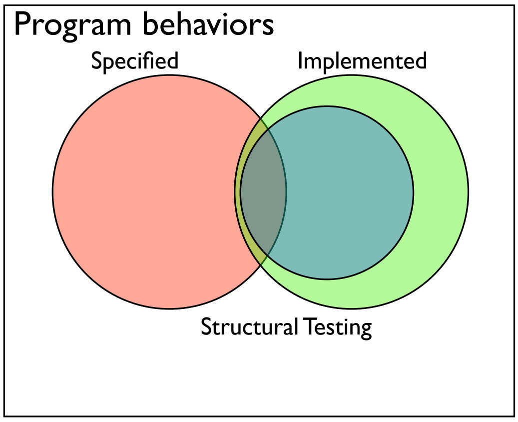
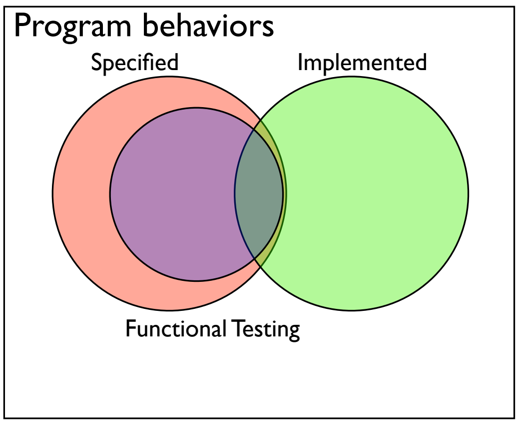
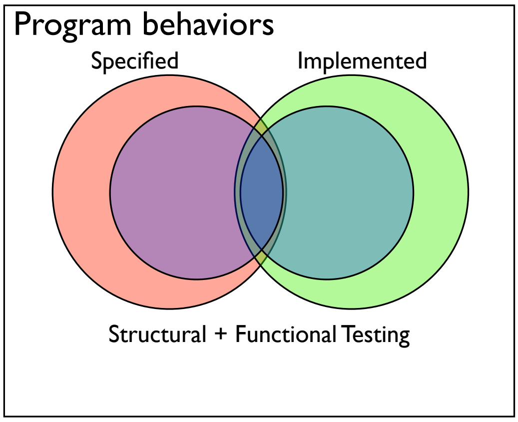
Specification-based Testing
Software Engineering
Gordon Fraser • Saarland University



Program behaviors
Specified
Implemented
Structural Testing

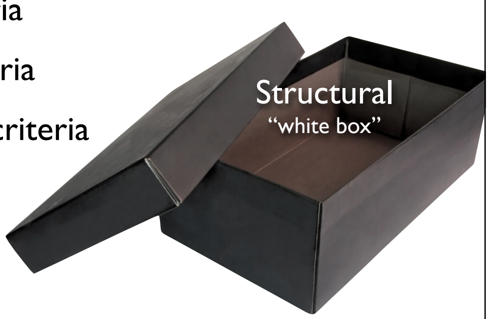


Program behaviors
Specified
Implemented
Functional Testing



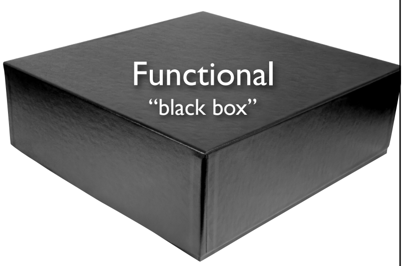
Structural Testing

- Path coverage criteria
- Logic coverage criteria
- Dataflow coverage criteria
- Mutation testing

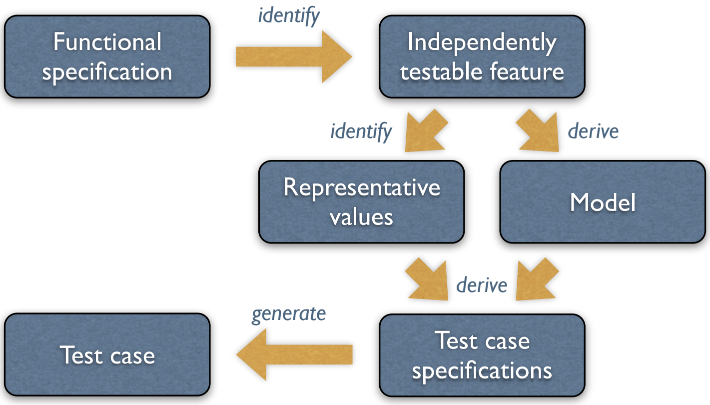


Functional Testing

- Boundary Value Testing
- Equivalence Class Testing
- Decision Table-Based Testing
- Combinatorial Testing
- Grammar-based Testing
- Model-based Testing

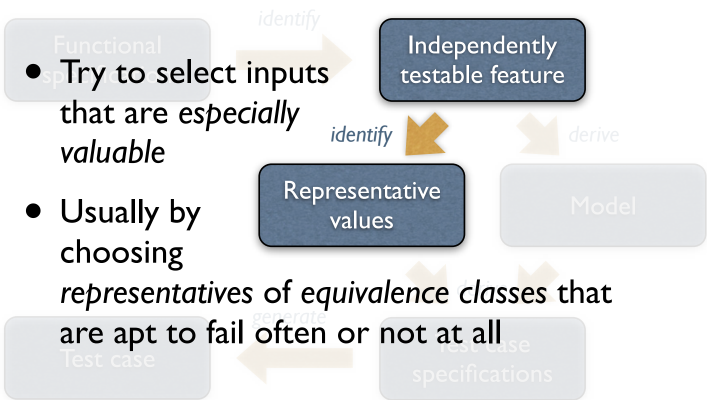


Specification-based Testing



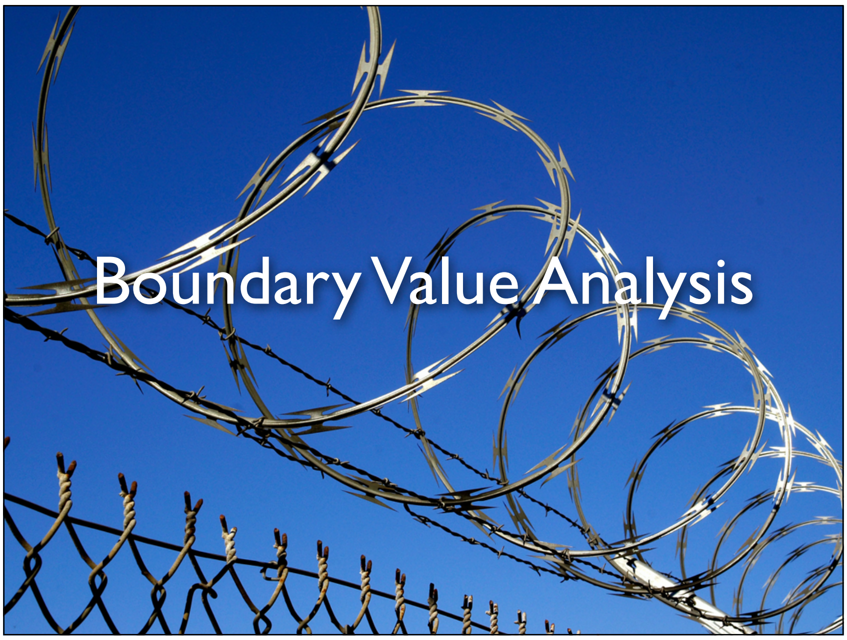
The main steps of a systematic approach to functional program testing
(from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

Representative Values



- Try to select inputs that are *especially valuable*
- Usually by choosing *representatives* of *equivalence classes* that are apt to fail often or not at all

The main steps of a systematic approach to functional program testing
(from Pezze + Young, “Software Testing and Analysis”, Chapter 10)



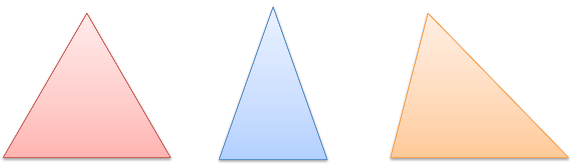
Single-fault assumption - therefore only one boundary value at a time

Boundary Value Testing

- Minimum, minimum+1, nominal, maximum-1, maximum
- Robustness testing
  Minimum-1, maximum+1
- Generalized - single fault assumption
  Boundary values for one, nominal values for others
- Worst-case testing
  All possible combinations

Single Fault Assumption

Failures occur rarely as the result of the simultaneous occurrence of two (or more) faults



Case	a	b	c	Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Invalid
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Invalid
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Invalid

Equivalence Partitioning

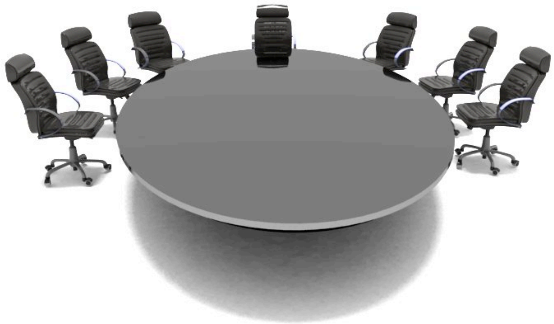
Input condition	Equivalence classes
range	one valid, two invalid (larger and smaller)
specific value	one valid, two invalid (larger and smaller)
member of a set	one valid, one invalid
boolean	one valid, one invalid

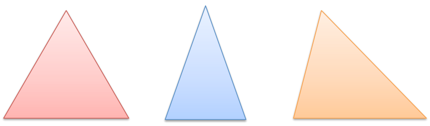
How do we choose equivalence classes? The key is to examine input conditions from the spec. Each input condition induces an equivalence class – valid and invalid inputs.

Equivalence Partitioning

- Weak equivalence class testing
  One test per equivalence class per input
- Strong equivalence class testing
  All combinations (cartesian product of equivalence classes)
- Robustness testing
  Include invalid values
- Combination with boundary value testing
  Test at boundaries of partitions

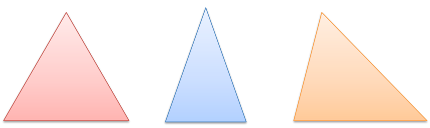
Decision Table Testing





a,b,c form a triangle	F	T	T	T	T	T	T	T	T
a = b	–	T	T	T	T	F	F	F	F
a = c	–	T	T	F	F	T	T	F	F
b = c	–	T	F	T	F	T	F	T	F
Not a triangle	X								
Scalene									X
Isosceles					X		X	X	
Equilateral		X							
Impossible			X	X		X			

Each column represents one test case

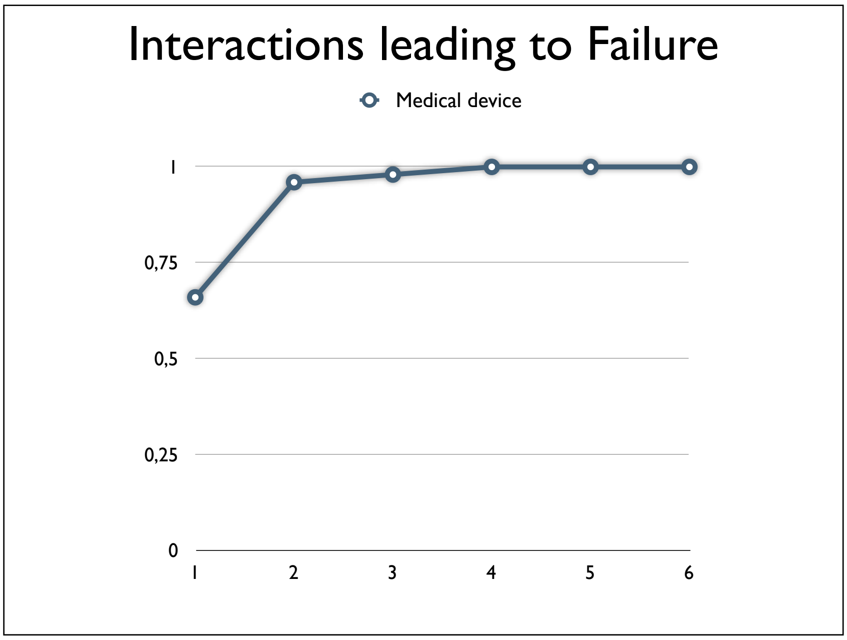


a < b + c	F	T	T	T	T	T	T	T	T	T	T
b < a + c		F	T	T	T	T	T	T	T	T	T
c < a + b			F	T	T	T	T	T	T	T	T
a = b				T	T	T	T	F	F	F	F
a = c				T	T	F	F	T	T	F	F
b = c				T	F	T	F	T	F	T	F
Not a triangle	X	X	X								
Scalene											X
Isosceles							X		X	X	
Equilateral				X							
Impossible					X	X		X			

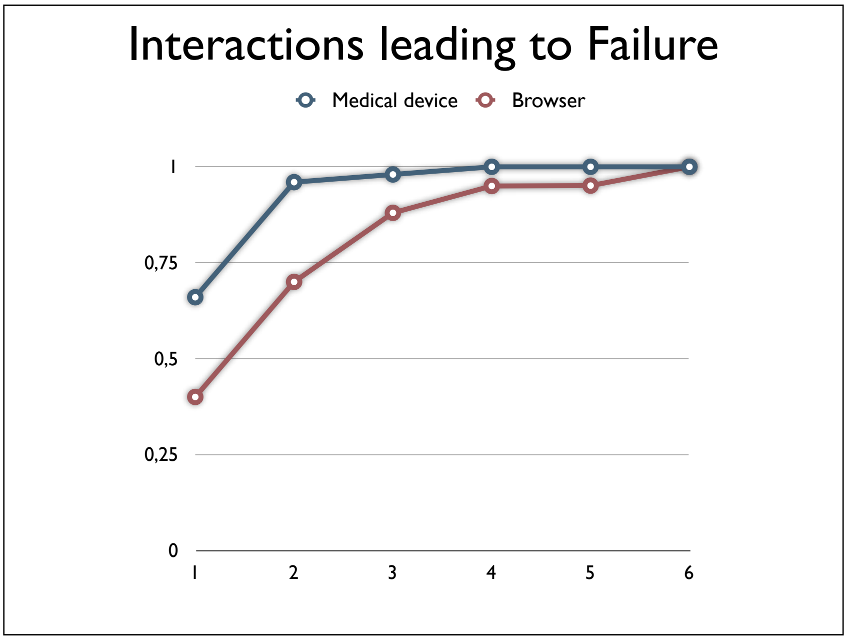
Decision Tables

- Outcome of decisions are not necessarily binary
- Tables can become huge
- Limited entry tables with N conditions have 2^N rules
- Don't care entries reduce the number of explicit rules by implying the existence of non-explicitly stated rules.

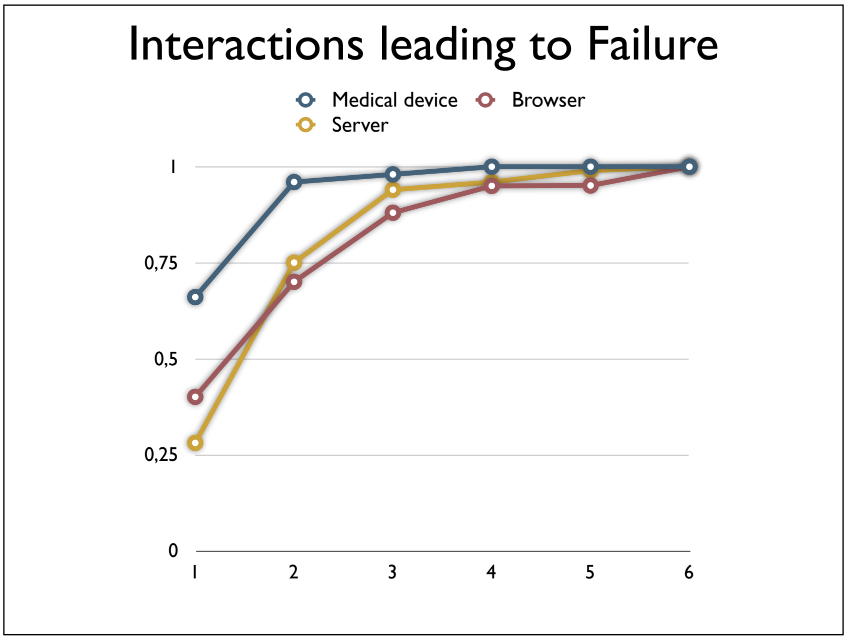

```
if (pressure < 10) {
  // do something
  if (volume > 300)  {
    // faulty code!  BOOM!
  }
  else {
    // good code, no problem
  }
}
else {
  // do something else
}
```



Interactions leading to Failure
Medical device
1
0,75
0,5
0,25
0
1
2
3
4
5
6

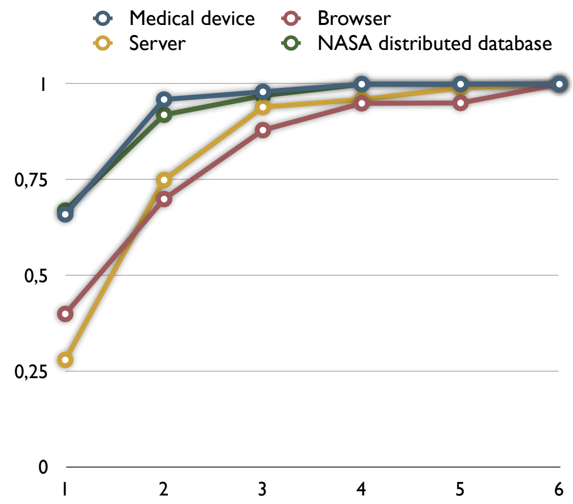


Interactions leading to Failure
Medical device
Browser
1
0,75
0,5
0,25
0
1
2
3
4
5
6

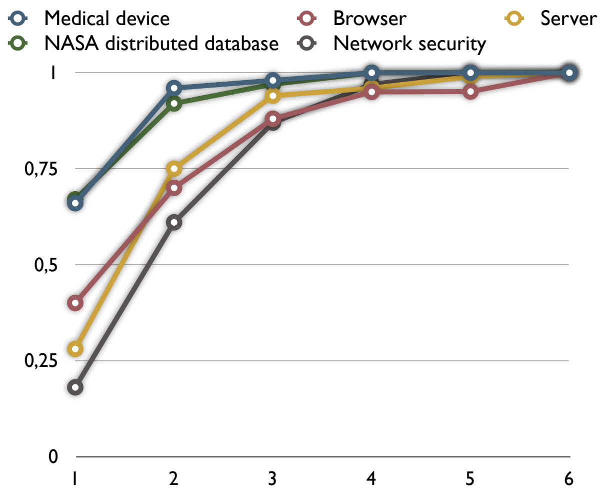


Interactions leading to Failure
Medical device
Browser
Server
1
0,75
0,5
0,25
0
1
2
3
4
5
6

Interactions leading to Failure

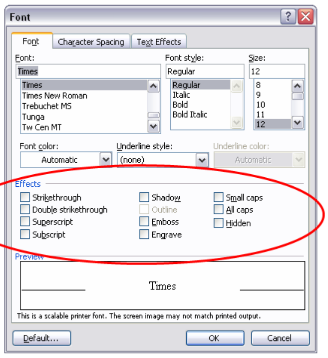


Interactions leading to Failure



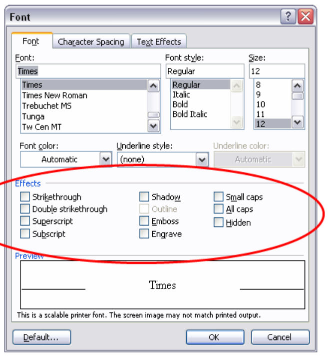
- Maximum interactions for fault triggering for studied applications was 6
  This correlates to the number of branch statements
- Reasonable evidence
  that maximum interaction strength for fault triggering is relatively small
- If all faults are triggered by the interaction of t or fewer variables
  then testing all t-way combinations can provide strong assurance
- Pairwise testing finds about 50% to 90% of flaws

How many tests?



- 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 …

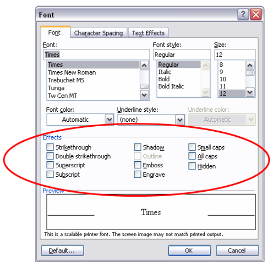
How many tests?



- 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

A Covering Array

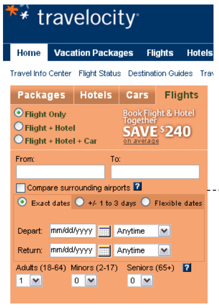


0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

0 = effect off
1 = effect on

- Each test covers 120 3-way combinations
- All 3-way combinations (960) in 13 tests
- Finding covering arrays is NP hard

Another familiar example



No silver bullet because:

Many values per variable

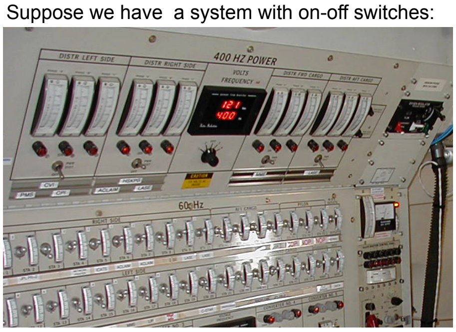
Need to abstract values

But we can still increase information per test

Plan: flt, flt+hotel, flt+hotel+car
From: CONUS, HI, Europe, Asia …
To: CONUS, HI, Europe, Asia …
Compare: yes, no
Date-type: exact, 1to3, flex
Depart: today, tomorrow, 1yr, Sun, Mon …
Return: today, tomorrow, 1yr, Sun, Mon …
Adults: 1, 2, 3, 4, 5, 6
Minors: 0, 1, 2, 3, 4, 5
Seniors: 0, 1, 2, 3, 4, 5

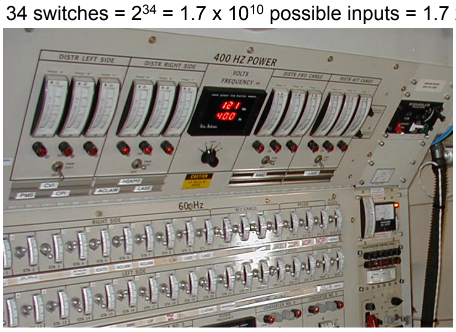
A Larger Example

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



How do we test this?

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



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

- 34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = **1.7 x 10^{10}** tests
- If only 3-way interactions, need only **33** tests
- For 4-way interactions, need only **85** tests

Two ways of using combinatorial testing

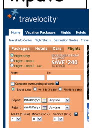
Use combinations here

or here

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

Configuration

Test data inputs



System under test

Testing Configurations

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

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

Combinatorial testing with existent test suite

1. Use t-way coverage for system configuration values
2. Apply existing tests

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

- Common practice in telecom industry

Generating Covering Arrays

- Search-based methods:
  - Mainly developed by scientists
  - Advantages: no restrictions on the input model, and very flexible, e.g., relatively easier to support parameter relations and constraints
  - Disadvantages: explicit search takes time, the resulting test sets are not optimal
- Algebraic methods:
  - Mainly developed by mathematicians
  - Advantages: very fast, and often produces optimal results
  - Disadvantages: limited applicability, difficult to support parameter relations and constraints

IPO Strategy

- Builds a t-way test set in an incremental manner
  - A t-way test set is first constructed for the first t parameters,
  - Then, the test set is extended to generate a t-way test set for the first t + 1 parameters
  - The test set is repeatedly extended for each additional parameter.
- Two steps involved in each extension for a new parameter:
  - Horizontal growth: extends each existing test by adding one value of the new parameter
  - Vertical growth: adds new tests, if necessary

```
Strategy In-Parameter-Order
begin
    /* for the first t parameters p1, p2 , …,  pt*/
    T := {(v1, v2, …, vt) | v1, v2, …, vt are values of
           p1, p2, …, pt , respectively}
    if n = t then stop;
    /* for the remaining parameters */
    for parameter pi, i = t + 1, …, n do
    begin
        /* horizontal growth */
        for each test (v1, v2, …, vi-1) in T do
             replace it with (v1, v2, …, vi-1, vi), where vi is a value of pi
        /* vertical growth */
        while T does not cover all the interactions between pi and
               each of p1, p2, …, pi-1 do
             add a new test for p1, p2, …, pi to T;
     end
end
```

Example

- Consider a system with the following parameters and values:
  - parameter A has values A1 and A2
  - parameter B has values B1 and B2
  - parameter C has values C1, C2, C3

A	B
A1	B1
A1	B2
A2	B1
A2	B2

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1

Horizontal Growth

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

Vertical Growth

Example

- Testing VoIP software:
- Caller, VoIP server, client
- CallerOS: Windows, Mac
- ServerOS: Linux, Sun, Windows
- CalleeOS: Windows, Mac

Example

Caller	Server	Callee
Win	Lin	Win
Win	Sun	Mac
Win	Win	Win
Mac	Lin	Mac
Mac	Sun	Win
Mac	Win	Mac

1. Pairwise testing ~~protects against~~ *might find some* pairwise bugs
2. while dramatically reducing the number of tests to perform *compared to testing all combinations, but not necessarily compared to testing just the combinations that matter.*
3. which is especially cool because pairwise bugs *might* represent the majority of combinatoric bugs *or might not, depending on the actual dependencies among variables in the product.*
4. and such bugs are a lot more likely to happen than ones that only happen with more *some* variables*, or less likely to happen, because user inputs are not uniformly distributed.*
5. Plus, you no longer need to create these tests by hand. *except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results.*