by selecting from a variety of processor types, e.g. Pentium versus Athelon, processor speeds, memory sizes, and several others.

Compatibility testing: Reduced combinations-2

While testing against all configurations will lead to more thorough testing of application X, it will also increase the number of factor combinations, and hence the time to test.

Combinatorial test design process



Modeling of input space or the environment is not exclusive and one might apply either one or both depending on the application under test.

Combinatorial test design process: steps

Step 1: Model the input space and/or the configuration space. The model is expressed in terms of factors (parameters) and their respective levels (values)

Step 2: The model is input to a combinatorial design procedure to generate a combinatorial object which is simply an array of factors and levels. Such an object is also known as a factor covering design.

Step 3: The combinatorial object generated is used to design a test set or a test configuration as the requirement might be.

Steps 2 and 3 can be automated.

Combinatorial test design process: summary

Combination of factor levels is used to generate one or more test cases. For each test case, the sequence in which inputs are to be applied to the program under test must be determined by the tester.

Further, the factor combinations do not indicate in any way the sequence in which the generated tests are to be applied to the program under test. This sequence too must be determined by the tester.

The sequencing of tests generated by most test generation techniques must be determined by the tester and is not a unique characteristic of test generated in combinatorial testing

Fault model

Faults aimed at by the combinatorial design techniques are known as interaction faults.

We say that an interaction fault is triggered when a certain combination of $t \geq 1$ input values causes the program containing the fault to enter an invalid state.

Of course, this invalid state must propagate to a point in the program execution where it effect is observable and hence is said to reveal the fault.

t-way interaction faults

Faults triggered by some value of one input variable, i.e. t = 1, regardless of the values of other input variables, are known as simple faults.

For t = 2, the faults are known as pairwise interaction faults.

In general, for any arbitrary value of t, the faults are known as t--way interaction faults.

Goal reviewed

The goal of the test generation techniques discussed here is to generate a sufficient number of runs such that tests generated from these runs reveal all t-way faults in the program under test

Rick Kuhn [2001, 2002, 2004, 2006] shows that testing for pairwise (t = 2) interaction faults may not be sufficient;

However empirical evidence suggests that testing up to t = 6 gives reasonable assurance in most cases

Goal reviewed

The number of such runs increases with the value of *t*. In many situations, *t* is set to 2 and hence the tests generated are expected to reveal pairwise interaction faults.

Of course, while generating t-way runs, one automatically generates some t+1, t+2, .., t+k-1, and k-way runs also. Hence, there is always a chance that runs generated with t = 2 reveal some higher level interaction faults.

Statistical Approaches

Tests generated from statistical approaches (design of experiments, fractional factorials, Latin squares, orthogonal arrays) generally cover pairwise interaction

Statistical approaches do not give assurance of higher than pairwise (t = 2) interaction coverage.

Orthogonal arrays

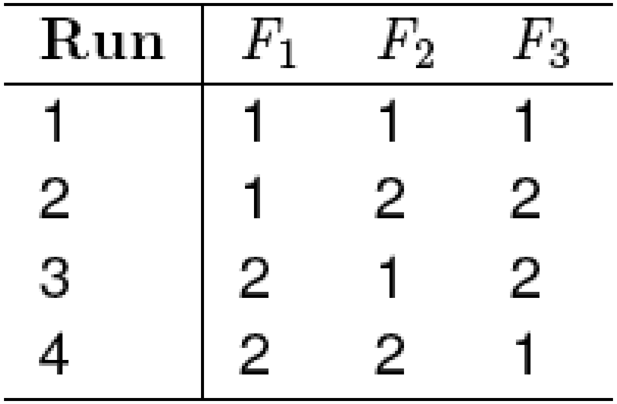
Fractional factorial designs of experiments are special class of orthogonal arrays

Latin squares, Graeco-Latin squares are special cases of orthogonal arrays

Results from test suites based on orthogonal arrays require statistical analysis

Simple orthogonal array

Examine this matrix and extract as many properties as you can:

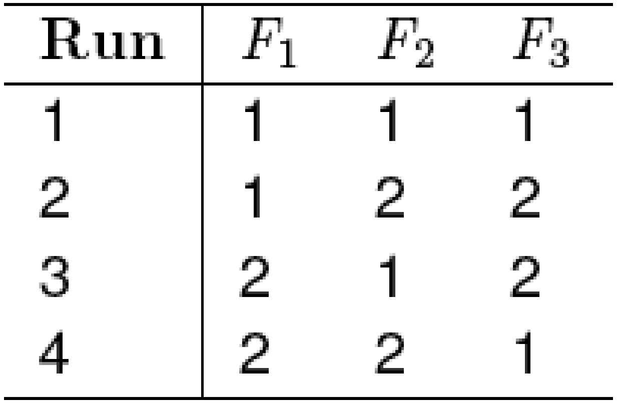


Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

An orthogonal array, such as the one above, is an N x k matrix in which the entries are from a finite set S of s symbols such that any N x t subarray contains each t-tuple exactly the same number of times. Such an orthogonal array is denoted by OA(N, k, s, t).

Orthogonal arrays: Example

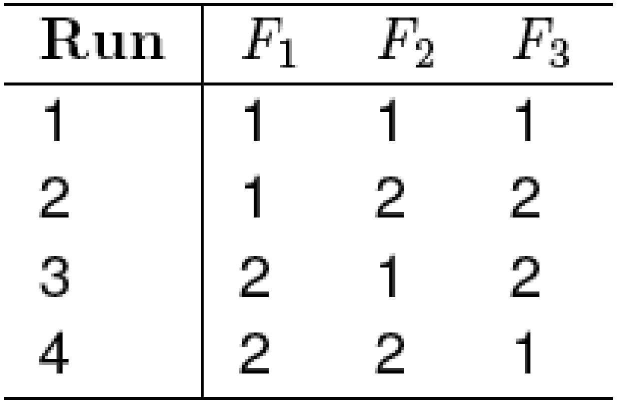
The following orthogonal array has 4 runs and has a strength of 2. It uses symbols from the set {1, 2}. This array is denoted as OA(4, 3, 2, 2). Note that the value of parameter *k* is 3 and hence we have labeled the columns as F1, F2, and F3 to indicate the three factors.



Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Orthogonal arrays: Index

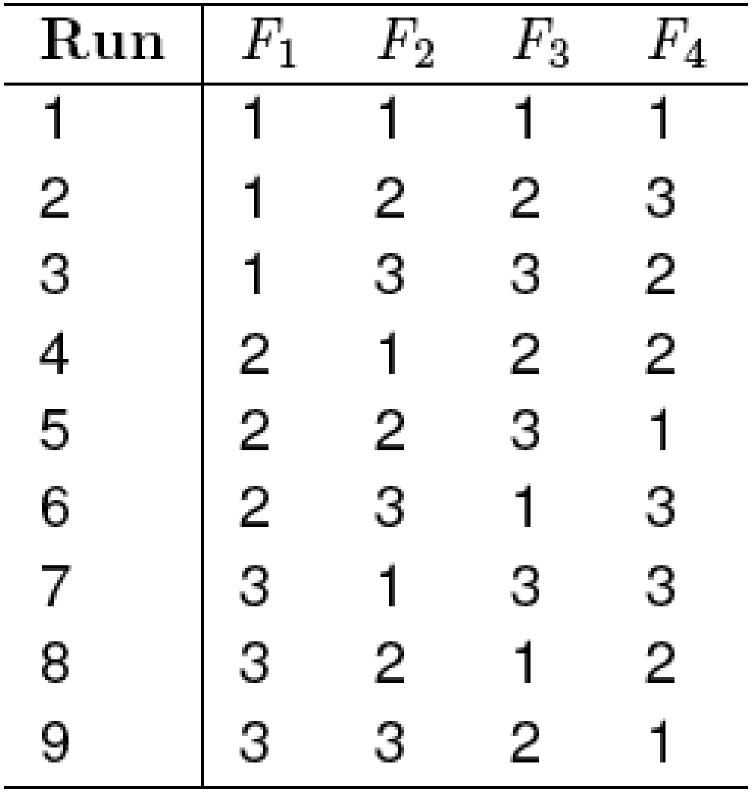
The index of an orthogonal array is denoted by λ and is equal to N/s^t. N is referred to as the number of runs and *t* as the strength of the orthogonal array.



Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

$\lambda = 4/2^2 = 1$ implying that each pair (t=2) appears exactly once ($\lambda = 1$) in any 4 x 2 sub-array. There is a total of $s^t = 2^2 = 4$ pairs given as (1, 1), (1, 2), (2, 1), and (2, 2). It is easy to verify that each of the four pairs appears exactly once in each 4 x 2 sub-array.

Orthogonal arrays: Another example



Run	F_1	F_2	F_3	F_4
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1

What kind of an OA is this?

It has 9 runs and a strength of 2. Each of the four factors can be at any one of 3 levels. This array is denoted as OA(9, 4, 3, 2) and has an index of 1.

An orthogonal array of index 1, when it exists is the most optimal (smallest size) combinatorial design

Orthogonal arrays: Alternate notations - due to Genichi Taguchi

$L_N(s^k)$ Orthogonal array of N runs where k factors take on any value from a set of s symbols.

Arrays shown earlier are $L_4(2^3)$ and $L_9(3^3)$

L_N denotes an orthogonal array of 9 runs. t, k, s are determined from the context, i.e. by examining the array itself.

Mixed level Orthogonal arrays

So far we have seen fixed (same) level orthogonal arrays. This is because the design of such arrays assumes that all factors assume values from the same set of s values.

In many practical applications, one encounters more than one factor, each taking on a different set of values. Mixed orthogonal arrays are useful in designing test configurations for such applications.

Mixed level Orthogonal arrays: Notation

$MA(N, s_1^{k_1} s_2^{k_2} \ldots s_p^{k_p}, t)$

Strength = t. Runs = N.
k1 factors at s1 levels, k2 at s2 levels, and so on.

Total factors: $\Sigma_{i=1}^{p} k_i$

Mixed level Orthogonal arrays: Index and balance

The formula used for computing the index λ of an orthogonal array does not apply to the mixed level orthogonal array as the count of values for each factor is a variable.

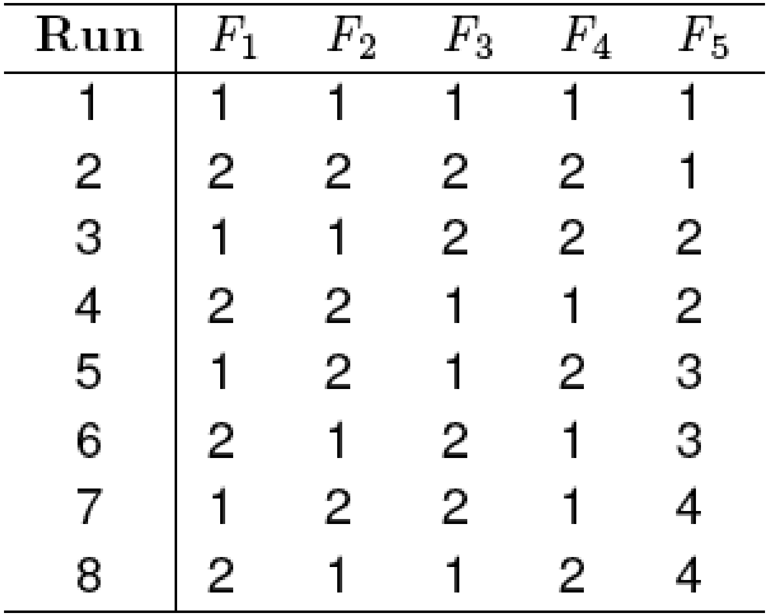
The balance property of orthogonal arrays remains intact for mixed level orthogonal arrays in that any N x t sub-array contains each t-tuple corresponding to the t columns, exactly the same number of times, which is λ.

Mixed level Orthogonal arrays: Example

$$MA(8, 2^4 4^1, 2)$$

This array can be used to design test configurations for an application that contains 4 factors each at 2 levels and 1 factor at 4 levels.

Can you identify some properties?



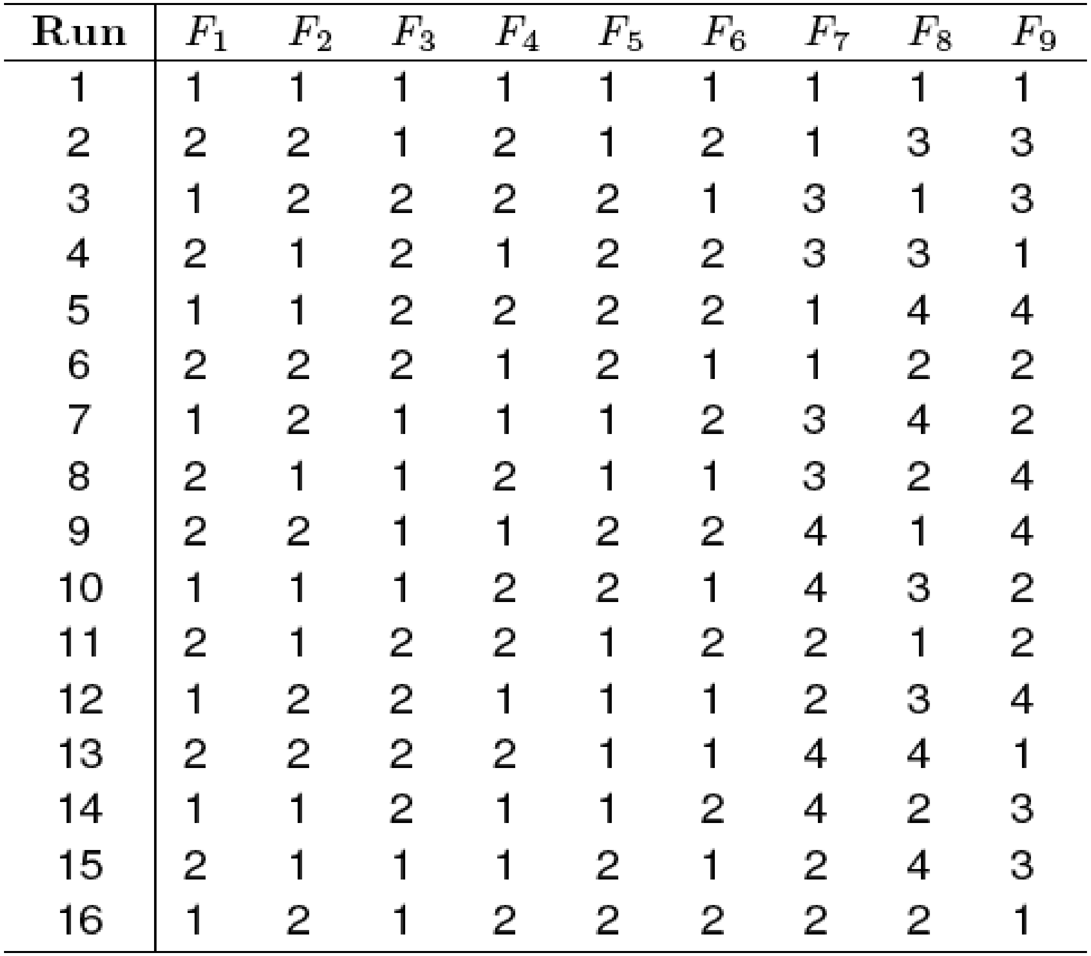
Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	2	2	2	1
3	1	1	2	2	2
4	2	2	1	1	2
5	1	2	1	2	3
6	2	1	2	1	3
7	1	2	2	1	4
8	2	1	1	2	4

Balance: In any subarray of size 8 x 2, each possible pair occurs exactly the same number of times. In the two leftmost columns, each pair occurs exactly twice. In columns 1 and 3, each pair also occurs exactly twice. In columns 1 and 5, each pair occurs exactly once.

Mixed level Orthogonal arrays: Example

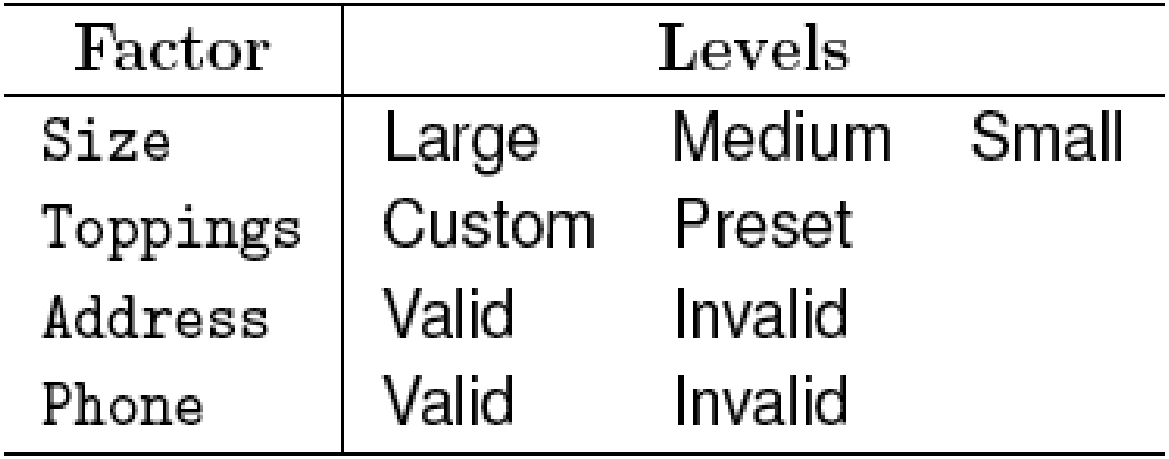
$MA(16, 2^6, 4^3, 2)$

This array can be used to generate test configurations when there are six binary factors, labeled F1 through F6 and three factors each with four possible levels, labeled F7 through F9.



Run	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
1	1	1	1	1	1	1	1	1	1
2	2	2	1	2	1	2	1	3	3
3	1	2	2	2	2	1	3	1	3
4	2	1	2	1	2	2	3	3	1
5	1	1	2	2	2	2	1	4	4
6	2	2	2	1	2	1	1	2	2
7	1	2	1	1	1	2	3	4	2
8	2	1	1	2	1	1	3	2	4
9	2	2	1	1	2	2	4	1	4
10	1	1	1	2	2	1	4	3	2
11	2	1	2	2	1	2	2	1	2
12	1	2	2	1	1	1	2	3	4
13	2	2	2	2	1	1	4	4	1
14	1	1	2	1	1	2	4	2	3
15	2	1	1	1	2	1	2	4	3
16	1	2	1	2	2	2	2	2	1

Mixed level Orthogonal arrays: Test generation: Pizza delivery

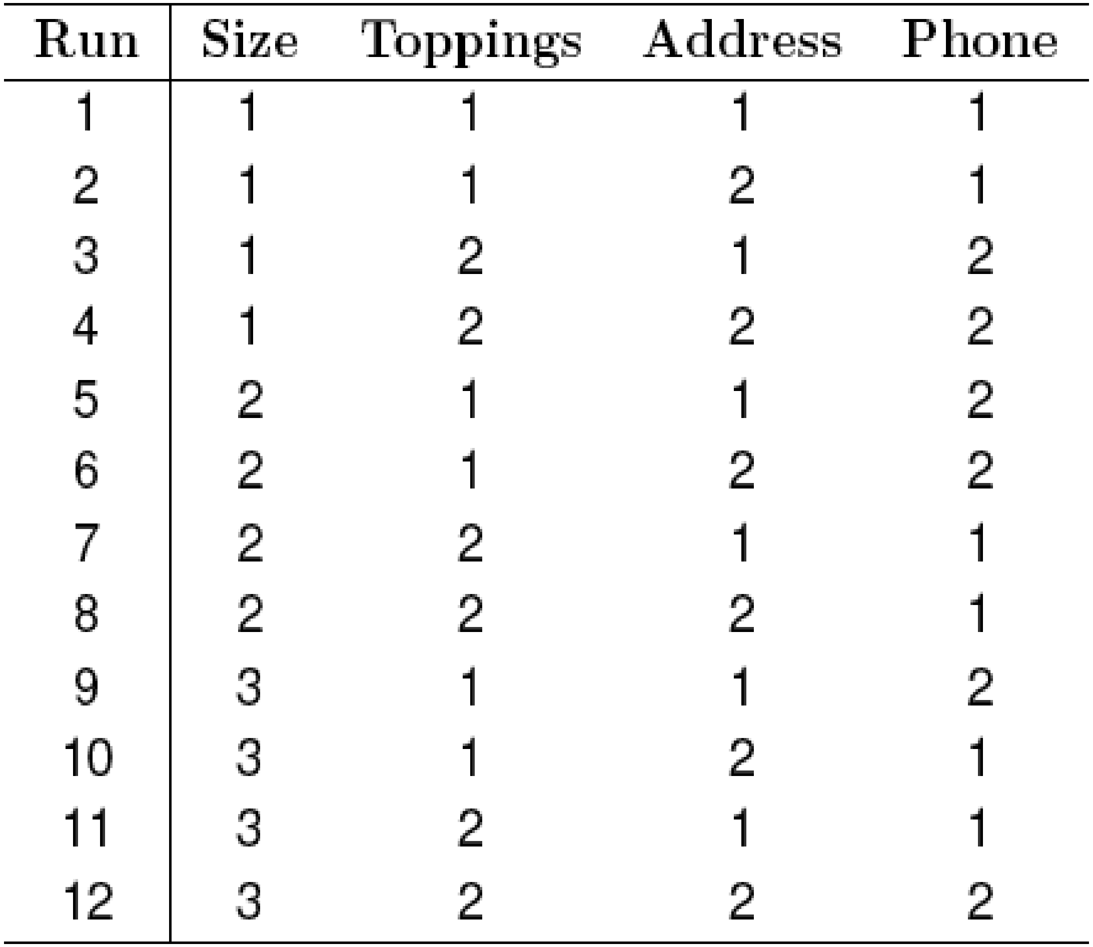


Factor	Levels		
Size	Large	Medium	Small
Toppings	Custom	Preset	
Address	Valid	Invalid	
Phone	Valid	Invalid	

We have 3 binary factors and one factor at 3 levels. Hence we can use the following array to generate test configurations:

$$MA(12, 2^3, 3^1, 2)$$

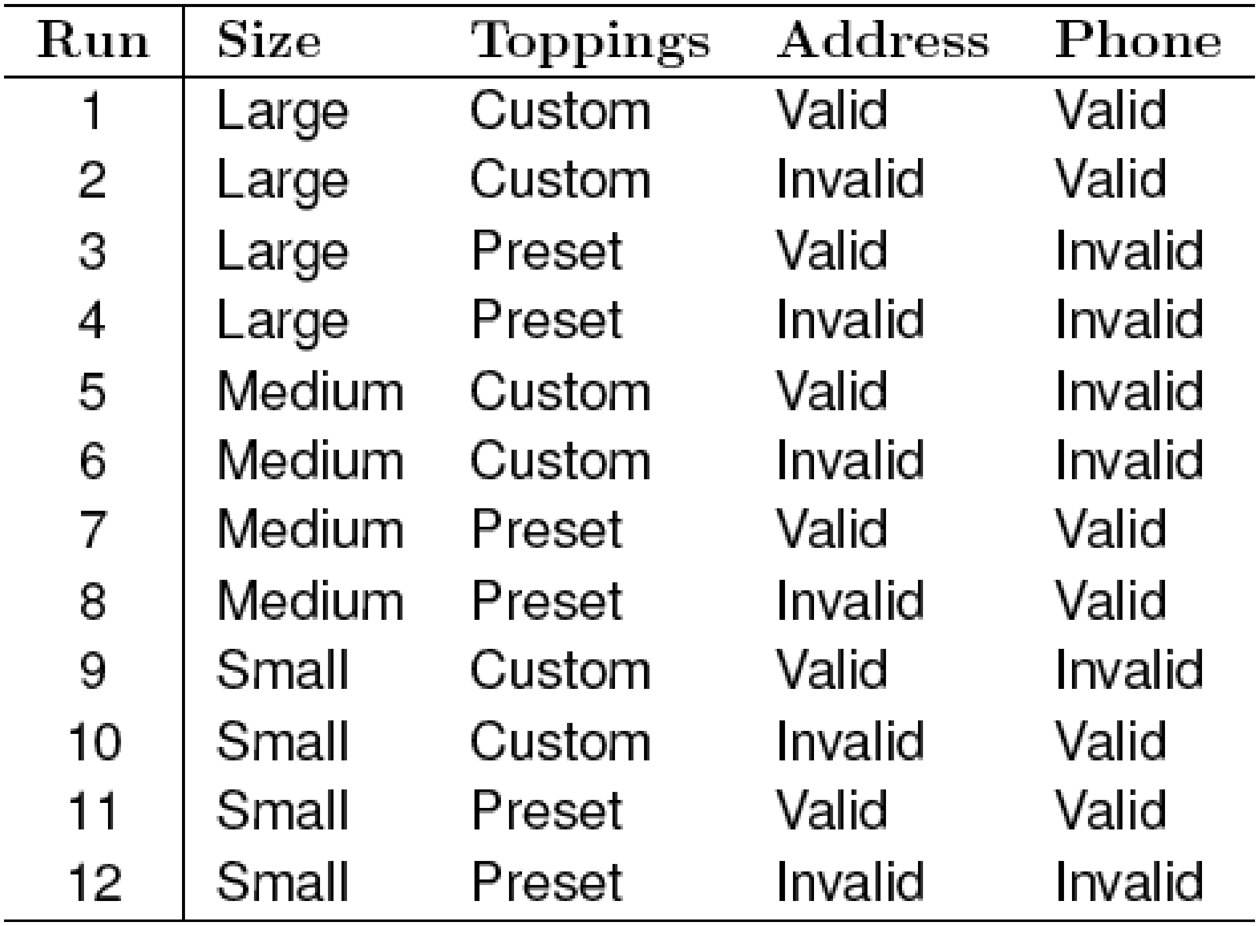
Test generation: Pizza delivery: Array



Run	Size	Toppings	Address	Phone
1	1	1	1	1
2	1	1	2	1
3	1	2	1	2
4	1	2	2	2
5	2	1	1	2
6	2	1	2	2
7	2	2	1	1
8	2	2	2	1
9	3	1	1	2
10	3	1	2	1
11	3	2	1	1
12	3	2	2	2

Check that all possible pairs of factor combinations are covered in the design above. What kind of errors will likely be revealed when testing using these 12 configurations?

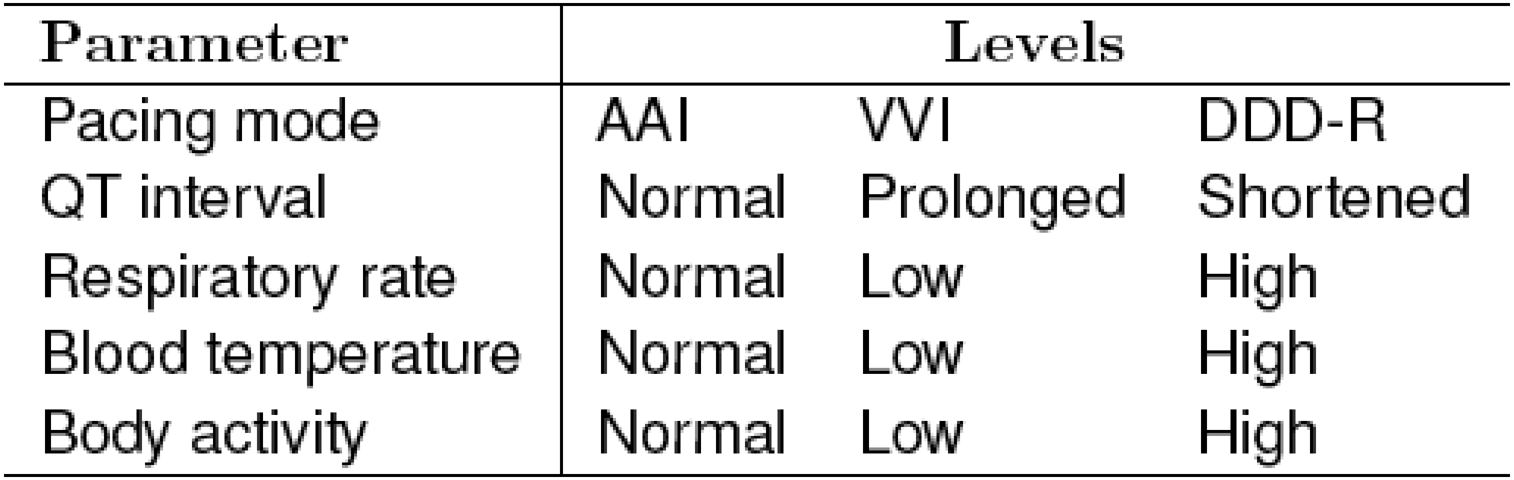
Test generation: Pizza delivery: test configurations



Run	Size	Toppings	Address	Phone
1	Large	Custom	Valid	Valid
2	Large	Custom	Invalid	Valid
3	Large	Preset	Valid	Invalid
4	Large	Preset	Invalid	Invalid
5	Medium	Custom	Valid	Invalid
6	Medium	Custom	Invalid	Invalid
7	Medium	Preset	Valid	Valid
8	Medium	Preset	Invalid	Valid
9	Small	Custom	Valid	Invalid
10	Small	Custom	Invalid	Valid
11	Small	Preset	Valid	Valid
12	Small	Preset	Invalid	Invalid

Arrays of strength >2

Designs with strengths higher than 2 may be needed to achieve higher confidence in the correctness of software. Consider the following factors in a pacemaker



Parameter	Levels		
Pacing mode	AAI	VVI	DDD-R
QT interval	Normal	Prolonged	Shortened
Respiratory rate	Normal	Low	High
Blood temperature	Normal	Low	High
Body activity	Normal	Low	High

Pacemaker example

Due to the high reliability requirement of the pacemaker, we would like to test it to ensure that there are no pairwise or 3-way interaction errors.

Thus we need a suitable combinatorial object with strength 3. We could use an orthogonal array OA(54, 5, 3, 3) that has 54 runs for 5 factors each at 3 levels and is of strength 3. Thus a total of 54 tests will be required to test for all 3-way interactions of the 5 pacemaker parameters

Could a design of strength 2 cover some triples and higher order tuples?

Covering arrays and mixed-level covering arrays

Observation [Dalal and Mallows, 1998]: The balance requirement is often essential in statistical experiments, it is not always so in software testing.

For example, if a software application has been tested once for a given pair of factor levels, there is generally no need for testing it again for the same pair, unless the application is known to behave non-deterministically.

For deterministic applications, and when repeatability is not the focus, we can relax the balance requirement and use covering arrays, or mixed level covering arrays for combinatorial designs.

Statistical approaches versus covering array approach

Statistical approaches estimate parameters of a statistical model to search of fault trigging interactions (pairwise)

Statistical approaches useful when system is subject to significant random error.

Combinatorial test suites based on covering arrays are not balanced and do not use statistical analysis.

Covering array

A covering array CA(N, k, s, t) is an N x k matrix in which entries are from a finite set S of s symbols such that each N x t subarray contains each possible t-tuple at least λ times.

N denotes the number of runs, k the number factors, s, the number of levels for each factor, t the strength, and λ the index

While generating test cases or test configurations for a software application, we use $\lambda=1$. Why?

Covering array and orthogonal array

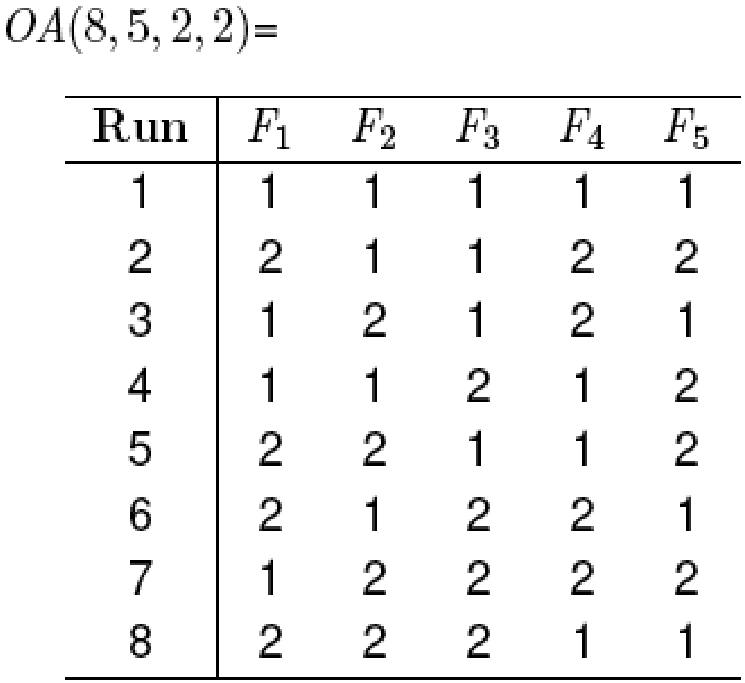
While an orthogonal array OA(N, k, s, t) covers each possible t-tuple λ times in any N x t subarray, a covering array CA(N, k, s, t) covers each possible t-tuple at least λ times in any N x t subarray.

Thus covering arrays do not meet the balance requirement that is met by orthogonal arrays. This difference leads to combinatorial designs that are often smaller in size than orthogonal arrays.

Covering arrays are also referred to as unbalanced designs. We are interested in minimal (size, number of test runs) covering arrays.

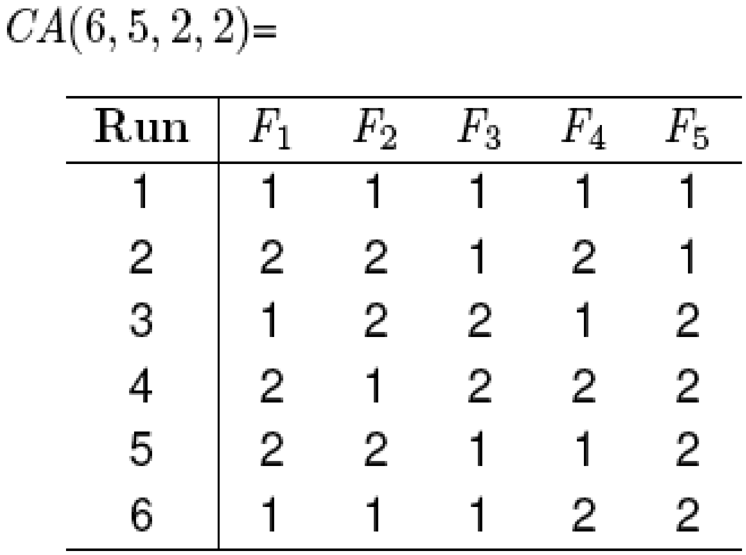
Covering array: Example

A balanced design of strength 2 for 5 binary factors, requires 8 runs and is denoted by OA(8, 5, 2, 2). However, a covering design with the same parameters requires only 6 runs.



$OA(8, 5, 2, 2)$=

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	1	1	2	2
3	1	2	1	2	1
4	1	1	2	1	2
5	2	2	1	1	2
6	2	1	2	2	1
7	1	2	2	2	2
8	2	2	2	1	1



$CA(6, 5, 2, 2)$=

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	2	1	2	1
3	1	2	2	1	2
4	2	1	2	2	2
5	2	2	1	1	2
6	1	1	1	2	2

Mixed level covering arrays

A mixed-level covering array is denoted as

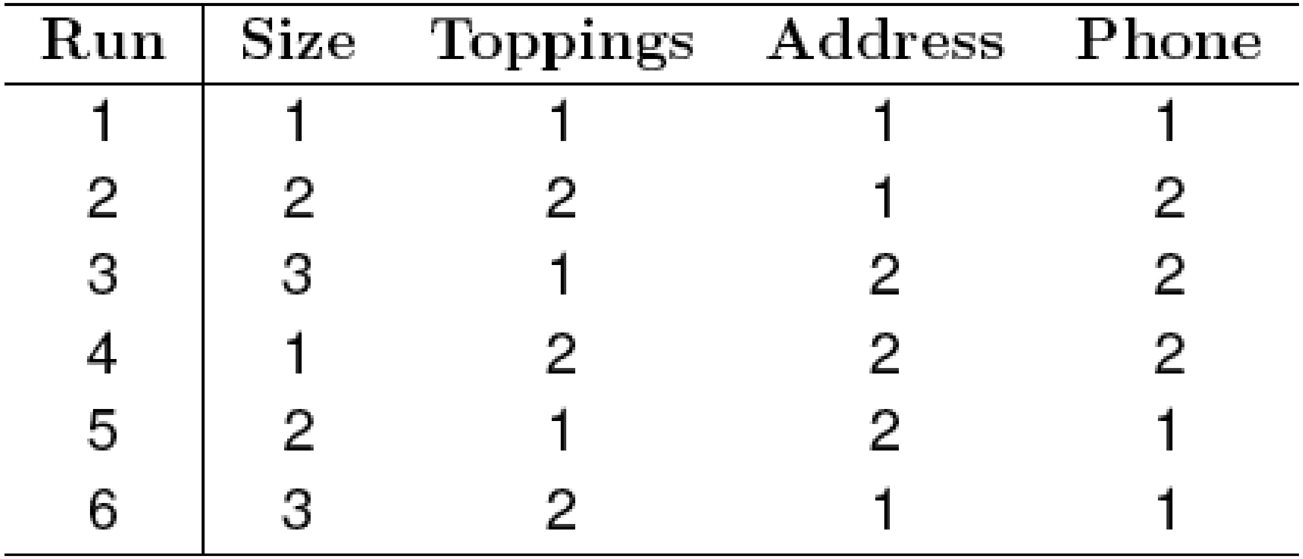
$$MCA(N, s_1^{k_1} s_2^{k_2} \ldots s_p^{k_p}, t)$$

and refers to an N x Q matrix of entries such that, Q= $\sum_{i=1}^{p} k_i$ and each N x t subarray contains at least one occurrence of each t-tuple corresponding to the t columns. s1, s2,,… denote the number of levels of each the corresponding factor.

Mixed-level covering arrays are generally smaller than mixed-level orthogonal arrays and more appropriate for use in software testing.

Mixed level covering array: Example

$MCA(6, 2^3 3^1, 2)$



Run	Size	Toppings	Address	Phone
1	1	1	1	1
2	2	2	1	2
3	3	1	2	2
4	1	2	2	2
5	2	1	2	1
6	3	2	1	1

Comparing this with $MA(12, 2^3 3^1, 2)$we notice a reduction of 6 configurations.

Is the above array balanced?

Generating mixed level covering arrays

We will now study a procedure due to Lei and Tai (Professor Jeff Yu Lei is Faculty Researcher in NIST) for the generation of mixed level covering arrays.

The procedure is known as In-parameter Order (IPO) procedure.

Inputs: (a) $n \geq 2$: Number of parameters (factors). (b) Number of values (levels) for each parameter.

Output: MCA

IPO procedure

Consists of three steps:

Step 1: Main procedure.

Step 2: Horizontal growth.

Step 3: Vertical growth.

IPO procedure: Example

Consider a program with three factors A, B, and C. A assumes values from the set {a1, a2, a3}, B from the set {b1, b2}, and C from the set {c1, c2, c3}. We want to generate a mixed level covering array for these three factors..

We begin by applying the Main procedure which is the first step in the generation of an MCA using the IPO procedure.

IPO procedure: main procedure

Main: Step 1: Construct all runs that consist of pairs of values of the first two parameters. We obtain the following set.

$$\mathcal{T} = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$$

Let us denote the elements of $\mathcal{T}$ as t1, t2,...t6.

The entire IPO procedure would terminate at this point if the number of parameters n=2. In our case n=3 hence we continue with horizontal growth.

IPO Algorithm

Run	a	b	c
t1	1	1	
t2	1	2	
t3	2	1	
t4	2	2	
t5	3	1	
t6	3	2	

IPO procedure: Horizontal growth

HG: Step 1: Compute the set of all pairs AP between parameters A and C, and parameters B and C. This leads us to the following set of fifteen pairs.

$$AP = \{(a_1, c_1), (a_1, c_2), (a_1, c_3), (a_2, c_1), (a_2, c_2), (a_2, c_3), (a_3, c_1), (a_3, c_2), (a_3, c_3)$$

$$(b_1, c_1), (b_1, c_2), (b_1, c_3), (b_2, c_1), (b_2, c_2), (b_2, c_3)\}$$

HG: Step 2: AP is the set of pairs yet to be covered. Let T' denote the set of runs obtained by extending the runs in T. At this point T' is empty as we have not extended any run in T.

Horizontal growth: Extend

HG: Steps 3, 4: Expand t1, t2, t3 by appending c1, c2, c3. This gives us:

t1’=(a1, b1, c1), t2’=(a1, b2, c2), and t3’=(a2, b1, c3)

Update T’ it becomes {(a1, b1, c1), (a1, b2, c2), (a2, b1, c3)}

Update pairs remaining to be covered AP={(a1, c3), (a2, c1), (a2, c2), (a3, c1), (a3, c2), (a3, c3), (b1, c2), (b2, c1), (b2, c3)}

IPO Algorithm

Run	a	b	c
t1’	1	1	1
t2’	1	2	2
t3’	2	1	3
t4	2	2	
t5	3	1	
t6	3	2	

Horizontal growth: Optimal extension

HG. Step 5: We have not extended t4, t5, t6. We find the best way to extend these in the next step.

HG: Step 6: Expand t4, t5, t6 by suitably selected values of C.

If we extend t4=(a2, b2) by c1 then we cover two of the uncovered pairs from AP, namely, (a2, c1) and (b2, c1). If we extend it by c2 then we cover one pair from AP. If we extend it by c3 then we cover one pairs in AP. Thus we choose to extend t4 by c1.

Horizontal growth: Update and extend remaining

T’={(a1, b1, c1), (a1, b2, c2), (a2, b1, c3), (a2, b2, c1)}

AP= {(a1, c3), (a2, c2), (a3, c1), (a3, c2), (a3, c3), (b1, c2), (b2, c3)}

HG: Step 6: Similarly we extend t5 and t6 by the best possible values of parameter C. This leads to:
t5’=(a3, b1, c3) and t6’=(a3, b2, c1)

T’={(a1, b1, c1), (a1, b2, c2), (a2, b1, c3), (a2, b2, c1), (a3, b1, c3), (a3, b2, c1)}
AP= {(a1, c3), (a2, c2), (a3, c2), (b1, c2), (b2, c3)}

IPO Algorithm

Run	a	b	c
t1’	1	1	1
t2’	1	2	2
t3’	2	1	3
t4	2	2	1
t5	3	1	3
t6	3	2	1

Horizontal growth: Done

We have completed the horizontal growth step. However, we have five pairs remaining to be covered. These are:
AP= {(a1, c3), (a2, c2), (a3, c2), (b1, c2), (b2, c3)}

Also, we have generated six complete runs namely:

T’={(a1, b1, c1), (a1, b2, c2), (a2, b1, c3), (a2, b2, c1), (a3, b1, c3), (a3, b2, c1)}

We now move to the vertical growth step of the main IPO procedure to cover the remaining pairs.

Vertical growth

For each missing pair p from AP, we will add a new run to T' such that p is covered. Let us begin with the pair p= (a1, c3).

The run t = (a1, *, c3) covers pair p. Note that the value of parameter Y does not matter and hence is indicated as a * which denotes a don't care value.

Next , consider p=(a2, c2). This is covered by the run (a2, *, c2)

Next , consider p=(a3, c2). This is covered by the run (a3, *, c2)

Vertical growth (contd.)

Next , consider p=(b2, c3). We already have (a1, *, c3) and hence we can modify it to get the run (a1, b2, c3). Thus p is covered without any new run added.

Finally, consider p=(b1, c2). We already have (a3, *, c2) and hence we can modify it to get the run (a3, b1, c2). Thus p is covered without any new run added.

Final covering array: MCA (9, 2^1x3^2,2)

Run	F1(X)	F2(Y)	F3(Z)
t1	1	1	1
t2	1	2	2
t3	2	1	3
t4	2	2	1
t5	3	1	3
t6	3	2	1
t7	1	2	3
t8	2	* (1)	2
t9	3	1	2

ACTS Tool

ACTS (NIST/UTA) tool freely available from NIST can generate test suites developed from generalization of IPO algorithm for generating covering arrays of any desired strength of t = 6 or more

-Small size test suites and efficient generation

Tutorial Overview

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

Pairwise testing is popular, but is it enough?

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

Finding 90% of flaws is pretty good, right?

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

NIST
National Institute of Standards and Technology

Software Failure Analysis

- We studied software failures in a variety of fields including 15 years of FDA medical device recall data
- What causes software failures?
  - logic errors?
  - calculation errors?
  - interaction faults?
  - inadequate input checking? Etc.
- What testing and analysis would have prevented failures?
- Would statement coverage, branch coverage, all-values, all-pairs etc. testing find the errors?

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

Software Failure Internals

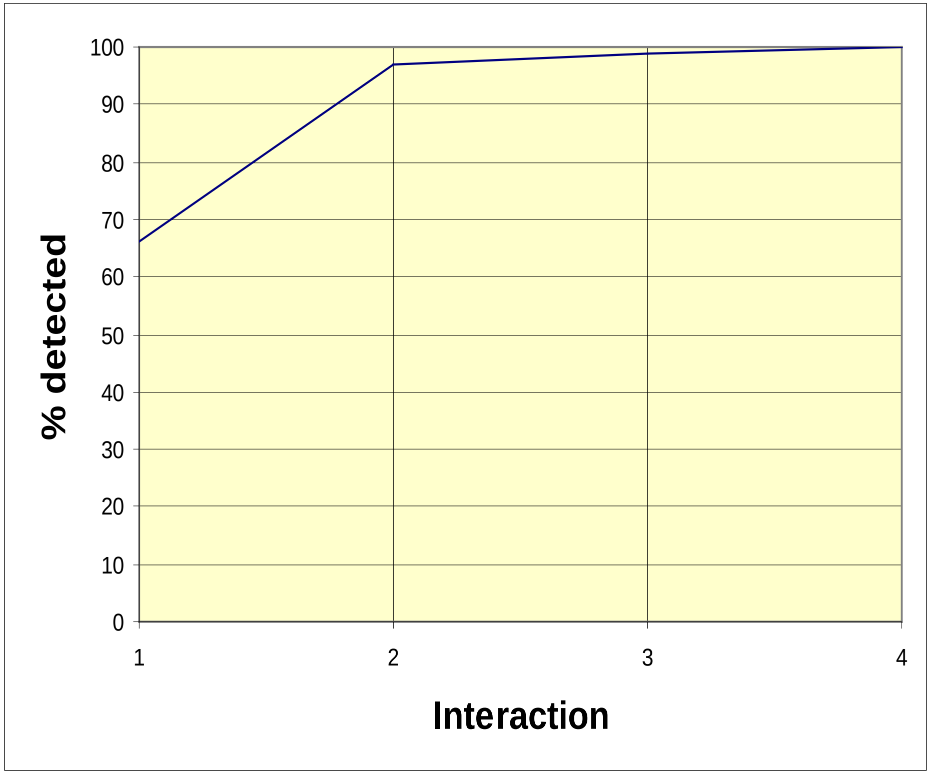
- How does an interaction fault manifest itself in code?

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

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

How about hard-to-find flaws?

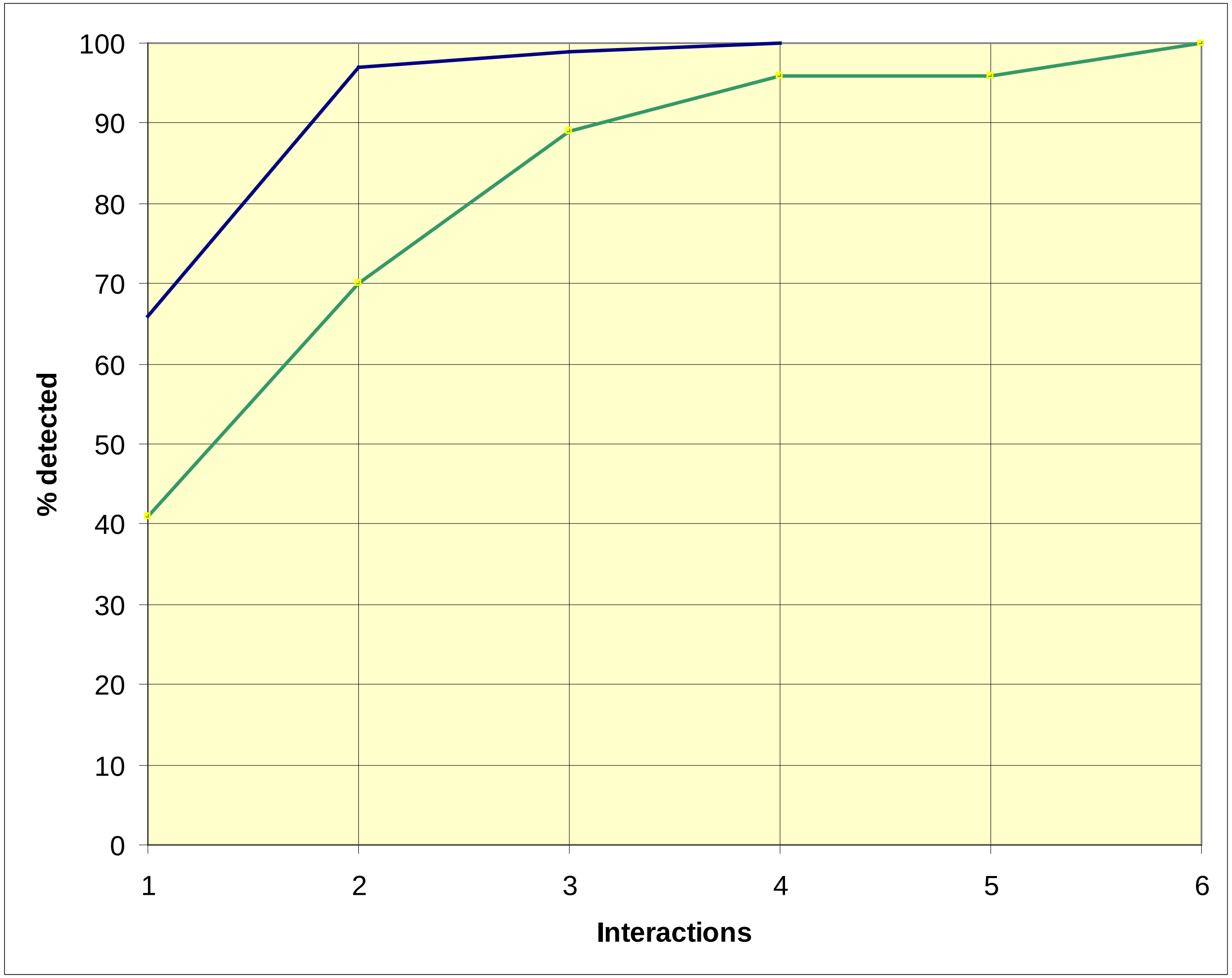
- Interactions e.g., failure occurs if
- pressure < 10 (1-way interaction)
- pressure < 10 & volume > 300 (2-way interaction)
- pressure < 10 & volume > 300 & velocity = 5 (3-way interaction)
- The most complex failure reported required 4-way interaction to trigger



Interesting, but that's just one kind of application.

How about other applications?

Browser (green)

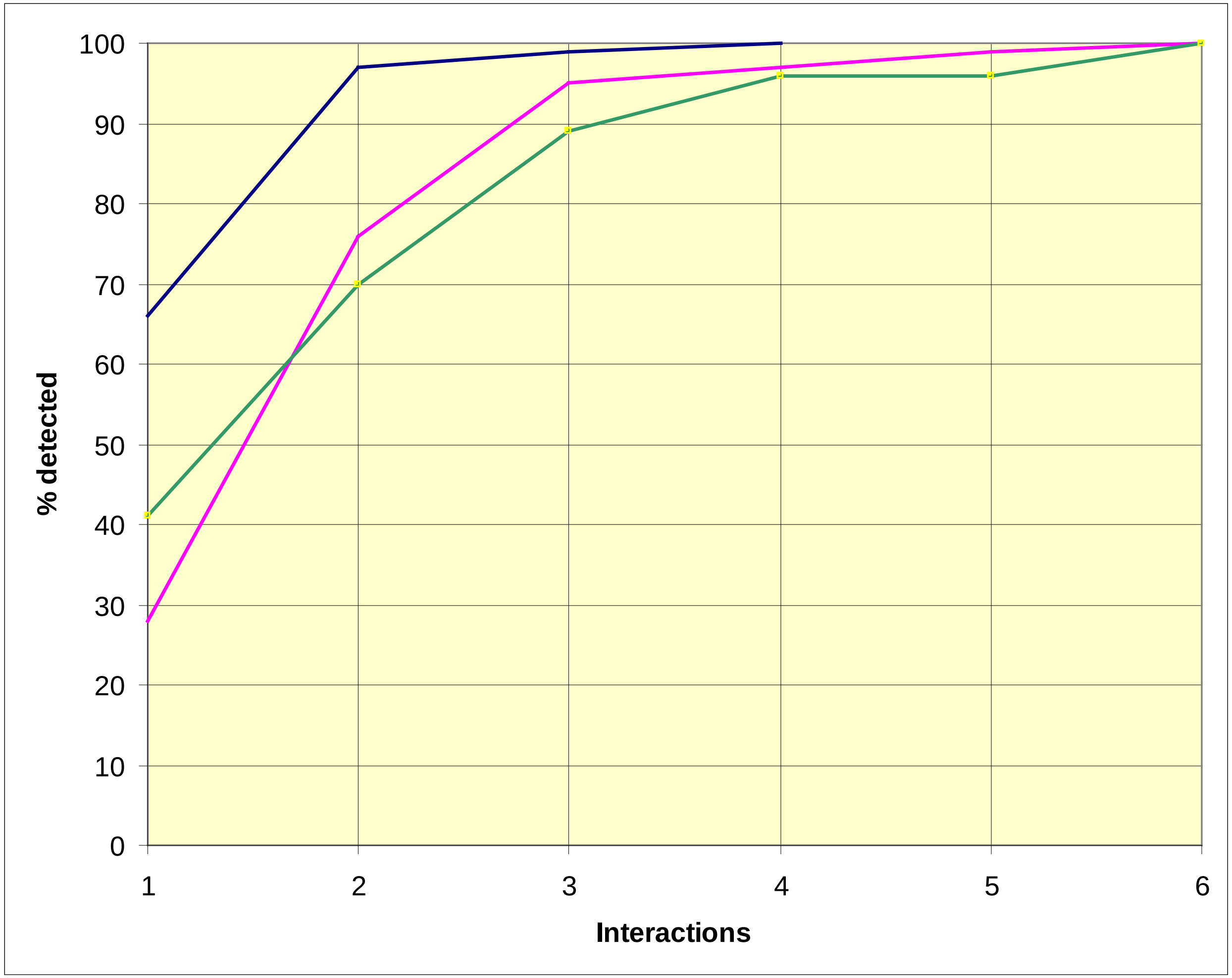


These faults more complex than medical device software!!

Why?

And other applications?

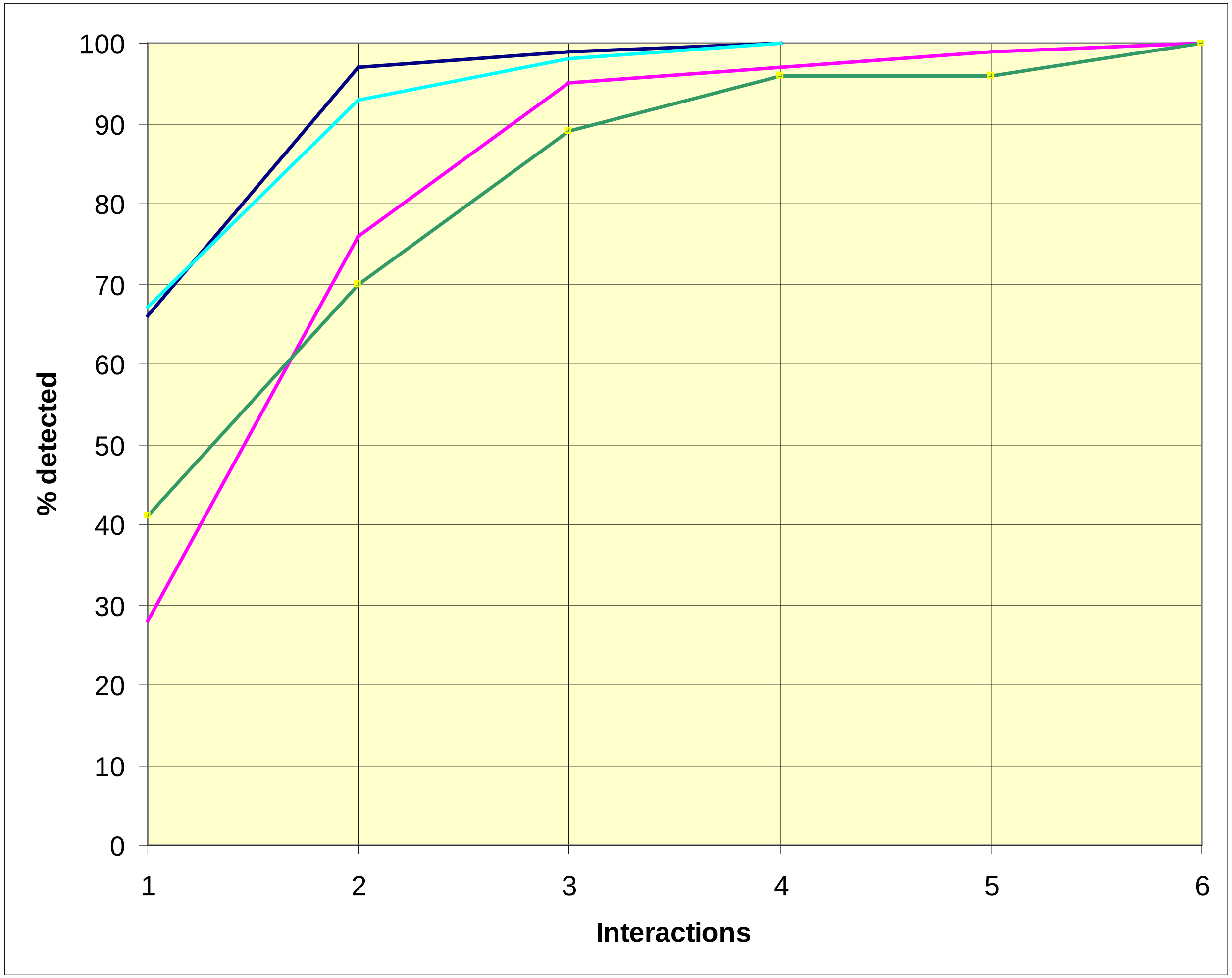
Server (magenta)



Still more?

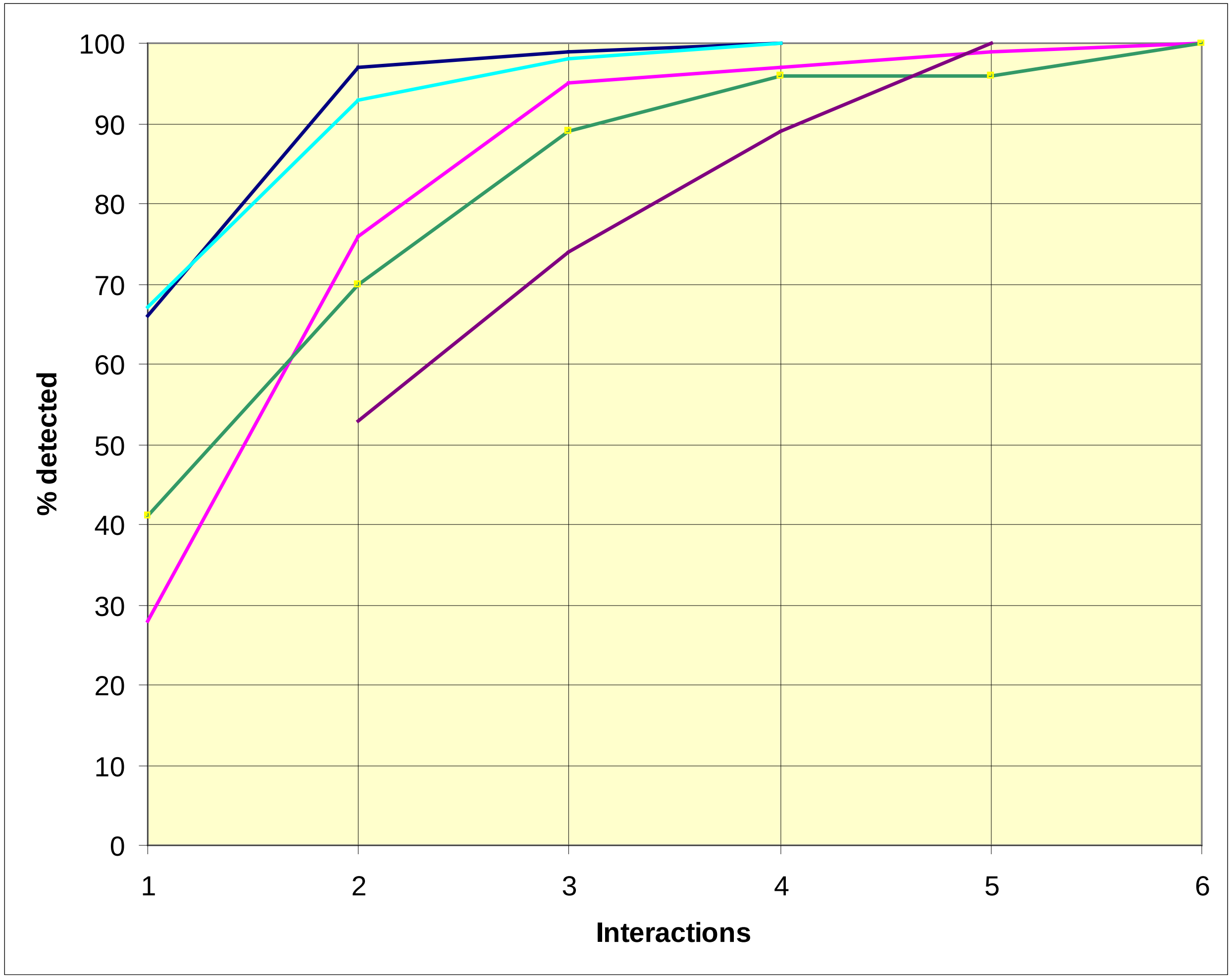
NASA distributed database

(light blue)



Even more?

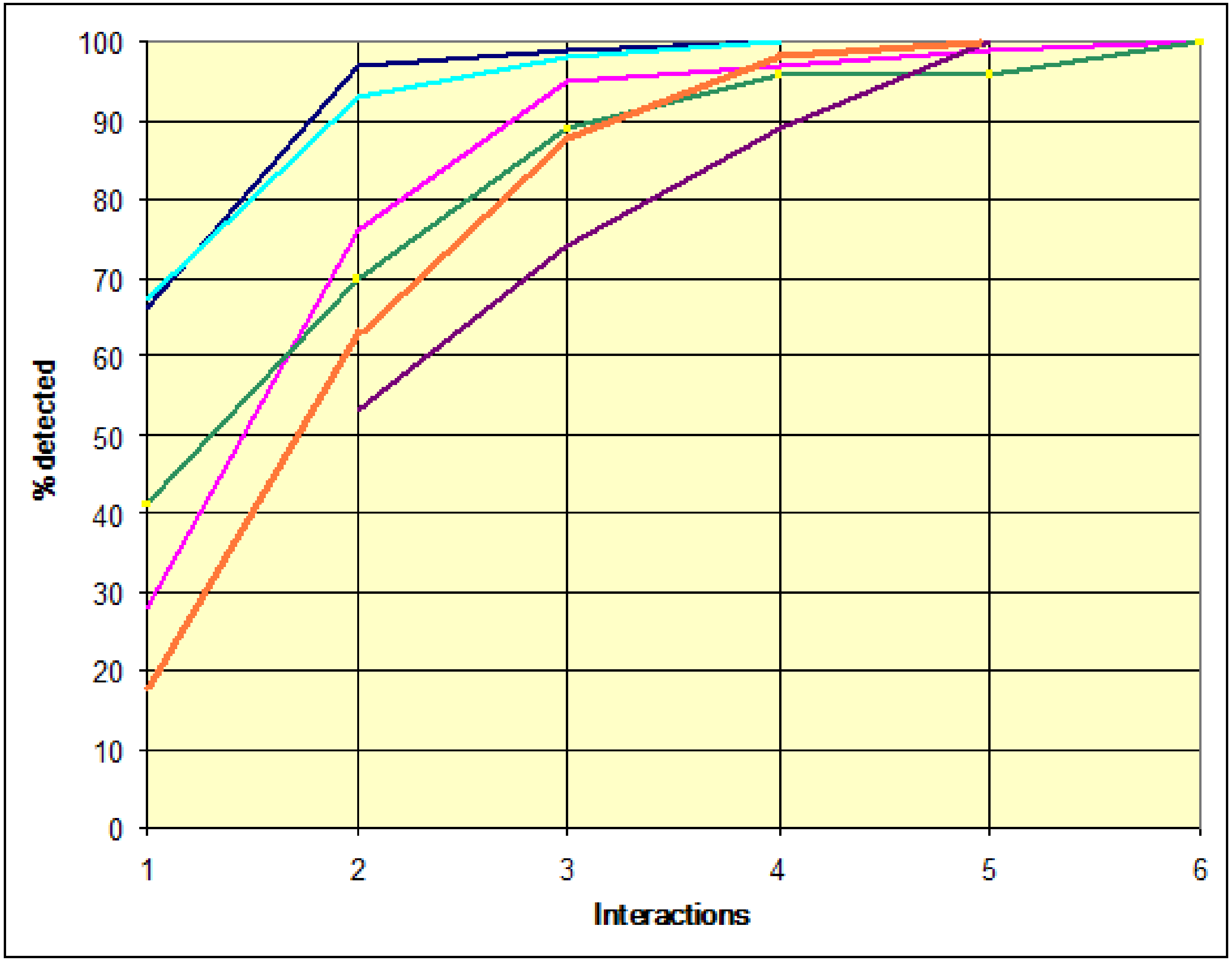
Traffic Collision Avoidance System module (seeded errors) (purple)



Finally

Network security (Bell, 2006)

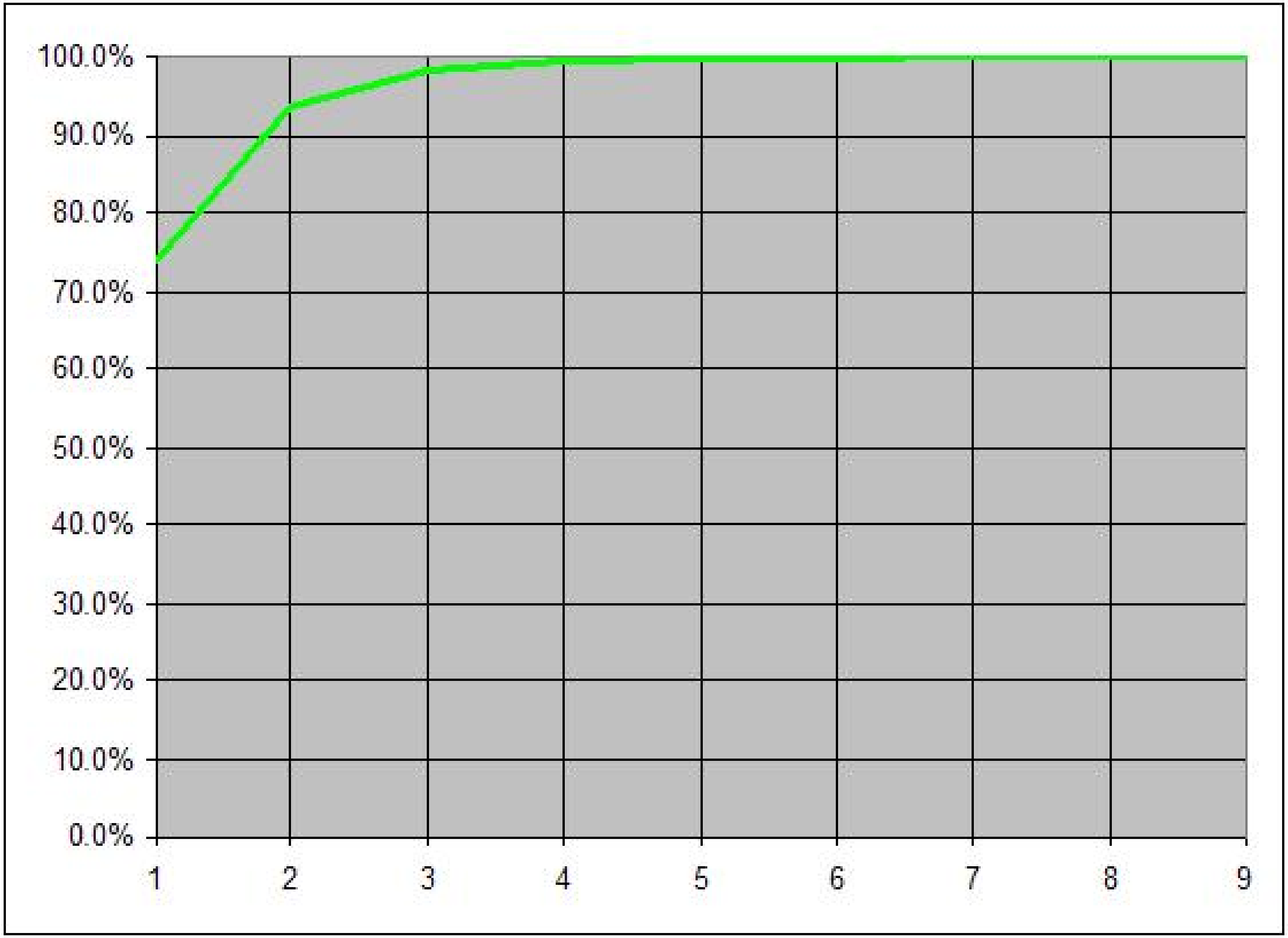
(orange)



Curves appear to be <u>similar</u> across a variety of application domains.

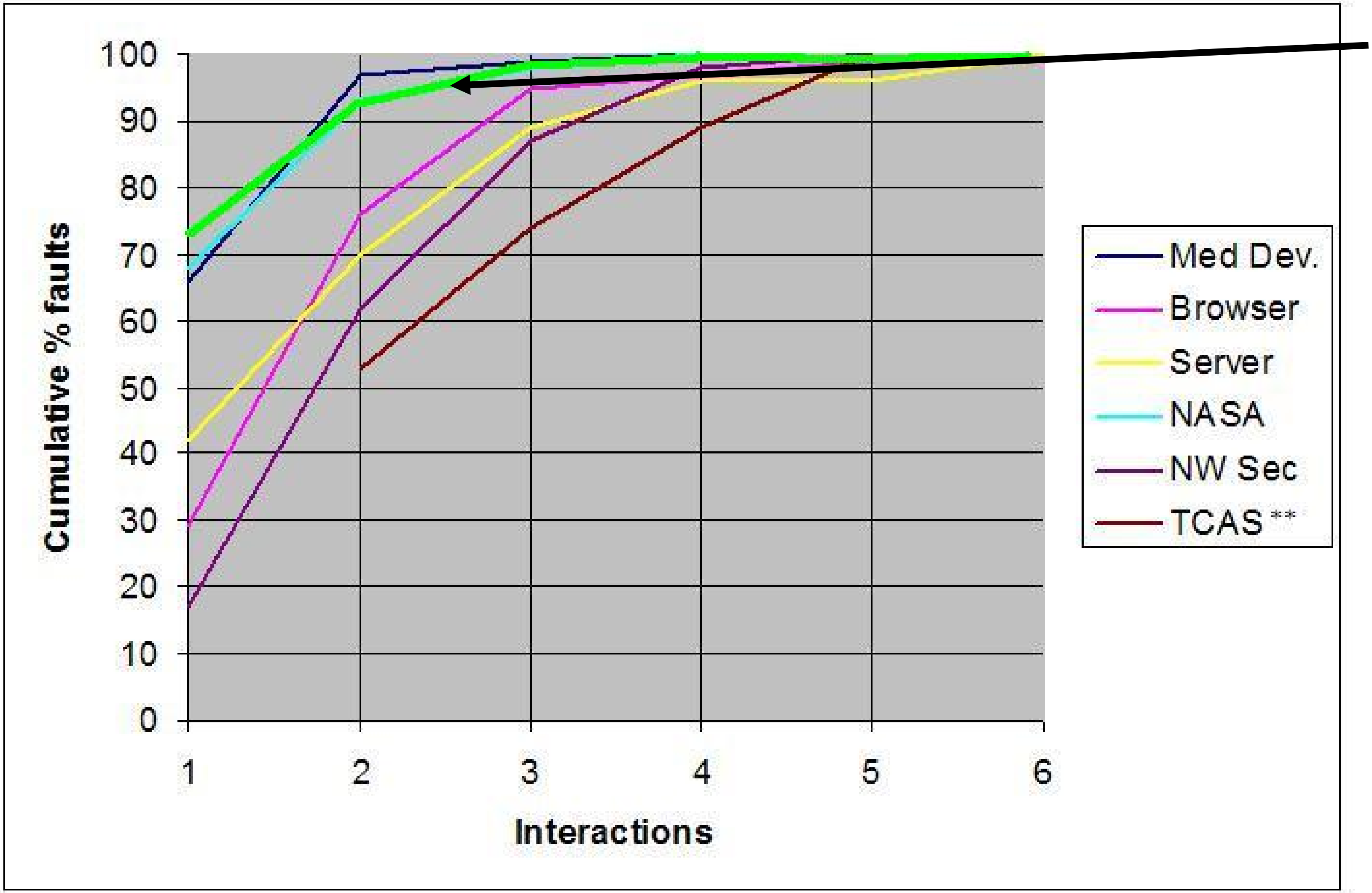
Why this distribution?

What causes this distribution?



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

Comparing with Failure Data



Branch statements

So, how many parameters are involved in really tricky faults?

- Maximum interactions for fault triggering for these applications was 6
- Much more empirical work needed
- Reasonable evidence that maximum interaction strength for fault triggering is relatively small

How does it help me to know this?

How does this knowledge help?

Biologists have a “Central Dogma”, and so do we:

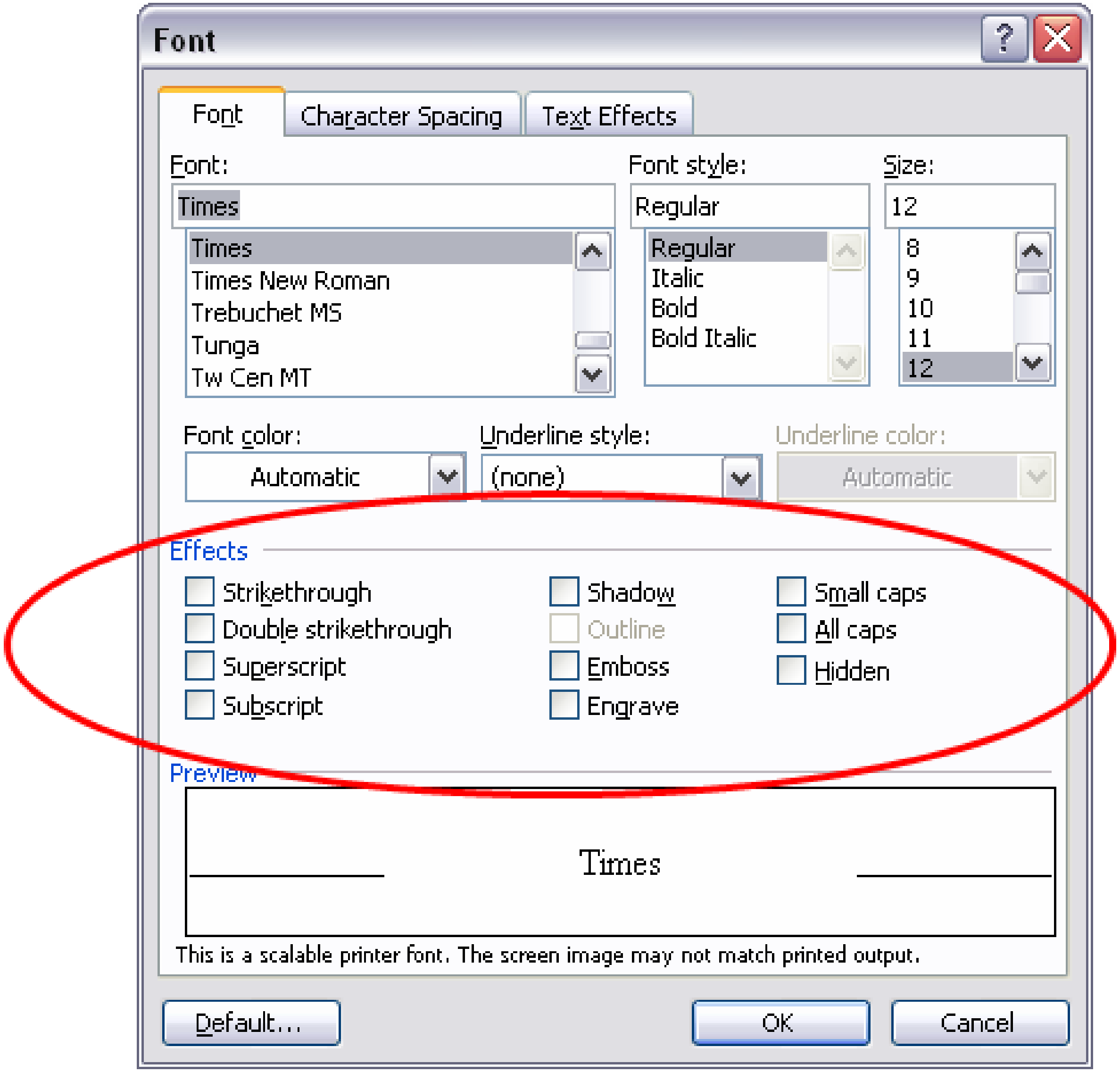
If all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations can provide strong assurance.

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

Tutorial Overview

1. What is combinatorial testing?
2. Why are we doing this?
3. **How is it used and how long does it take?**
   - scaling up -> real-world examples
   - different application domains
4. What tools are available?
5. What's next?

A simple example



Font

Font | Character Spacing | Text Effects

Font: Times

Times
Times New Roman
Trebuchet MS
Tunga
Tw Cen MT

Font style: Regular

Regular
Italic
Bold
Bold Italic

Size: 12

8
9
10
11
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.

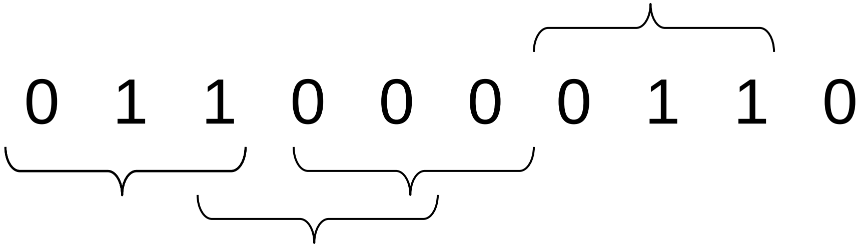
Default... | OK | Cancel

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:


0 1 1 0 0 0 0 1 1 0

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

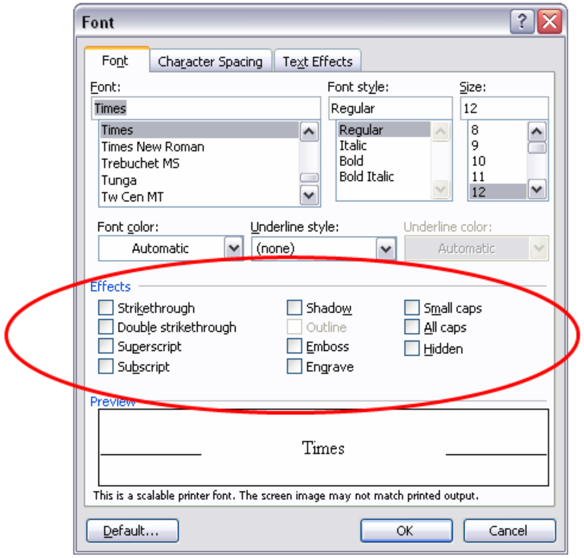
A covering array

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

Each row is a test:

Each column is a parameter:

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


Font

Font | Character Spacing | Text Effects

Font: Times

Times
Times New Roman
Trebuchet MS
Tunga
Tw Cen MT

Font style: Regular

Regular
Italic
Bold
Bold Italic

Size: 12

8
9
10
11
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

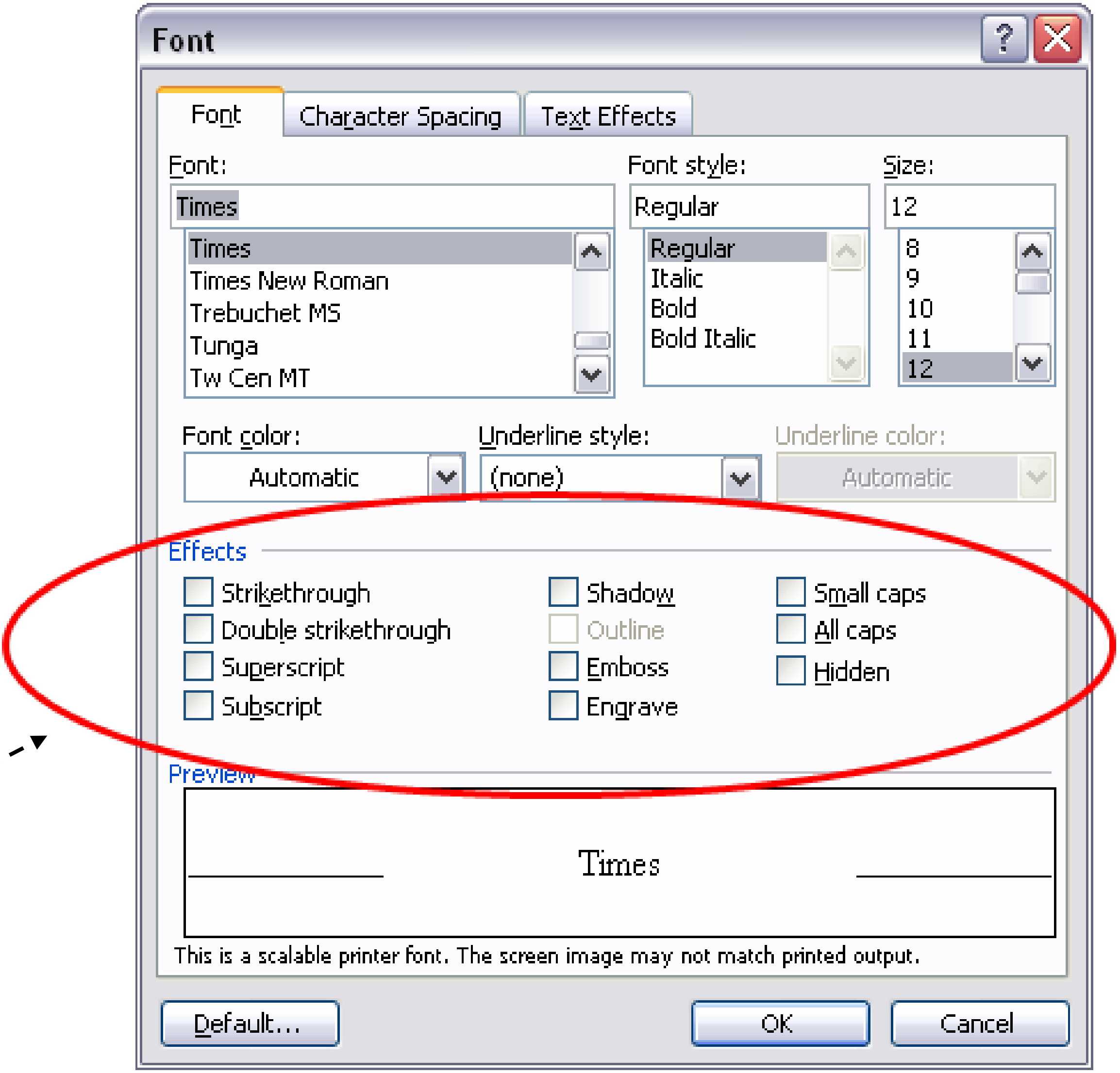
This is a scalable printer font. The screen image may not match printed output.

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



Font

Font | Character Spacing | Text Effects

Font: Times
- Times
- Times New Roman
- Trebuchet MS
- Tunga
- Tw Cen MT

Font style: Regular
- Regular
- Italic
- Bold
- Bold Italic

Size: 12
- 8
- 9
- 10
- 11
- 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.

0 = effect off
1 = effect on

13 tests for **all 3-way combinations**

2^{10} = **1,024** tests for **all combinations**

Testing Configurations - Example

- Example: Android smart phone testing using emulator (project for DARPA)

- Apps should work on all combinations of platform options,
  but there are 3 x 3 x 4 x 3 x 5 x 4 x 4 x 5 x 4 = 172,800 configurations

HARDKEYBOARDHIDDEN_NO
HARDKEYBOARDHIDDEN_UNDEFINED
HARDKEYBOARDHIDDEN_YES

KEYBOARDHIDDEN_NO
KEYBOARDHIDDEN_UNDEFINED
KEYBOARDHIDDEN_YES

KEYBOARD_12KEY
KEYBOARD_NOKEYS
KEYBOARD_QWERTY
KEYBOARD_UNDEFINED

NAVIGATIONHIDDEN_NO
NAVIGATIONHIDDEN_UNDEFINED
NAVIGATIONHIDDEN_YES

NAVIGATION_DPAD
NAVIGATION_NONAV
NAVIGATION_TRACKBALL
NAVIGATION_UNDEFINED
NAVIGATION_WHEEL

ORIENTATION_LANDSCAPE
ORIENTATION_PORTRAIT
ORIENTATION_SQUARE
ORIENTATION_UNDEFINED

SCREENLAYOUT_LONG_MASK
SCREENLAYOUT_LONG_NO
SCREENLAYOUT_LONG_UNDEFINED
SCREENLAYOUT_LONG_YES

SCREENLAYOUT_SIZE_LARGE
SCREENLAYOUT_SIZE_MASK
SCREENLAYOUT_SIZE_NORMAL
SCREENLAYOUT_SIZE_SMALL
SCREENLAYOUT_SIZE_UNDEFINED

TOUCHSCREEN_FINGER
TOUCHSCREEN_NOTOUCH
TOUCHSCREEN_STYLUS
TOUCHSCREEN_UNDEFINED

NIST
National Institute of
Standards and Technology

Testing Android Combinatorially

- 3 x 3 x 4 x 3 x 5 x 4 x 4 x 5 x 4 = 172,800 configurations
- Effort substantially reduced with t-way combinations:

t	Number tests	Pct of all configs
2	34	0.02
3	139	0.08
4	634	0.4
5	2783	1.6
6	10762	6.2

A larger example

- Suppose we have a system with 34 on-off switches:

How do we test this?

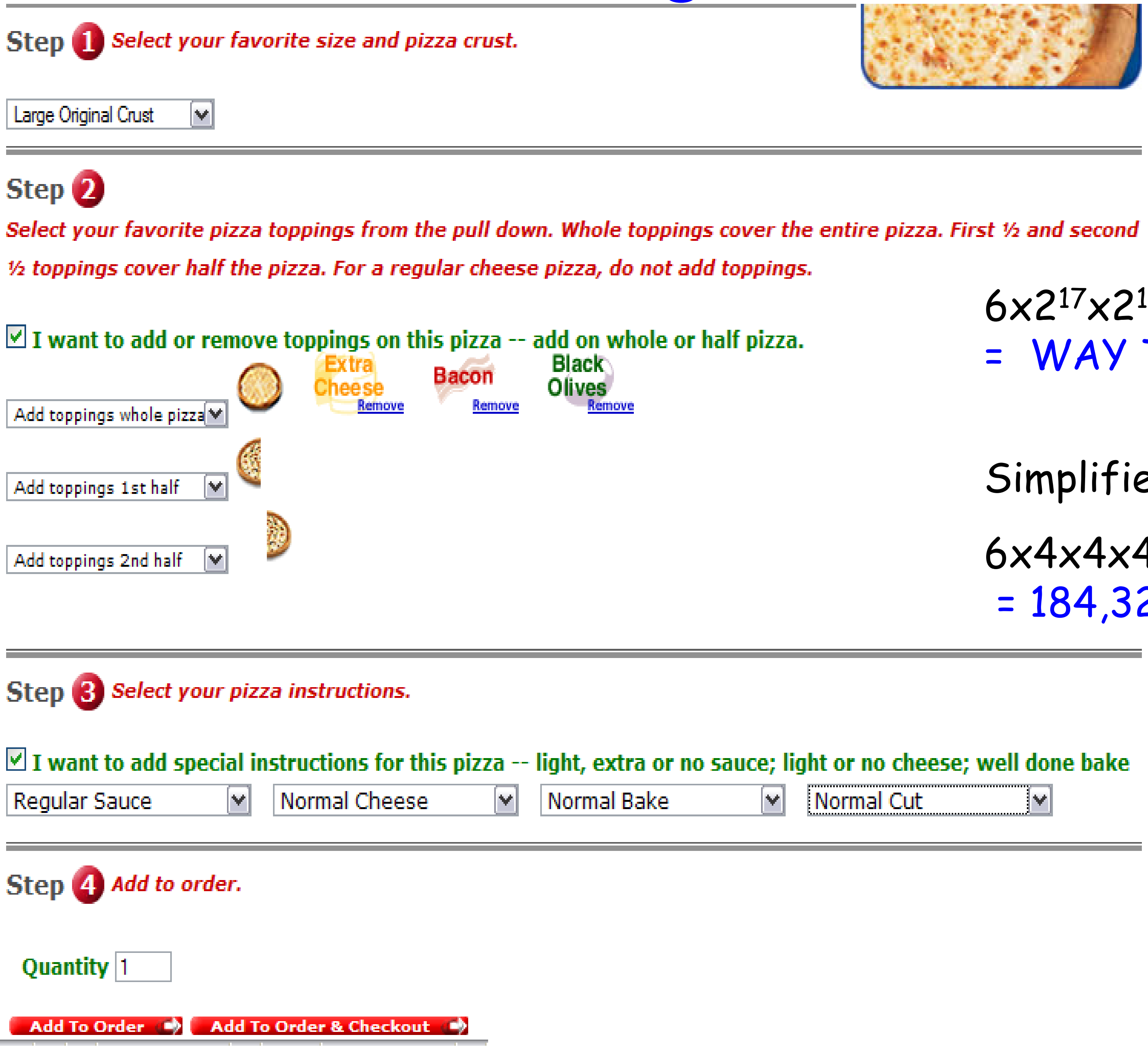
- 34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = 1.7 x 10^{10} tests

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

Ordering Pizza

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.

Add toppings whole pizza

Add toppings 1st half

Add toppings 2nd half

Step 3 *Select your pizza instructions.*

☑ I want to add special instructions for this pizza -- light, extra or no sauce; light or no cheese; well done bake

Regular Sauce | Normal Cheese | Normal Bake | Normal Cut

Step 4 *Add to order.*

Quantity 1

$6x2^{17}x2^{17}x2^{17}x4x3x2x2x5x2$

= WAY TOO MUCH TO TEST

Simplified pizza ordering:

$6x4x4x4x4x3x2x2x5x2$

= 184,320 possibilities

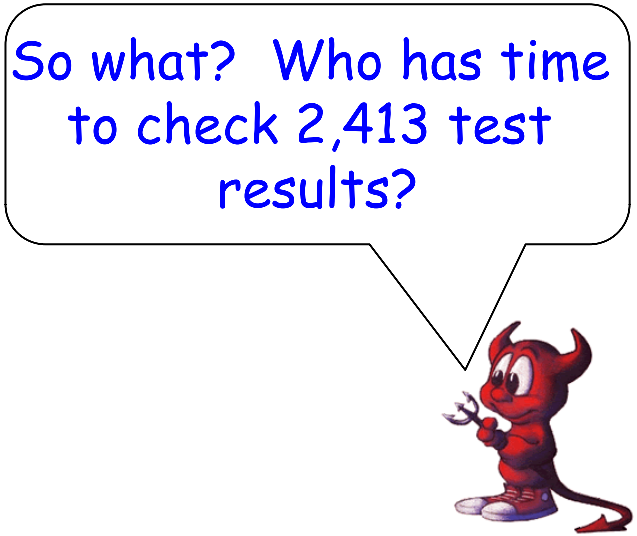
Ordering Pizza Combinatorially

Simplified pizza ordering:

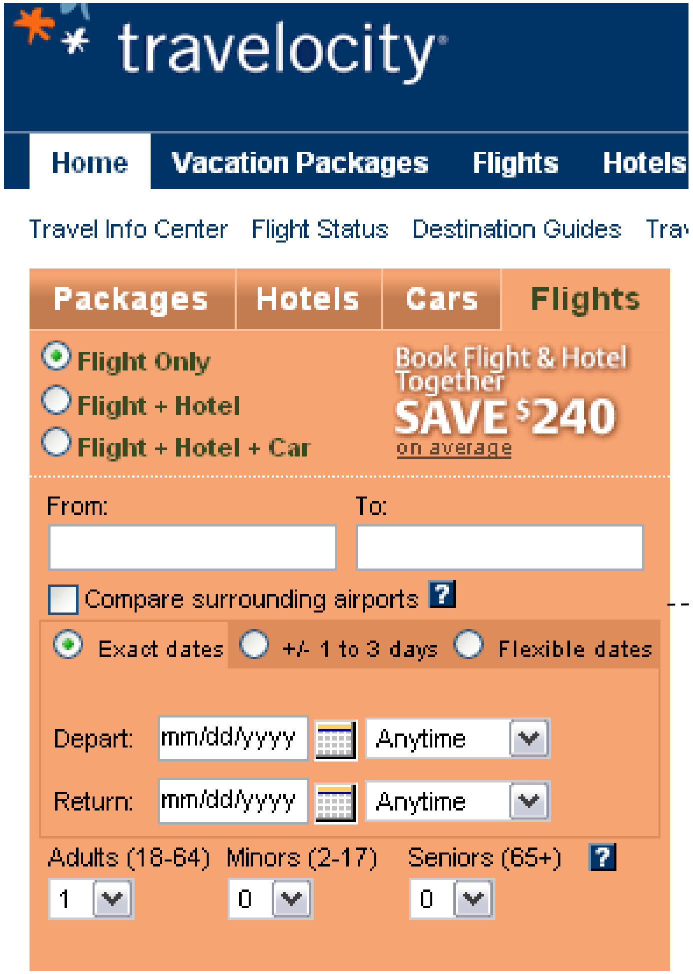
6x4x4x4x4x3x2x2x5x2
= 184,320 possibilities

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.



Another familiar example



travelocity

Home | Vacation Packages | Flights | Hotels

Travel Info Center | Flight Status | Destination Guides | Tra

Packages | Hotels | Cars | Flights

● Flight Only
○ Flight + Hotel
○ Flight + Hotel + Car

Book Flight & Hotel Together
SAVE $240
on average

From: To:

☐ Compare surrounding airports ?

● Exact dates ○ +/- 1 to 3 days ○ Flexible dates

Depart: mm/dd/yyyy Anytime

Return: mm/dd/yyyy Anytime

Adults (18-64) 1 | Minors (2-17) 0 | Seniors (65+) 0 ?

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

Two ways of using combinatorial testing: (1) test inputs (2) configurations

Use combinations here

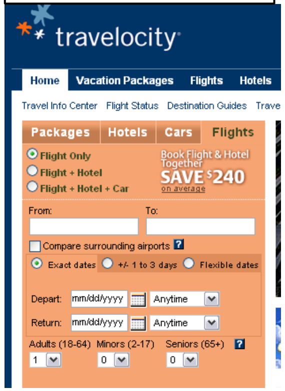
or here

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



● Flight Only
○ Flight + Hotel
○ Flight + Hotel + Car

From:
To:

□ Compare surrounding airports ?

● Exact dates ○ +/- 1 to 3 days ○ Flexible dates

Depart: mm/dd/yyyy Anytime
Return: mm/dd/yyyy Anytime

Adults (18-64) 1 Minors (2-17) 0 Seniors (65+) 0 ?

Difference with conventional practice

Conventional

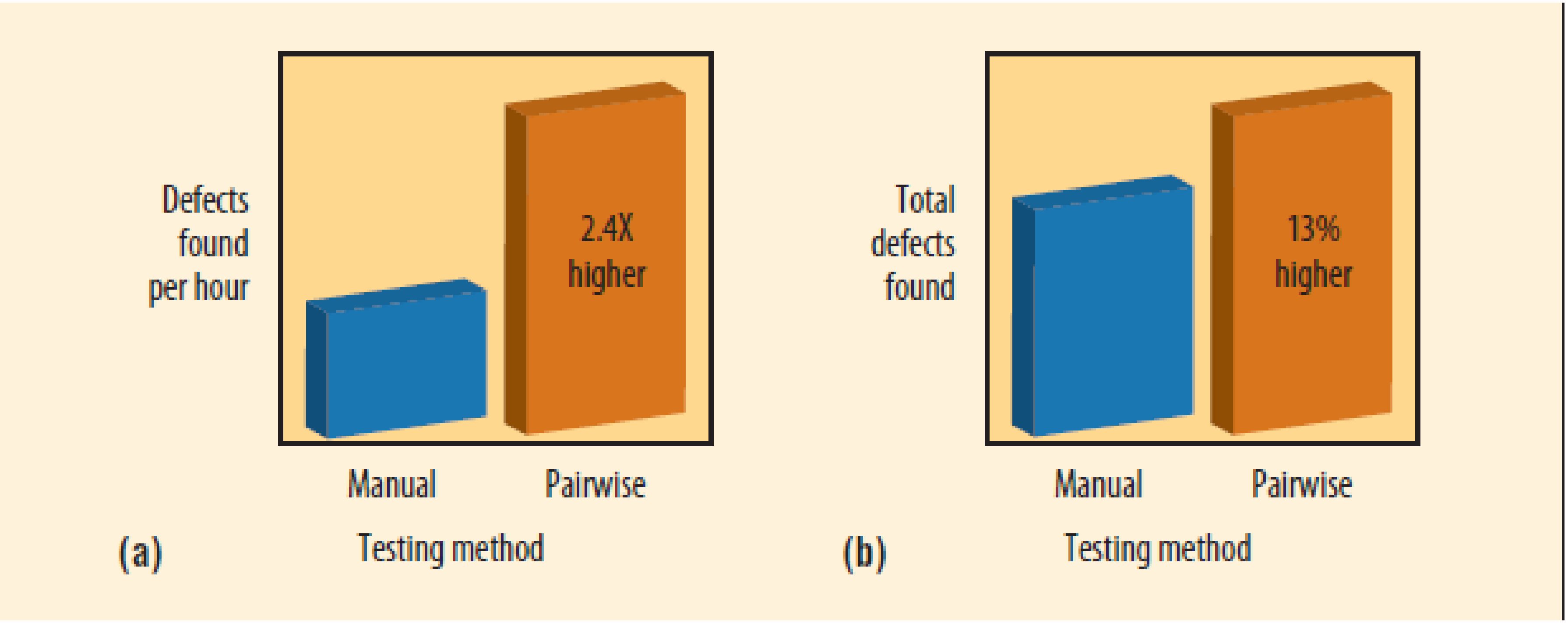
- Use cases
  - typical cases
  - outliers
- Abstraction and equivalence classes

Combinatorial

- Use cases
  - t-way combinations
- Abstraction and equivalence classes

Experimental comparison with conventional practice

- Real-world experiment by Justin Hunter
- 10 projects, 6 companies
- 2-way only, no higher t-way combinations



Testing efficiency

Testing quality

National Institute of Standards and Technology

Example 1

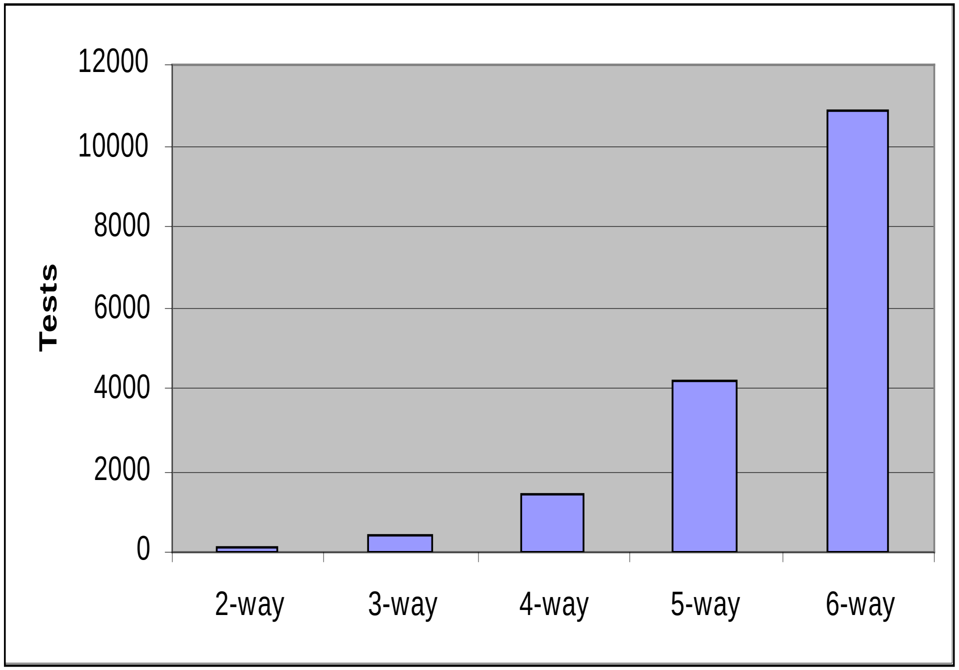
Traffic Collision Avoidance System (TCAS) module

- 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

  - test inputs generated by ACTS
  - results generated by model checker
  - full testing on each version complete in a few minutes

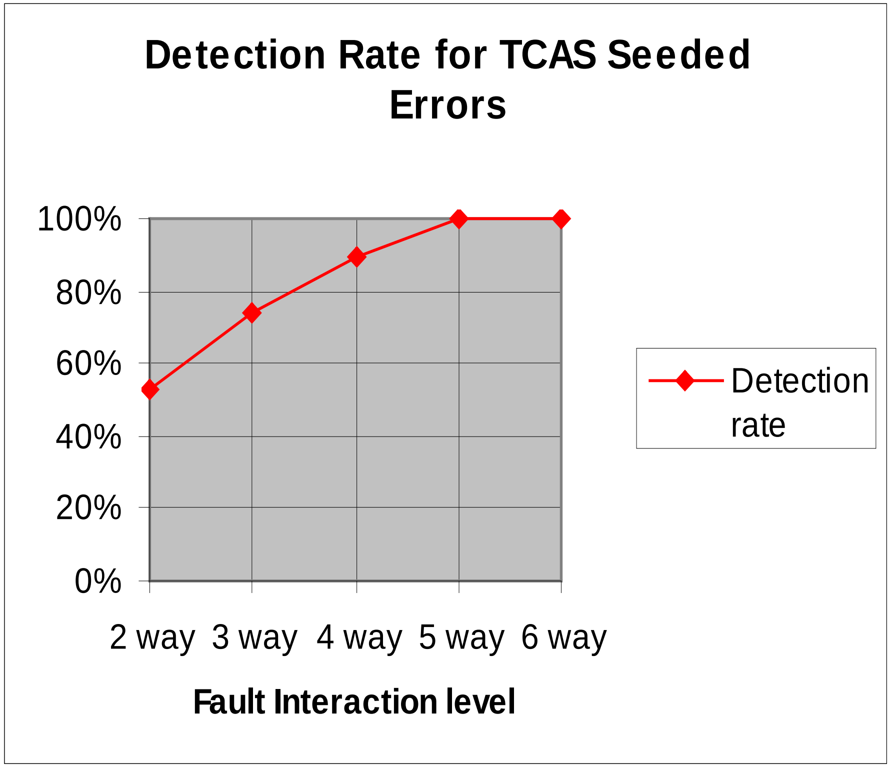
Tests generated

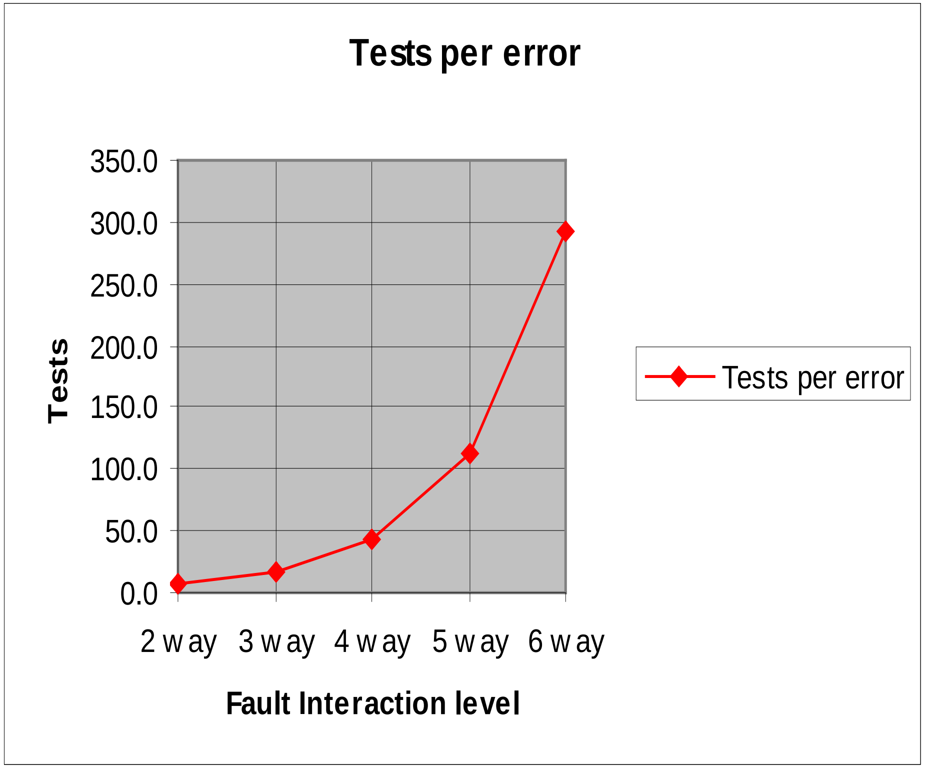
t	Test cases
2-way:	156
3-way:	461
4-way:	1,450
5-way:	4,309
6-way:	11,094



Results

- Roughly consistent with real-world data on large systems
- But errors harder to detect than real-world examples

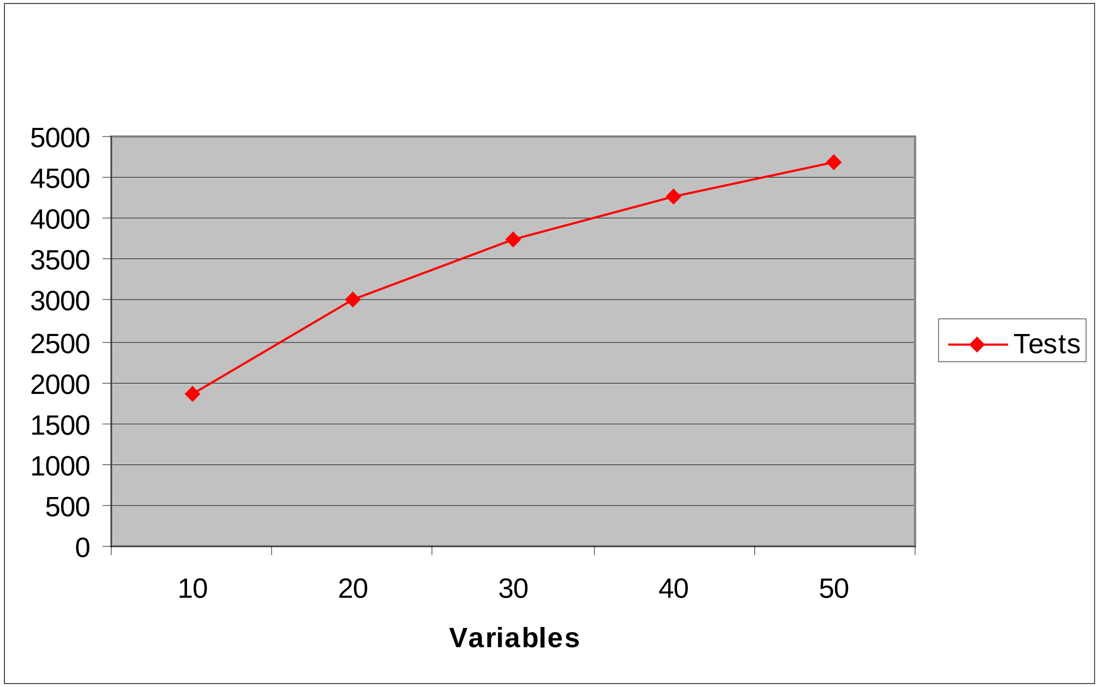




Cost and Volume of Tests

- Number of tests: proportional to $\underline{v^t \log n}$

  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:



Example 2:
Modeling & Simulation Application

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

Simulation System Configurations

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

5x3x4x4x4x4x2x2x2x4x4x4x4x4 = 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 <u>might never have been found</u> 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)

Example 3: Buffer Overflows

Sponsored by
DHS National Cyber Security Division/US-CERT
NIST
National Institute of Standards and Technology
National Vulnerability Database
automating vulnerability management, security measurement, and compliance checking

- 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

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
       if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
       ……
conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024, sizeof(char));
             ……
      pPostData=conn[sid].PostData;
      do {
              rc=recv(conn[sid].socket, pPostData, 1024, 0);
                ……
              pPostData+=rc;
              x+=rc;
      } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
 }
```

Interaction: request-method=”POST”, content-length = -1000, data= a string > 24 bytes

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
       if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
       ……
conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024,
sizeof(char));
             ……
      pPostData=conn[sid].PostData;
      do {
              rc=recv(conn[sid].socket, pPostData, 1024, 0);
                ……
              pPostData+=rc;
              x+=rc;
      } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
 }
```

Interaction: request-method=”POST”, content-length = -1000, data= a string > 24 bytes

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
       if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
       ……
conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024,
sizeof(char));
             ……
      pPostData=conn[sid].PostData;
      do {
              rc=recv(conn[sid].socket, pPostData, 1024, 0);
                ……
              pPostData+=rc;
              x+=rc;
      } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
 }
```

true branch

Interaction: request-method=”POST”, content-length = -1000, data= a string > 24 bytes

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
       if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
       ……
       conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024, sizeof(char));
             ……
      pPostData=conn[sid].PostData;
      do {
              rc=recv(conn[sid].socket, pPostData, 1024, 0);
                ……
              pPostData+=rc;
              x+=rc;
      } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
 }
```

true branch

Interaction: request-method=”POST”, content-length = -1000, data= a string > 24 bytes

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
       if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
```

true branch

```
       ……
       conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024, sizeof(char));
             ……
```

Allocate -1000 + 1024 bytes = 24 bytes

```
      pPostData=conn[sid].PostData;
      do {
              rc=recv(conn[sid].socket, pPostData, 1024, 0);
                ……
              pPostData+=rc;
              x+=rc;
      } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
 }
```

Interaction: request-method=”POST”, content-length = -1000, data= a string > 24 bytes

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
        if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
        ……
        conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024,
sizeof(char));
              ……
        pPostData=conn[sid].PostData;
        do {
                rc=recv(conn[sid].socket, pPostData, 1024, 0)
                  ……
                pPostData+=rc;
                x+=rc;
        } while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
 }
```

true branch

Allocate -1000 + 1024 bytes = 24 bytes

Boom!

How to automate checking correctness of output

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

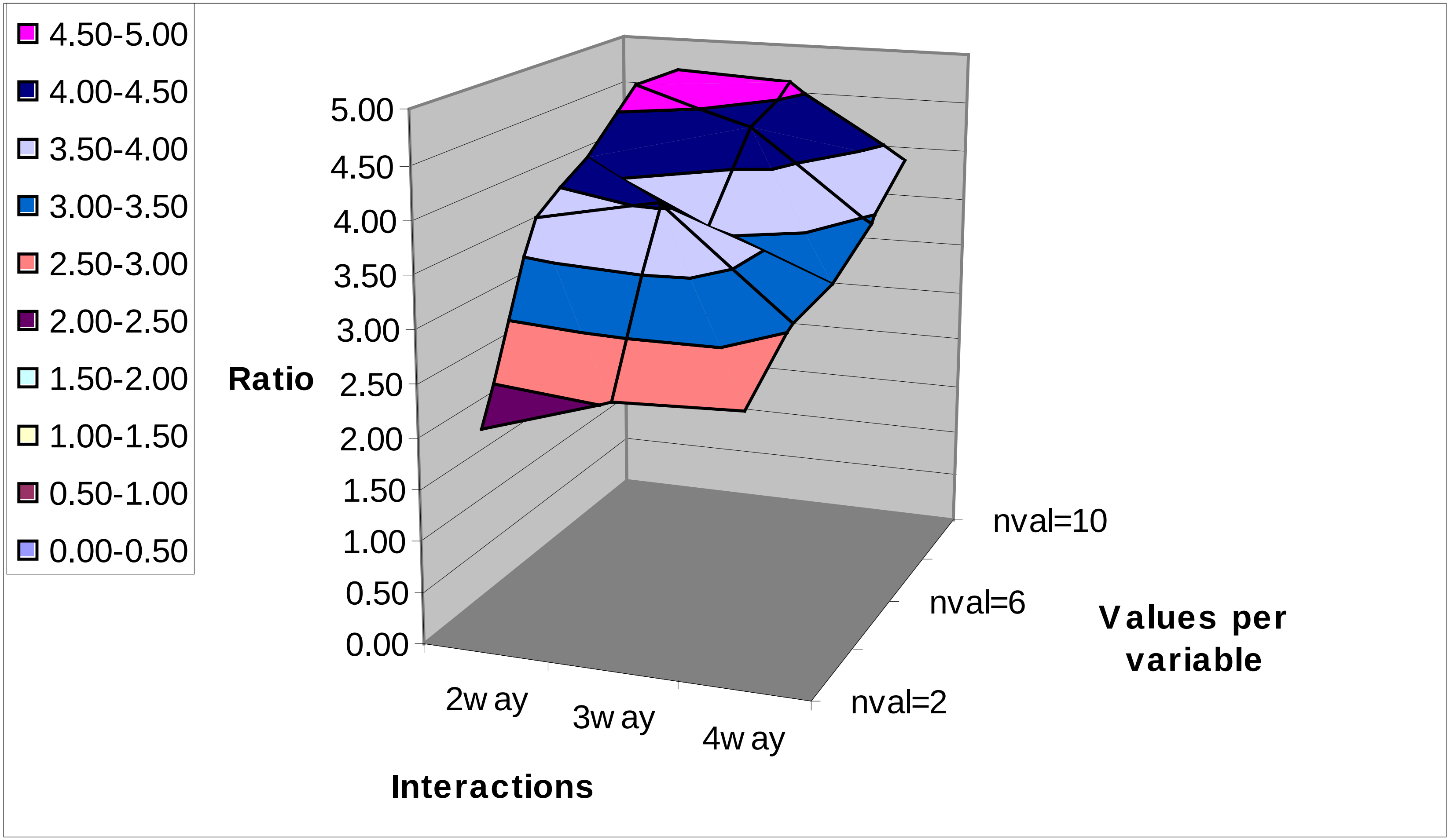
Crash Testing

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

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

- buffer overflows
- server crashes

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



NIST
National Institute of Standards and Technology

Embedded Assertions

Simple example:

```
assert( x != 0);    // ensure divisor is not zero
```

Or pre and post-conditions:

```
/requires amount >= 0;

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

Embedded Assertions

Assertions check properties of expected result:

```
    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

NIST
National Institute of
Standards and Technology

Tutorial Overview

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

New algorithms to make it practical

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

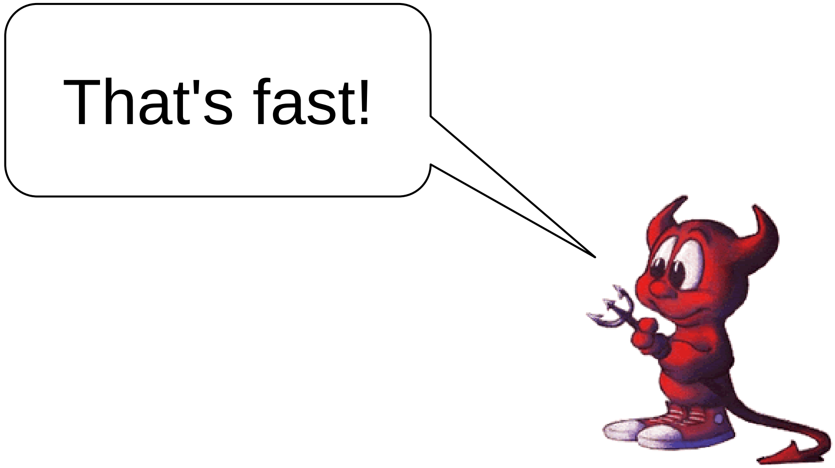
New algorithms

- Smaller test sets faster than other algorithms, 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

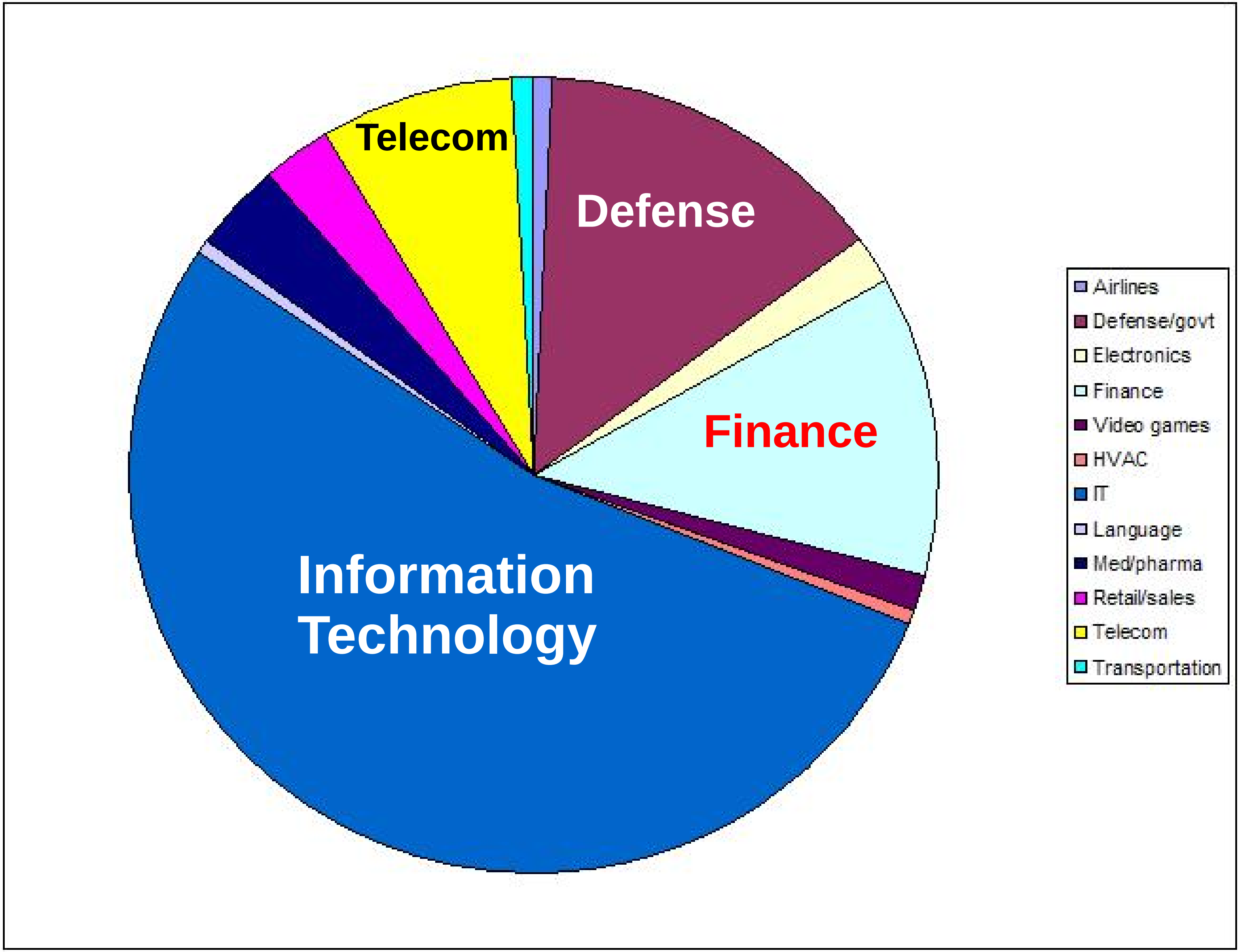
Traffic Collision Avoidance System (TCAS): $2^7 3^2 4^1 10^2$

Times in seconds

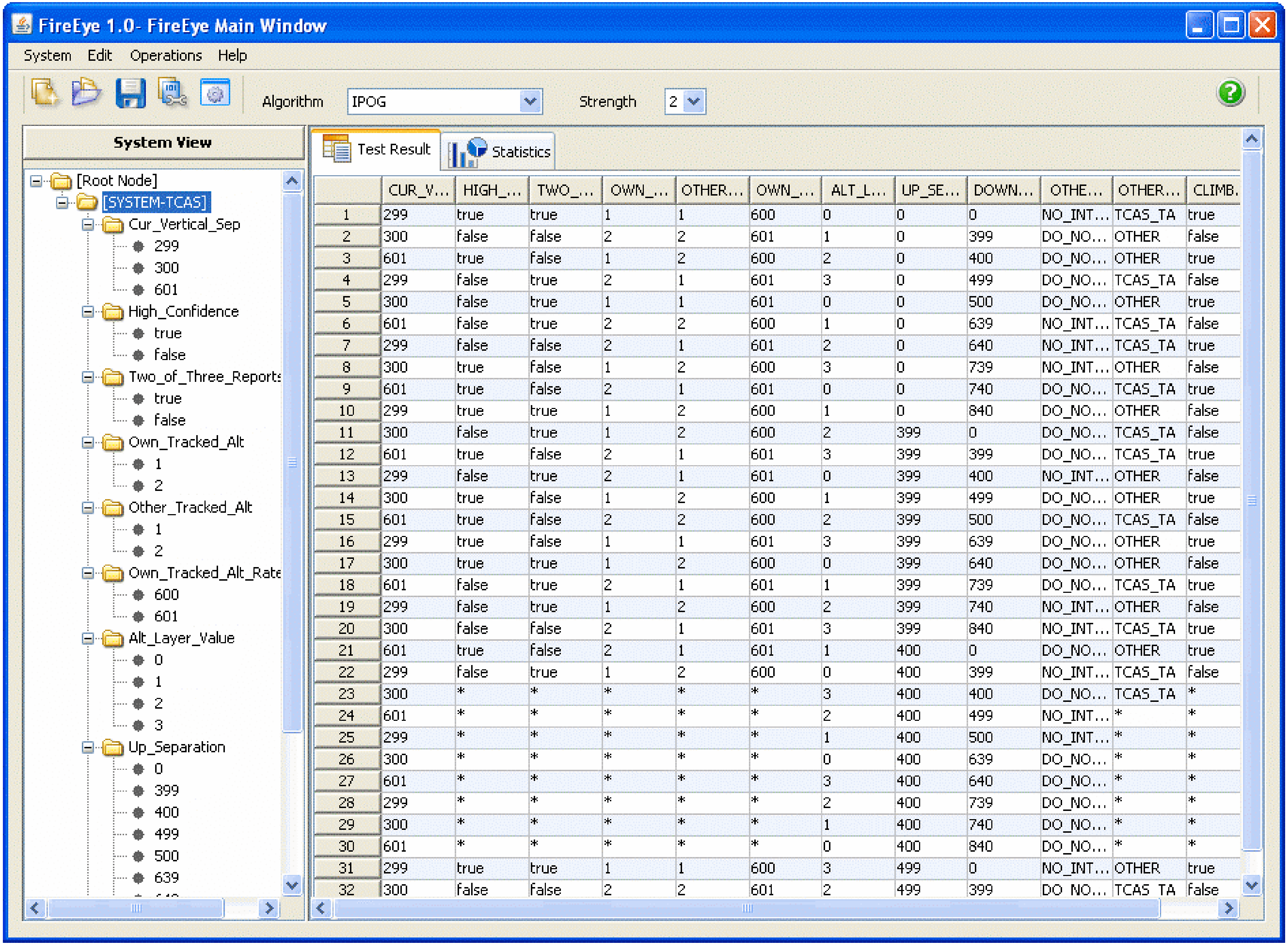


Unlike diet plans,
results ARE typical.

ACTS Users - 340+ in April 2010



ACTS Tool



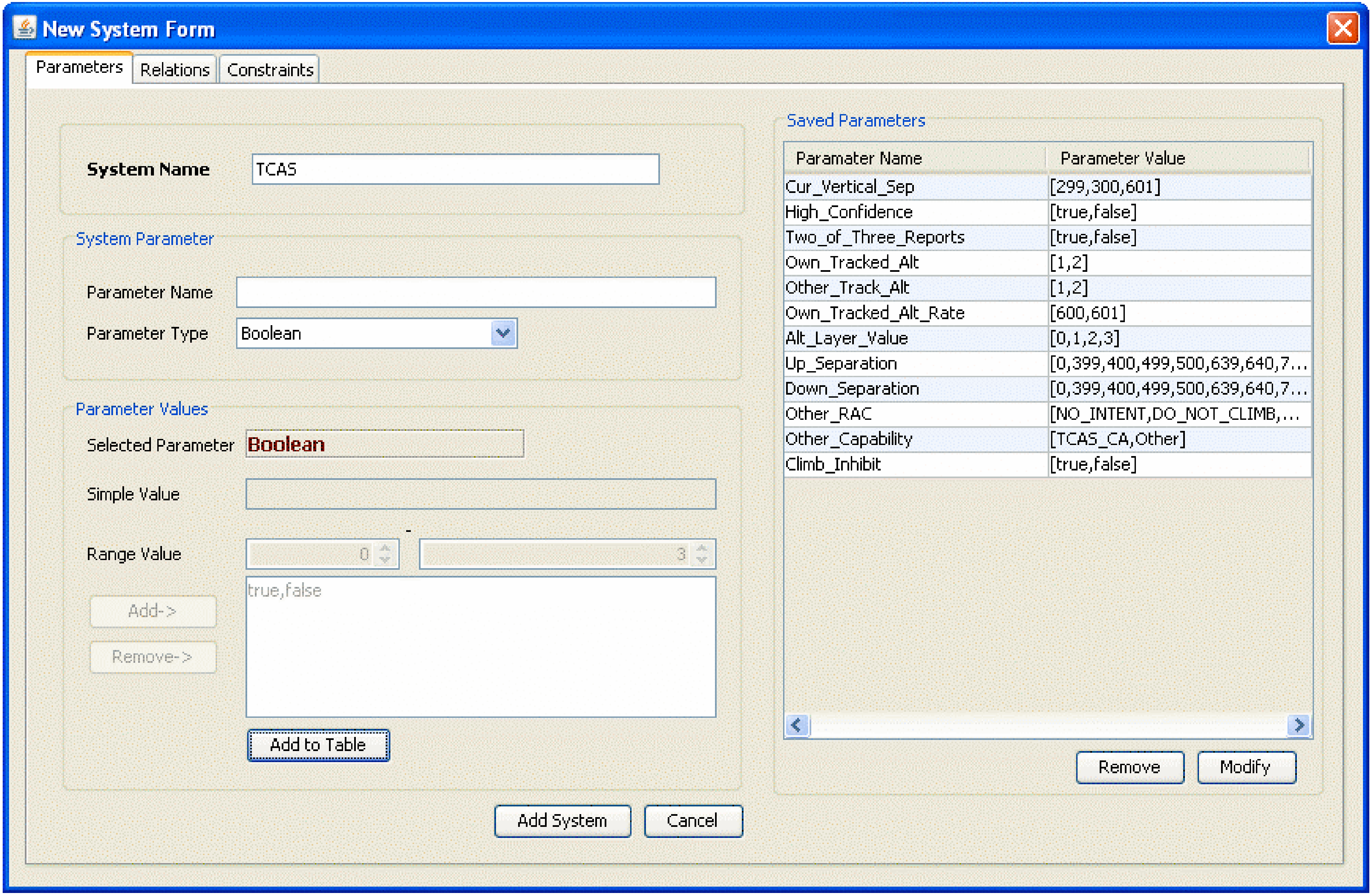
Algorithm IPOG

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

Defining a new system



New System Form

System Name: TCAS

System Parameter

Parameter Name:

Parameter Type: Boolean

Parameter Values

Selected Parameter: Boolean

Simple Value:

Range Value: 0 - 3

true,false

Saved Parameters

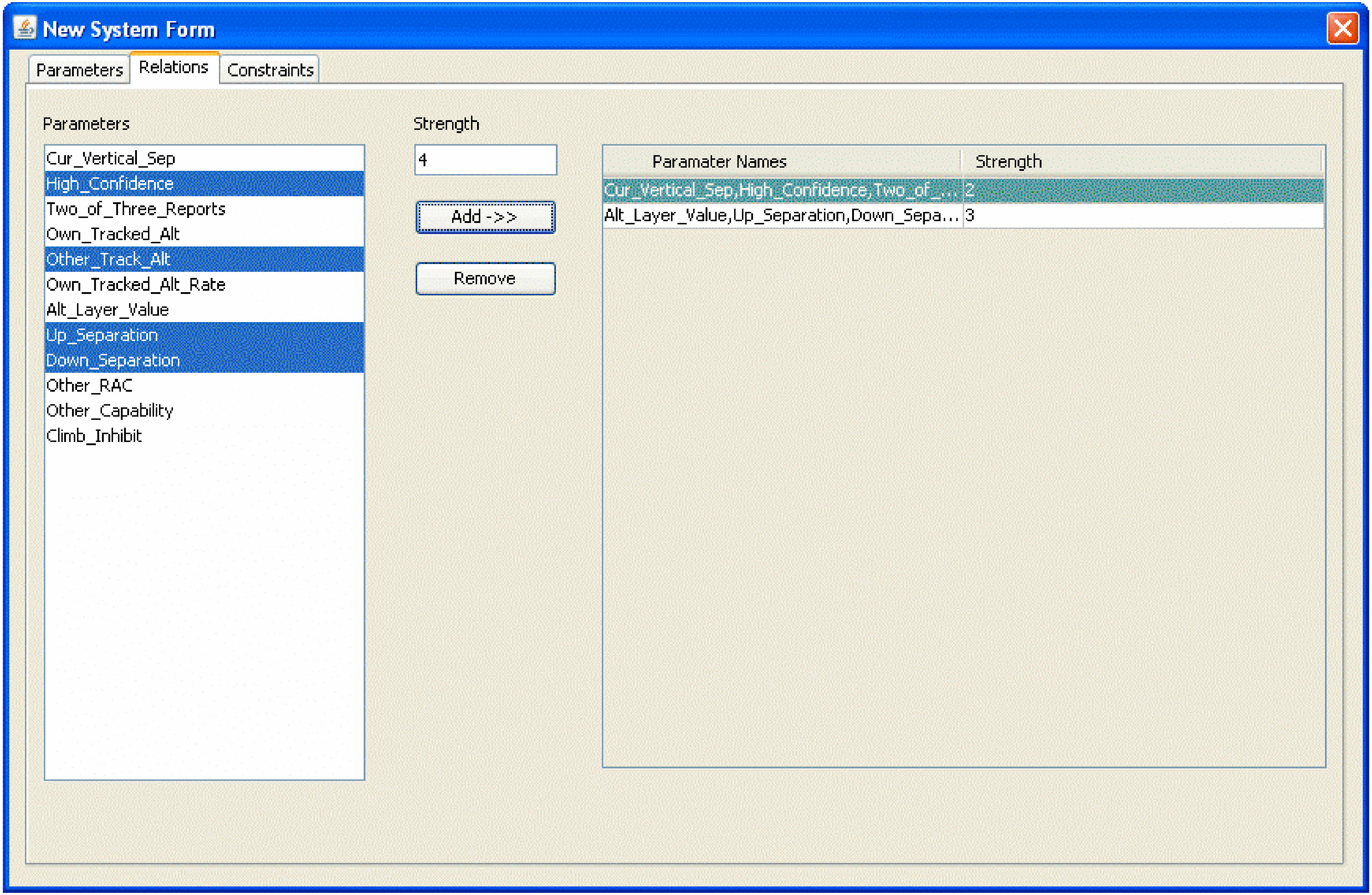
Paramater Name	Parameter Value
Cur_Vertical_Sep	[299,300,601]
High_Confidence	[true,false]
Two_of_Three_Reports	[true,false]
Own_Tracked_Alt	[1,2]
Other_Track_Alt	[1,2]
Own_Tracked_Alt_Rate	[600,601]
Alt_Layer_Value	[0,1,2,3]
Up_Separation	[0,399,400,499,500,639,640,7...
Down_Separation	[0,399,400,499,500,639,640,7...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

Variable interaction strength

- May want stronger interaction tests for some parameters
  - Example: 10 parameters,
  - Create 2-way covering array for P1 .. P10
  - May want 4-way testing for a subset P2, P4, P5, P6, P7
- Makes testing more efficient, saves on total number of tests

Variable interaction strength

National Institute of Standards and Technology



New System Form

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

Constraint 1: `(OS = “Windows”) => (Browser = “IE” || Browser = “FireFox” || Browser = “Netscape”)`

where OS and Browser are two parameters of type Enum.

(if OS is Windows, then Browser has to be IE, FireFox, or Netscape)

Constraint 2: `(P1 > 100) || (P2 > 100)`

where P1 and P2 are two parameters of type Number or Range.

(P1 or P2 must be greater than 100)

Constraint 3: `(P1 > P2) => (P3 > P4)`

where P1, P2, P3, and P4 are parameters of type Number or Range.

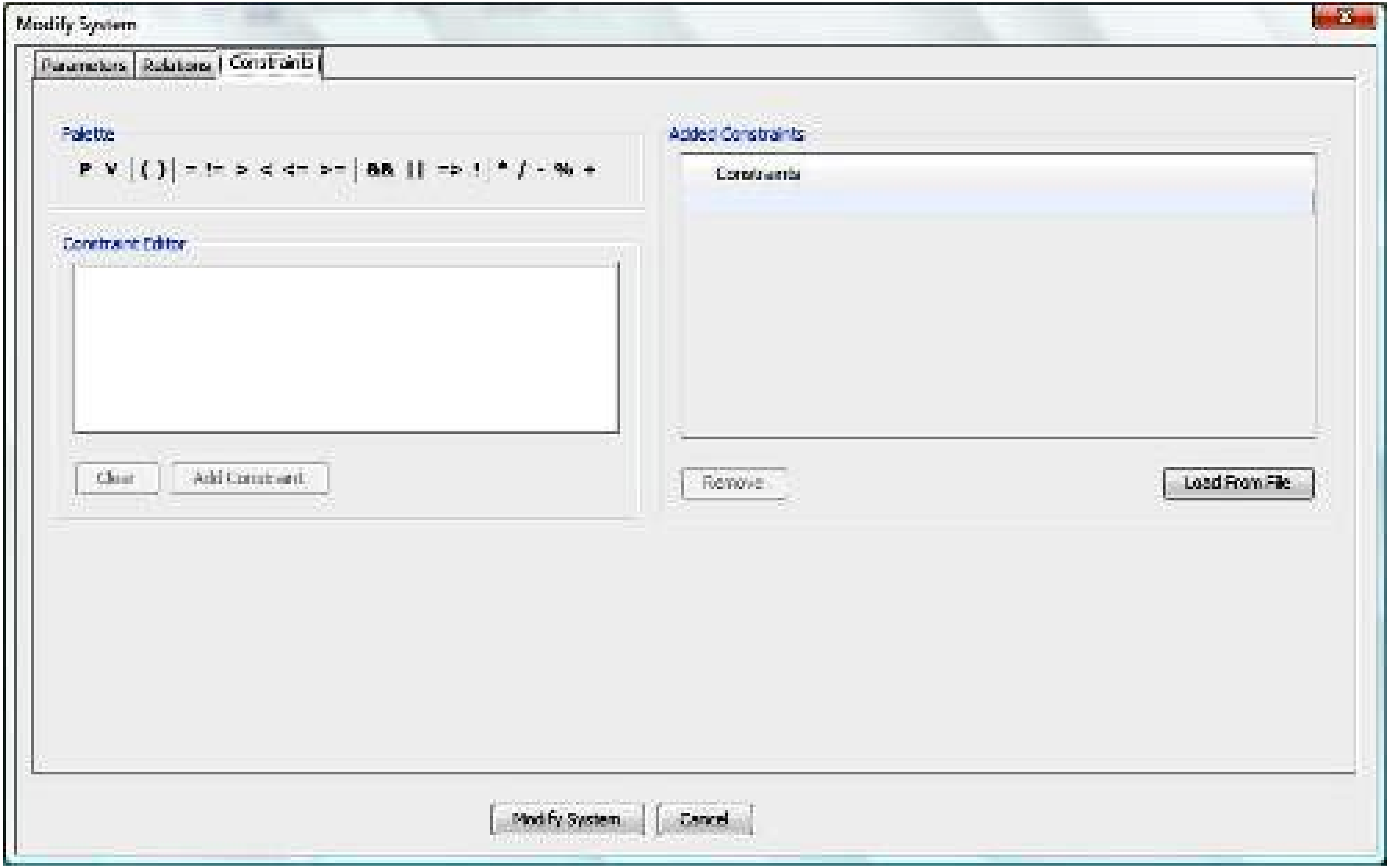
(if P1 is greater than P2, then P3 must be greater than P4)

Constraint 4: `(P1 = true || P2 >= 100) => (P3 = “ABC”)`

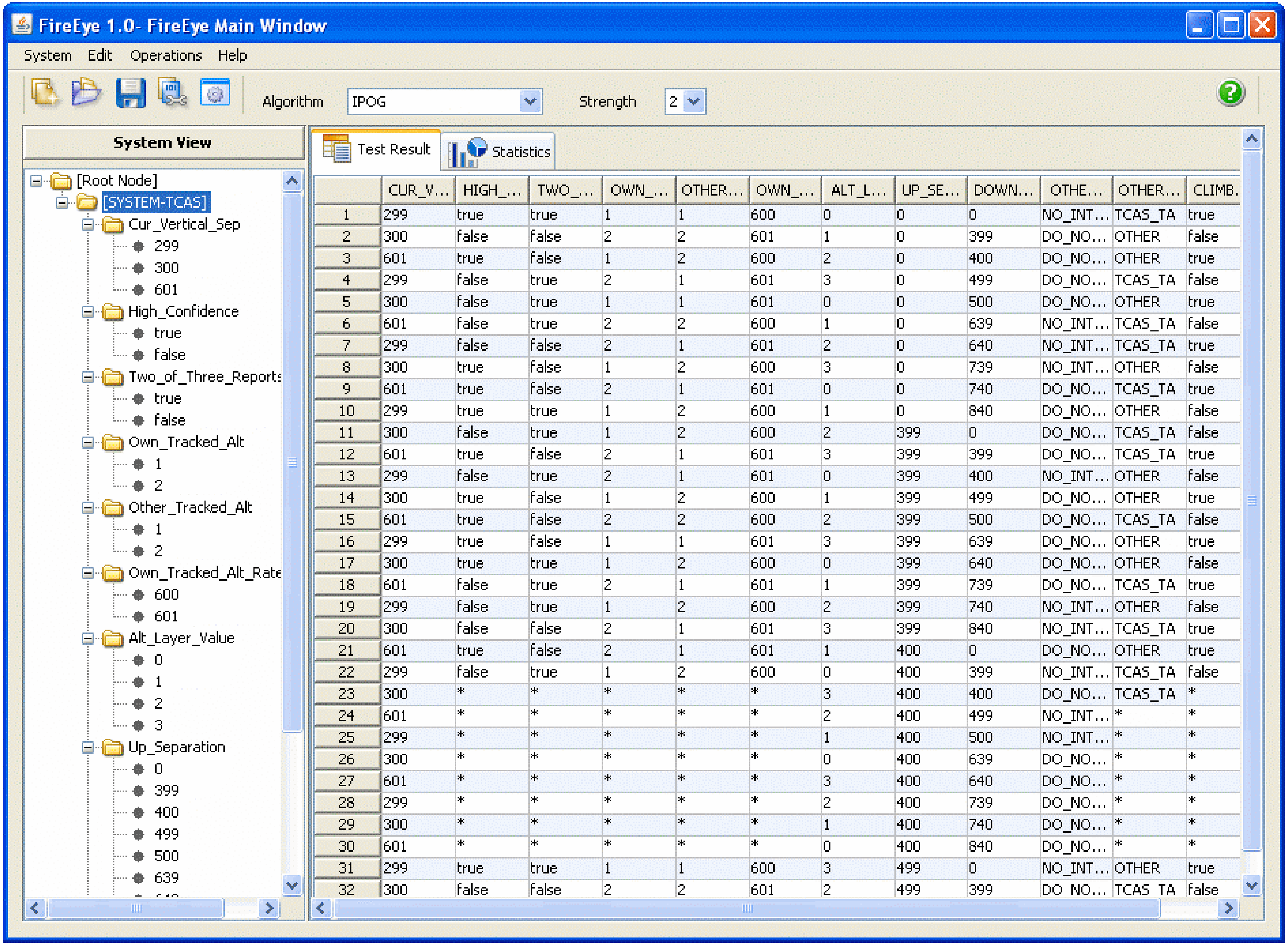
where P1 is a Boolean parameter, P2 is a parameter of type Number or Range, and P3 is of type Enum.

(if P1 is true and P2 is greater than or equal to 100, then P3 must be “ABC”)

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

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

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

- Post-process output using Perl scripts, etc.

Output options

Mappable values

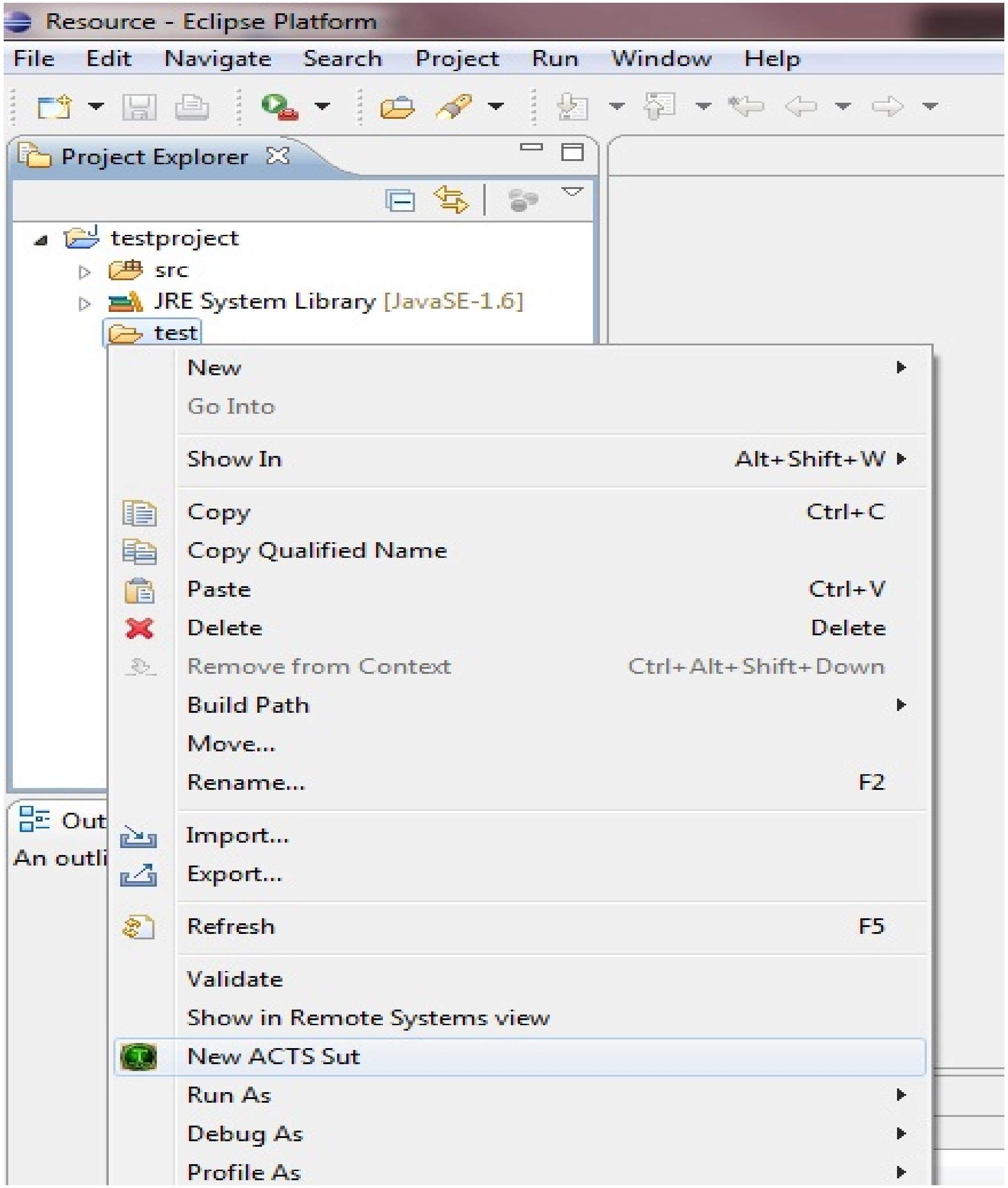
```
Degree of interaction
coverage: 2
Number of parameters: 12
Number of tests: 100

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

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

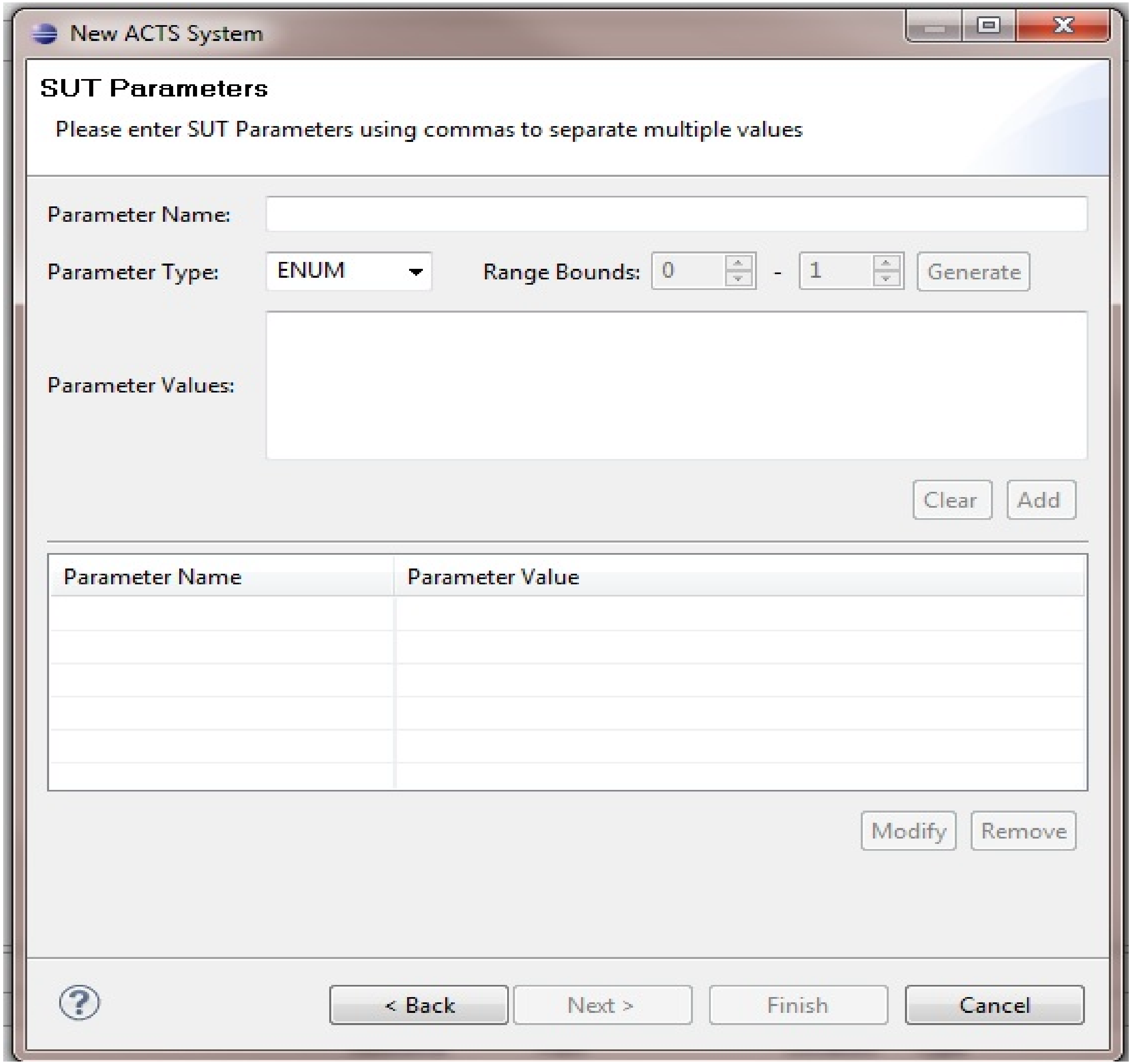
```
Degree of interaction coverage: 2
Number of parameters: 12
Maximum number of values per
parameter: 10
Number of configurations: 100
-----------------------------------
Configuration #1:

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



SUT Parameters

Please enter SUT Parameters using commas to separate multiple values

Parameter Name:

Parameter Type: ENUM

Range Bounds: 0 - 1

Parameter Values:

Parameter Name	Parameter Value

Defining parameters and values

Tutorial Overview

1. What is combinatorial testing?
2. Why are we doing this?
3. How is it used and how long does it take?
4. What tools are available?
5. **What's next?**
   - **Combinatorial coverage measurement**
   - **Combinatorial sequence testing**
   - **Fault location**
   - **Pairwise test prioritization**

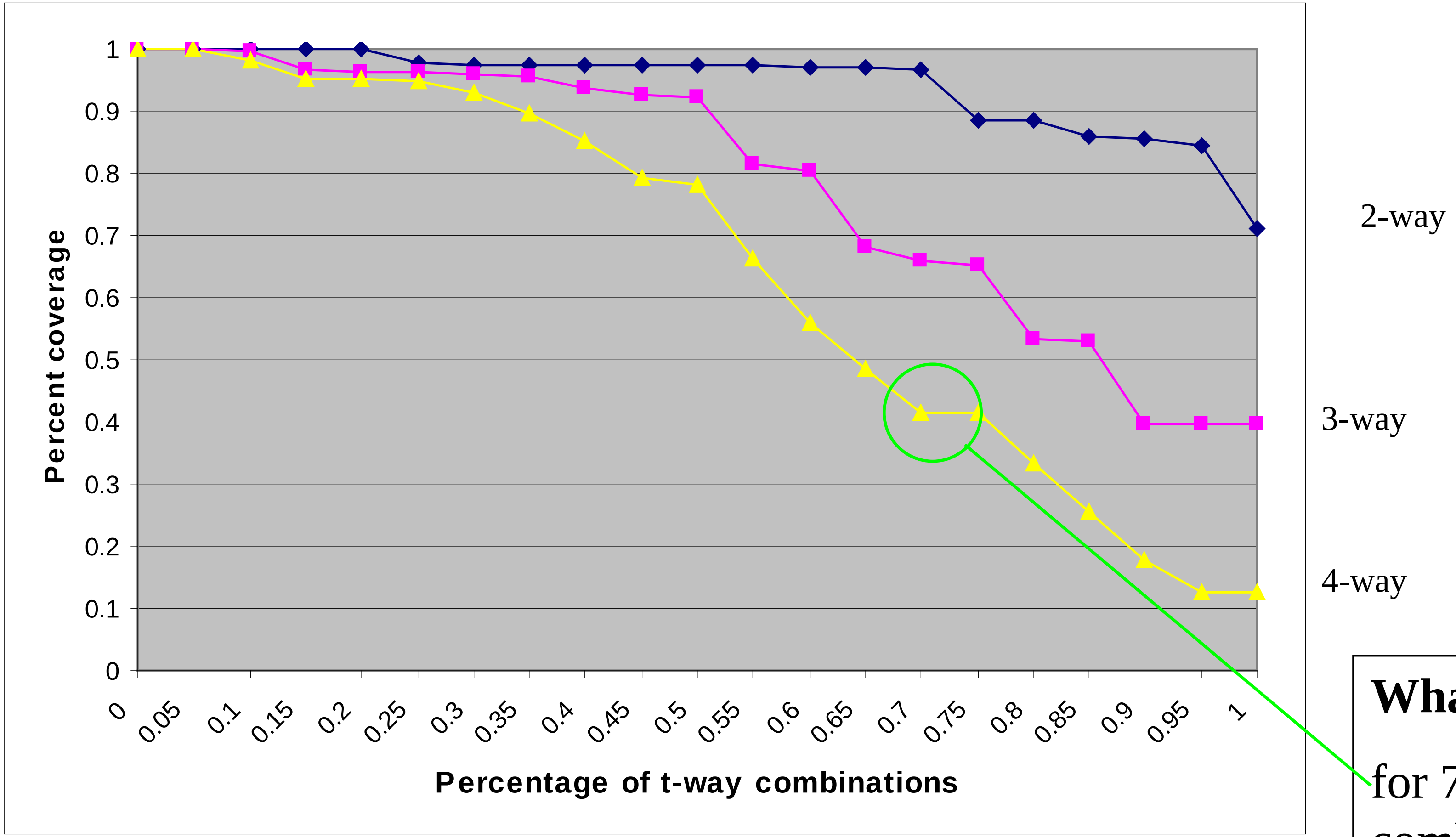
Combinatorial Coverage Measurement

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

NIST
National Institute of
Standards and Technology

Combinatorial Coverage Measurement



Configuration coverage for $2^{79}3^{1}4^{1}6^{1}9^{1}$ inputs.

What this means:

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

- Measure <u>coverage provided by existing test sets</u>
- Compare across methodologies

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?

- Example:

Event	Description
a	connect air flow meter
b	connect pressure gauge
c	connect satellite link
d	connect pressure readout
e	engage drive motor
f	engage steering control

Most failure reports indicate something like:
'failure occurred when <event A> if B is already connected'.

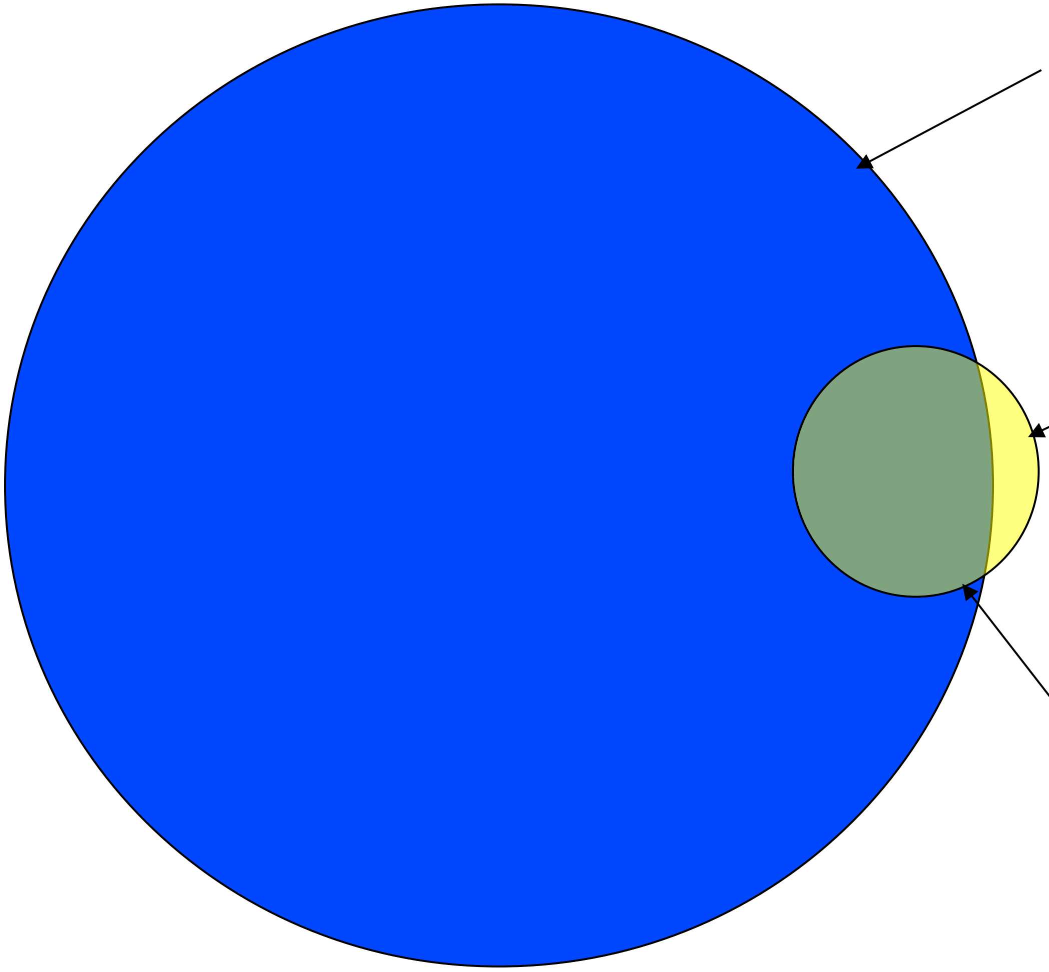
Combinatorial Sequence Testing

- With 5 events, all sequences = 5! = 120 tests
- Only 9 tests needed for all 3-way sequences, results even better for larger numbers of events
- Example:

	1	2	3	4	5
1	a	b	c	d	e
2	d	c	b	a	e
3	e	b	d	a	c
4	e	c	a	d	b
5	b	a	e	d	c
6	d	a	e	c	b
7	c	e	a	b	d
8	b	c	e	d	a
9	d	e	b	c	a

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

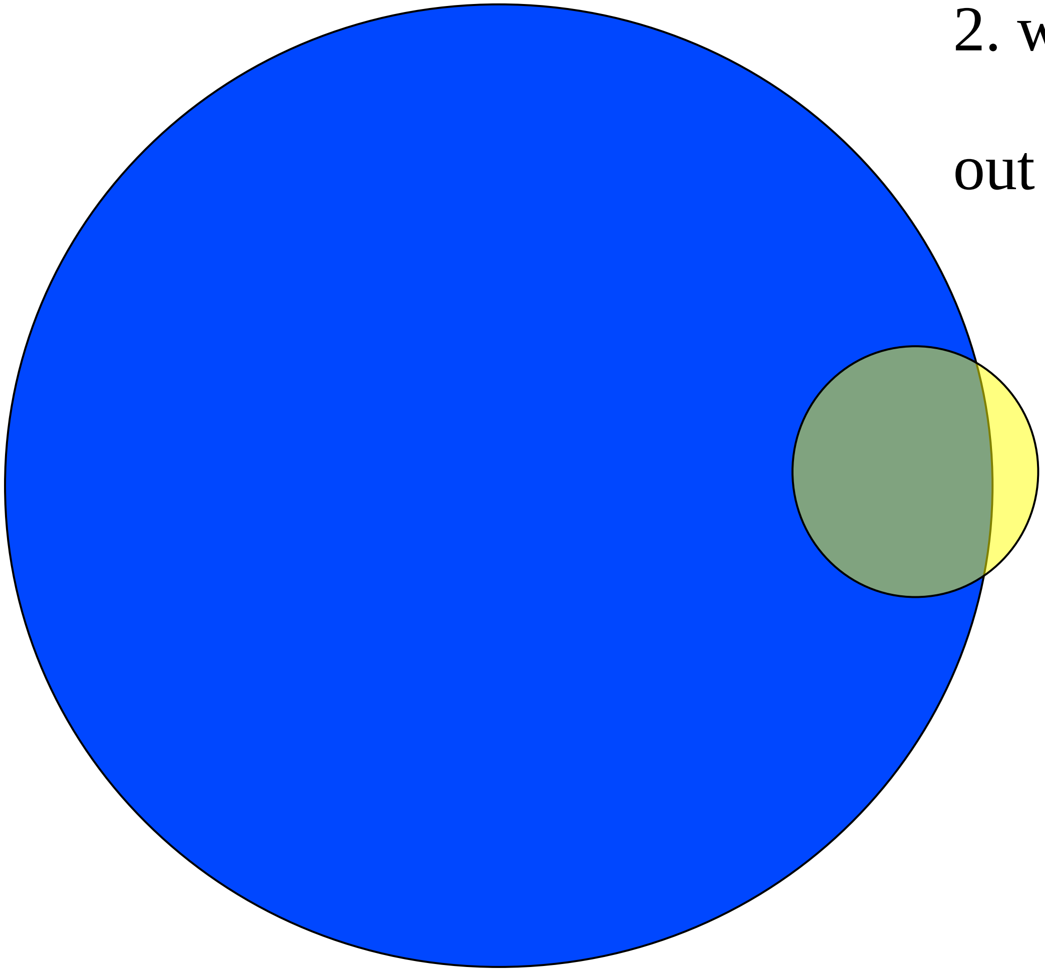
variable/value combinations in failing tests

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



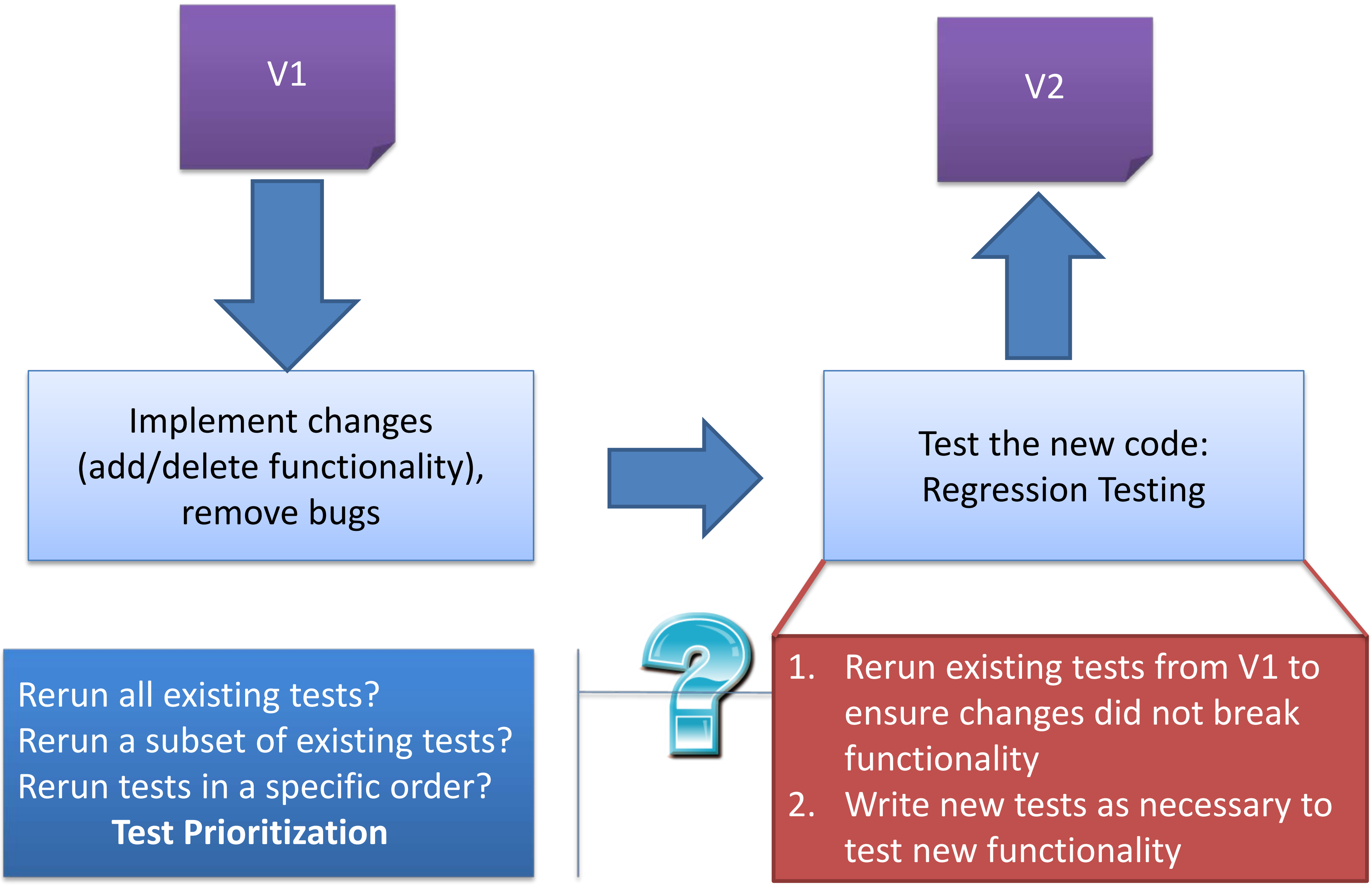
Example:
30 variables, 5 values each
= 445,331,250
5-way combinations

142,506 combinations in each test – which ones caused the fault?

Pairwise Test Prioritization

- Study of Mozilla web browser found 70% of defects with 2-way coverage; ~90% with 3-way; and 95% with 4-way. [Kuhn *et. al.,* 2002]
- Interaction testing of 109 software-controlled medical devices recalled by US FDA uncovered 97% of flaws with 2-way coverage; and only 3 required higher than 2. [Kuhn *et. al.,* 2004]
- Prior work is on generating pairwise adequate test suites. We examine pairwise testing in the context of test prioritization.

Regression Testing



V1

Implement changes (add/delete functionality), remove bugs

Test the new code: Regression Testing

V2

1. Rerun existing tests from V1 to ensure changes did not break functionality
2. Write new tests as necessary to test new functionality

Rerun all existing tests?
Rerun a subset of existing tests?
Rerun tests in a specific order?

Test Prioritization

Test Prioritization

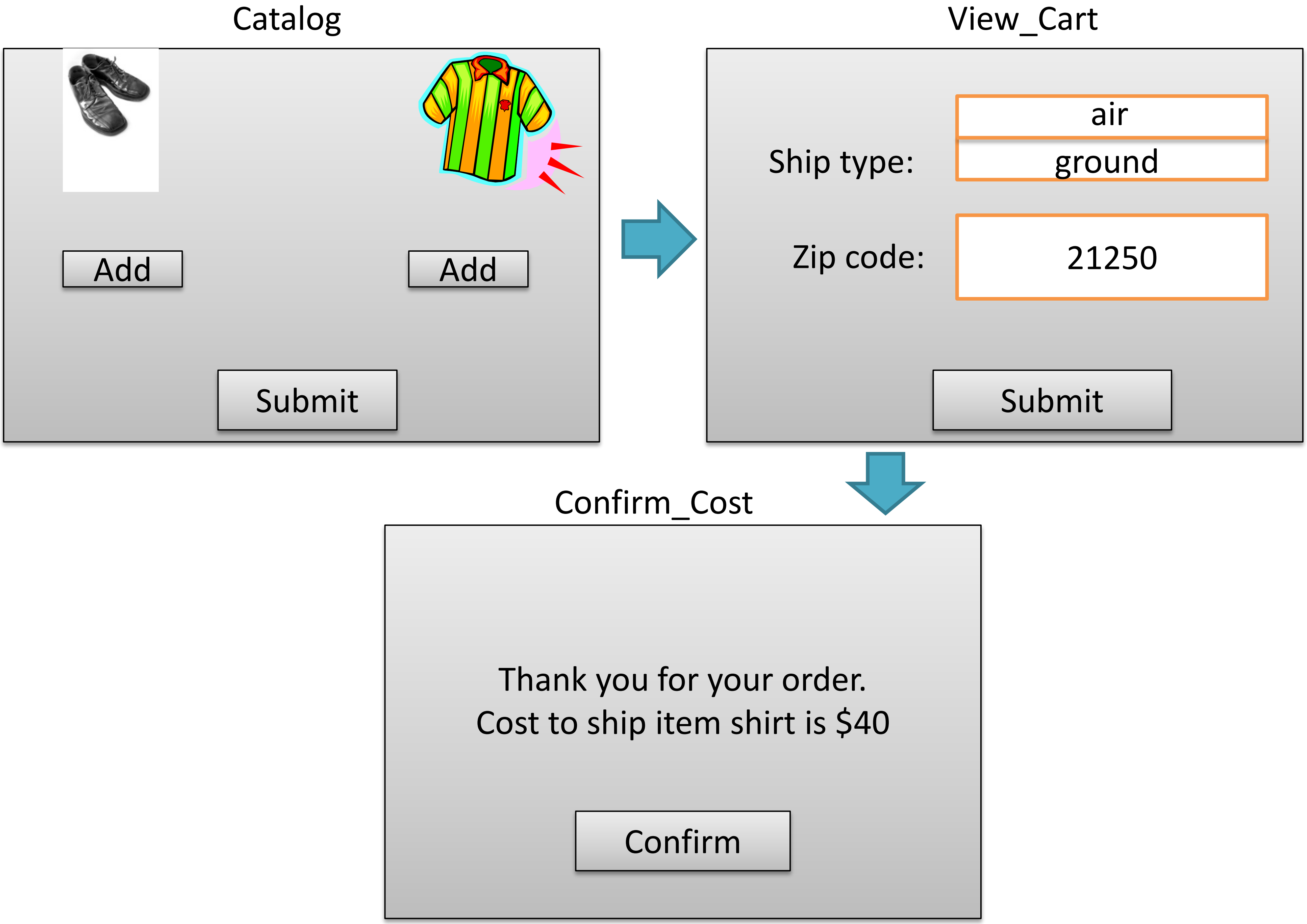
- Order existing tests based on some criterion to achieve a performance goal
  - Examples of criteria: total statement coverage, total method coverage
  - Performance goal: find faults quickly in test execution cycle
- We use number of pairwise (2-way) interactions a test covers as the prioritization criterion
- We evaluated the effectiveness of 2-way prioritization criterion with GUI and web-based systems
- Collaborative work with Renee Bryce at Utah State University, and Atif Memon at University of Maryland, College Park

Pairwise Interaction-based Prioritization: Underlying Idea

- Faults can be exposed by interactions of parameters set to values on different GUI windows/Web pages

- Order existing tests based on the number of pairwise interactions they cover to create test orders that find faults quickly

Example Web Application (Version 1)



Catalog

View_Cart

Ship type: air / ground

Zip code: 21250

Confirm_Cost

Thank you for your order.
Cost to ship item shirt is $40

Version 1

Example Test Case for V1

1 2

Test Case 1:

Catalog, item_name="shirt", item_weight="2"

View_Cart, ship_type="air", zip="21250"

3 4

Inter-window Pair-wise interactions:

(1,3) (1,4) (2,3) (2,4)

Version 1

Test Case 1:

1 2

Catalog, item_name="shirt", item_weight="2"

View_Cart, ship_type="air", zip="21250"

3 4

Pairwise interactions:
(1,3) (1,4) (2,3) (2,4)

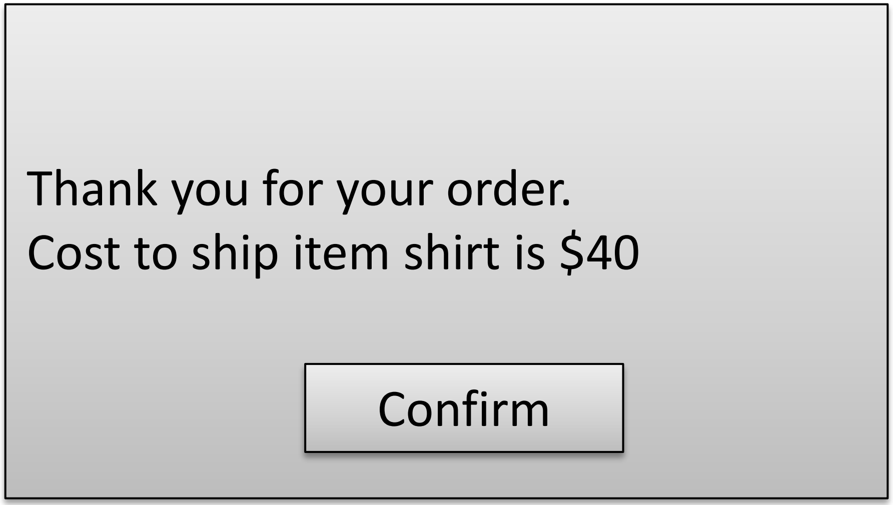
Underlying code after **Catalog** page is submitted:

```
$_SESSION['item_name'] = "shirt";
$_SESSION['item_weight']="2";

Display View_Cart page with
ship_type options <air, ground>
and zip textbox
```

Underlying code after **View_Cart** page is submitted:

```
Retrieve weight from $_SESSION and
ship_type from second window;

SELECT cost FROM Ship_Table WHERE
ship_type = air AND weight = 2;

Display database query response in
Confirm_Cost page
```

Confirm_Cost

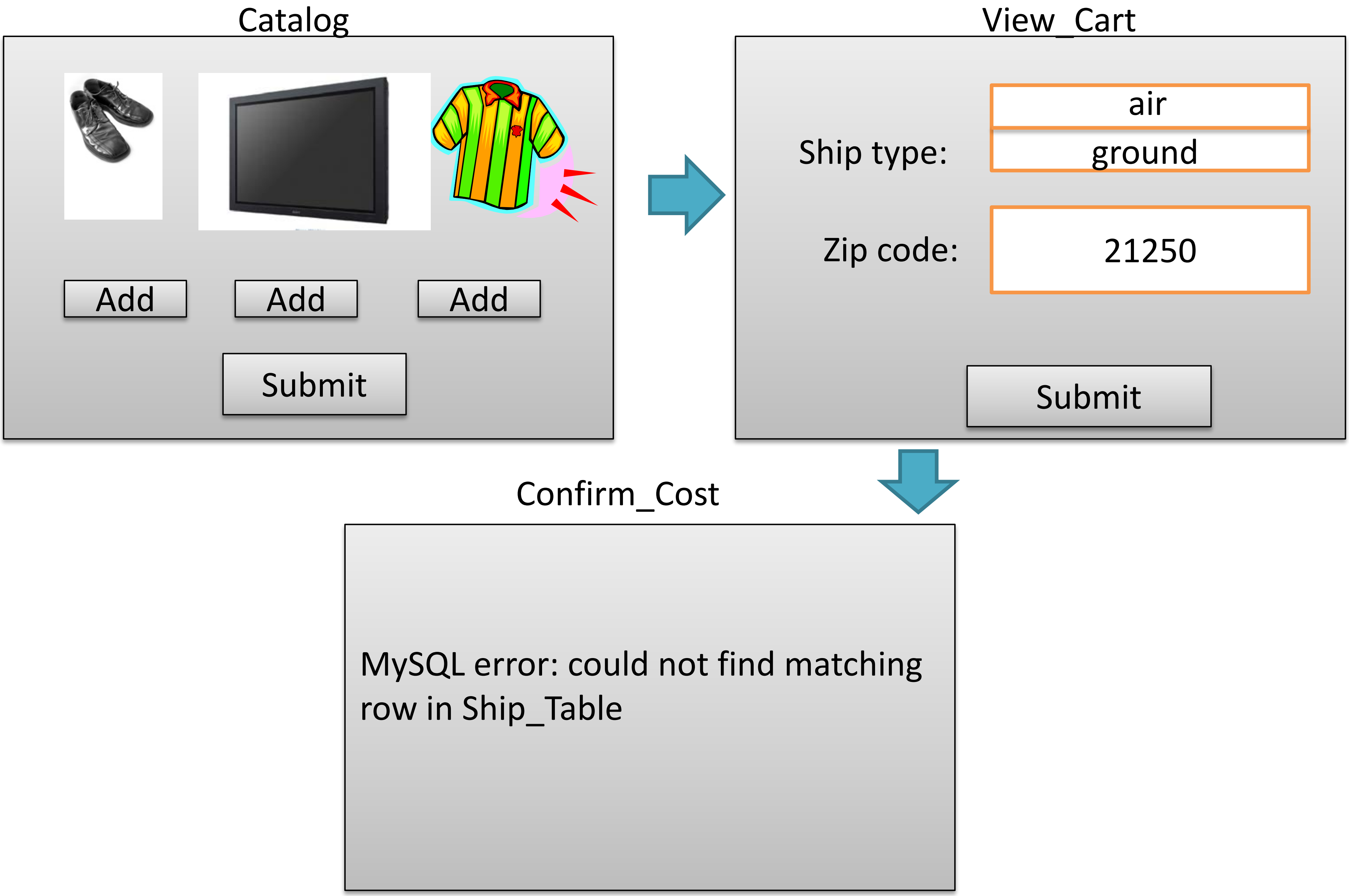


Thank you for your order.
Cost to ship item shirt is $40

Ship_Table

ship_type	weight	cost
air	1 to 2	20 to 40
ground	1 to 10	10 to 100

Update Items in Catalog (Version 2)



Ship type: air
ground

Zip code: 21250

MySQL error: could not find matching row in Ship_Table

Version 2

Test Case 2:

1 2

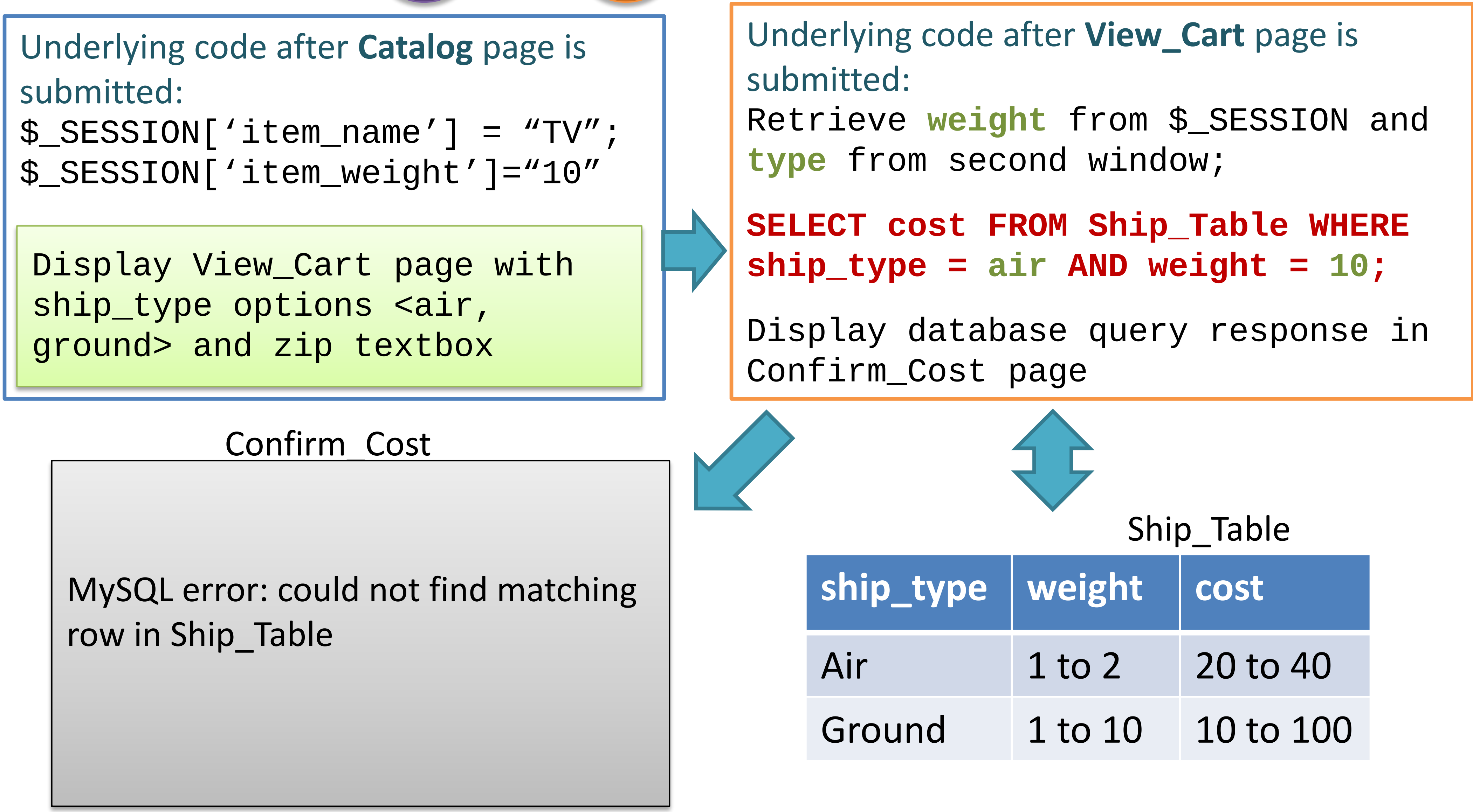
Catalog, item_name=“TV”, item_weight=“10”

View_Cart, ship_type=“air”, zip=“21250”

3 4

Pairwise interactions:

(1,3) (1,4) **(2,3)** (2,4)



Underlying code after **Catalog** page is submitted:

```
$_SESSION[‘item_name’] = “TV”;
$_SESSION[‘item_weight’]=“10”
```

```
Display View_Cart page with
ship_type options <air,
ground> and zip textbox
```

Underlying code after **View_Cart** page is submitted:

```
Retrieve weight from $_SESSION and
type from second window;

SELECT cost FROM Ship_Table WHERE
ship_type = air AND weight = 10;

Display database query response in
Confirm_Cost page
```

Confirm_Cost

MySQL error: could not find matching row in Ship_Table

Ship_Table

ship_type	weight	cost
Air	1 to 2	20 to 40
Ground	1 to 10	10 to 100

Pairwise Test Prioritization

Parameter-values

Window1	Window2	Window3
1	4	6
2	5	7
3		8
		9

Test	Windows visited	Parameter-values
T1	W1 -> W2 -> W1 -> W3	1 -> 4 -> 2 -> 8
T2	W2 -> W3	5 -> 6 -> 7
T3	W1 -> W3 -> W2 -> W1	3 -> 6 -> 4 -> 5 -> 1

Test	Pairwise interactions
T1	(1, 4) (1, 8) (4, 2) (4, 8) (2, 8)
T2	(5, 6) (5, 7)
T3	(3, 6) (3, 4) (3, 5) (6, 4) (6, 5) (6, 1) (4, 1) (5, 1)

Prioritize based on number of pairwise interactions in a test

Prioritized test order: T3, T1, T2

Experimental Evaluation

- Studied with 4 GUI and 3 web applications
  - 1000 to 18000 Lines of code
  - 125 to 900 test cases
- Evaluation used seeded faults
- Compared 2-way with 10 other prioritization criteria
- Measured rate of fault detection
  - Measures how quickly are faults detected
- Effectiveness of 2-way prioritization
  - the **best** or **second-best** criterion for 5 applications
  - among the **top-3** criteria for all 7 applications

R.C.Bryce, S. Sampath, A.M. Memon, "Developing a Single Model and Test Prioritization Strategies for Event-Driven Software", IEEE Transactions of Software Engineering, Preprint appeared online Jan 2010

CPUT: Tool to prioritize web test cases

- In collaboration with Renee Bryce at USU and Rick Kuhn and Raghu Kacker at NIST
- Java-based tool that parses web usage logs into test cases and prioritizes them
- Implements 4 prioritization criteria including 2-way
- Given a test suite, creates test orders as determined by the prioritization criterion

Ongoing work

- Hybrid test prioritization criteria
  - Hybrid of 2-way and other criteria for increased effectiveness in fault detection
- Extend application of 2-way prioritization to other software domains
- CPUT extensions
  - Command-line interface
  - Add test suite reduction functionality

Tutorial Overview

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

Conclusions

- Empirical research suggests that all software failures caused by interaction of a few parameters
- Combinatorial testing can exercise all t-way combinations of parameter values in a very tiny fraction of the time needed for exhaustive testing
- If all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations can provide strong assurance.
- New algorithms and faster processors make large-scale combinatorial testing possible - tools available, to be open source
- Project could produce better quality testing at lower cost for US industry and government

Please contact us if you are interested!

Rick Kuhn
kuhn@nist.gov

Raghu Kacker
raghu.kacker@nist.gov

Sreedevi Sampath
sampath@umbc.edu

http://csrc.nist.gov/acts

Or just search “combinatorial testing” - we’re #1!