中山大學
SUN YAT-SEN UNIVERSITY

Part II [Problem]

4. Test Adequacy (Blackbox, level-2)

SE-307 Software Testing Techniques

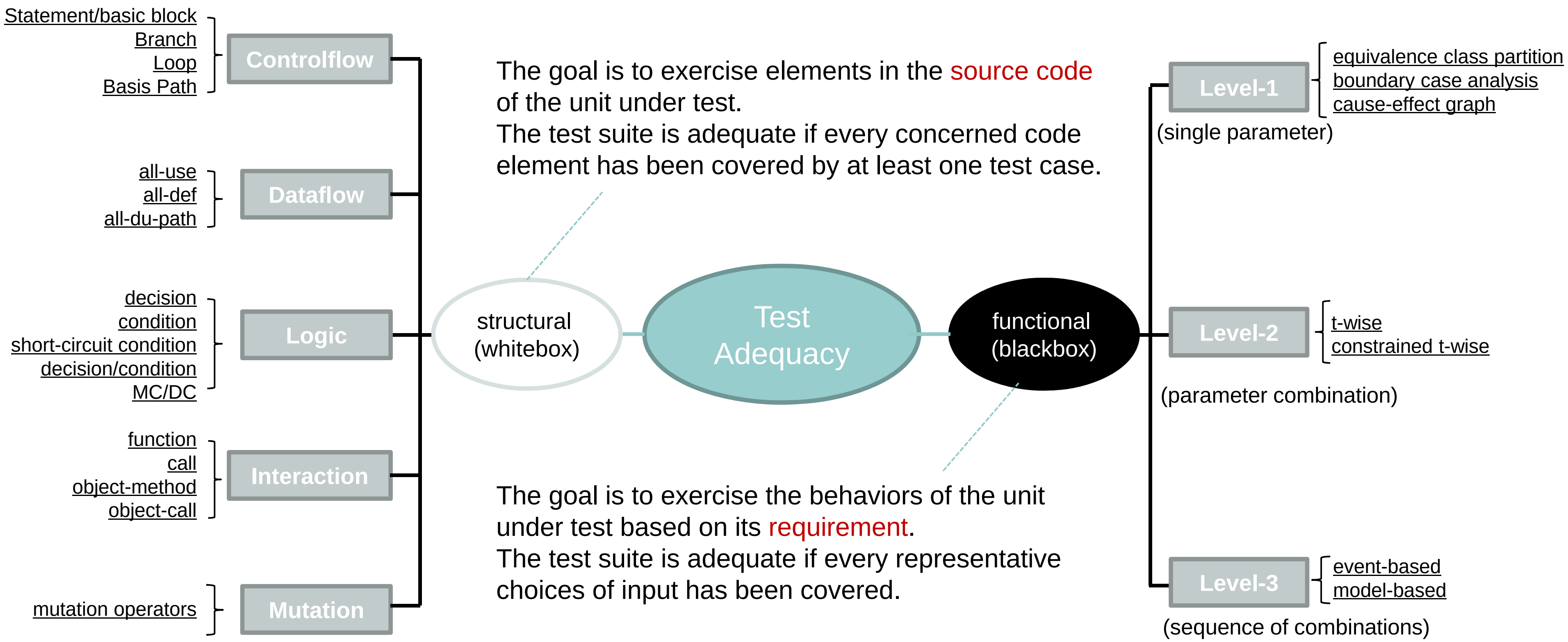
http://my.ss.sysu.edu.cn/wiki/display/SE307/Home

Instructor: Dr. Wang Xinming, School of Software, Sun Yat-Sen University

Review: Test Adequacy

- **Test adequacy** is about the questions:
  - What is the goal of test suite design?
  - Is our test suite ‘adequate’?
  - When can we stop designing new test cases?
- **Whitebox (Structural)** approach to test adequacy:
  - The goal is to exercise elements in the source code. e.g. statements, basic blocks, branches, loops, basis-paths, conditions, du-paths, function-calls, etc.
  - Our test suite is considered adequate **if every concerned code element is covered by at least one test case**.
- **Blackbox (Functional)** approach to test adequacy:
  - The goal is to exercise the behaviors of the units-under-test based on its requirement, without considering its implementation detail. Such a unit can be a method, a class, a sub-system, or the whole programs.
  - Our test suite is considered adequate **if it covers representative choices of input to the unit-under-test**.

Review



Test Adequacy

structural (whitebox)

The goal is to exercise elements in the source code of the unit under test.
The test suite is adequate if every concerned code element has been covered by at least one test case.

Controlflow: Statement/basic block, Branch, Loop, Basis Path

Dataflow: all-use, all-def, all-du-path

Logic: decision, condition, short-circuit condition, decision/condition, MC/DC

Interaction: function, call, object-method, object-call

Mutation: mutation operators

functional (blackbox)

The goal is to exercise the behaviors of the unit under test based on its requirement.
The test suite is adequate if every representative choices of input has been covered.

Level-1 (single parameter): equivalence class partition, boundary case analysis, cause-effect graph

Level-2 (parameter combination): t-wise, constrained t-wise

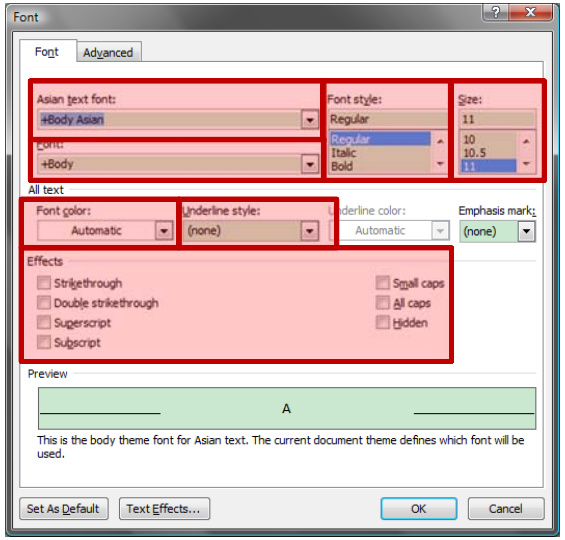
Level-3 (sequence of combinations): event-based, model-based

Review: 3 Levels of Blackbox Adequacy

- Level 1: ***Single*** input parameter
  - Cover the representative choices for **one single input parameter**.
  - e.g. the font type. (宋体 or 黑体 or Arial or ….)

Asian text font: +Body Asian

- Level 2: ***Combination*** of input parameters
  - Cover the representative **combinations of multiple parameters**.
  - e.g. settings in the whole font dialog. (宋体-加粗-11号-红色 or 宋体-普通-小四-黑色 or 黑体-加粗-12号-黄色 or …)



Asian text font: +Body Asian

Font: +Body

Font style: Regular (Regular, Italic, Bold)

Size: 11 (10, 10.5, 11)

Font color: Automatic

Underline style: (none)

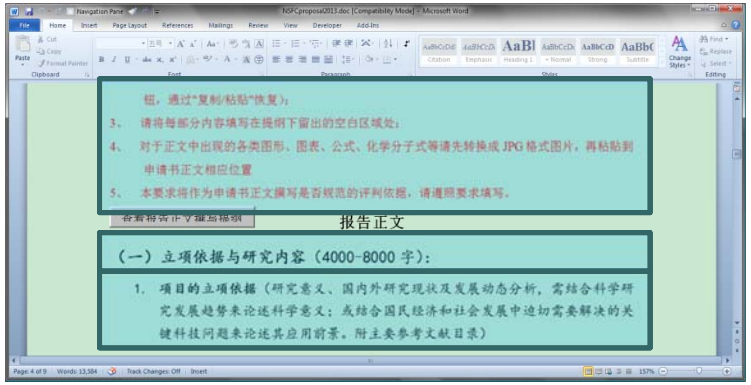
Underline color: Automatic

Emphasis mark: (none)

□ Strikethrough
□ Double strikethrough
□ Superscript
□ Subscript
□ Small caps
□ All caps
□ Hidden

This is the body theme font for Asian text. The current document theme defines which font will be used.

- Level 3: ***Sequence*** of parameter combinations
  - Cover the representative **sequences of parameter combinations**.
  - e.g. font settings for multiple paragraphs. (e.g. 为不同的段落设置不同的字体)。



钮，通过"复制/粘贴"恢复)；

3、 请将每部分内容填写在提纲下留出的空白区域处；

4、 对于正文中出现的各类图形、图表、公式、化学分子式等请先转换成 JPG 格式图片，再粘贴到申请书正文相应位置

5、 本要求将作为申请书正文撰写是否规范的评判依据，请遵照要求填写。

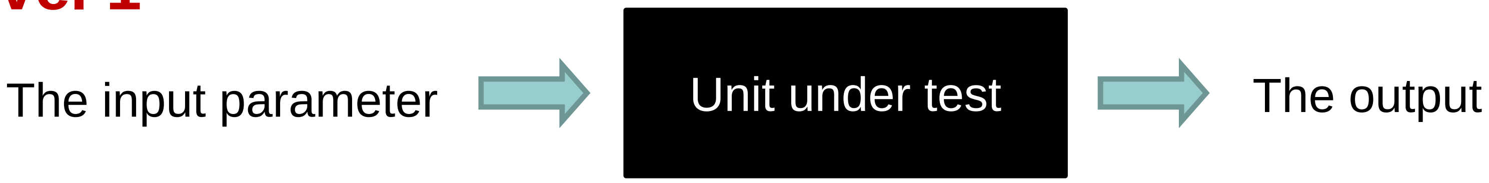
报告正文

（一）立项依据与研究内容（4000-8000 字）：

1. 项目的立项依据（研究意义、国内外研究现状及发展动态分析，需结合科学研究发展趋势来论述科学意义；或结合国民经济和社会发展中迫切需要解决的关键科技问题来论述其应用前景。附主要参考文献目录）

Illustration of The Three Levels

Level 1

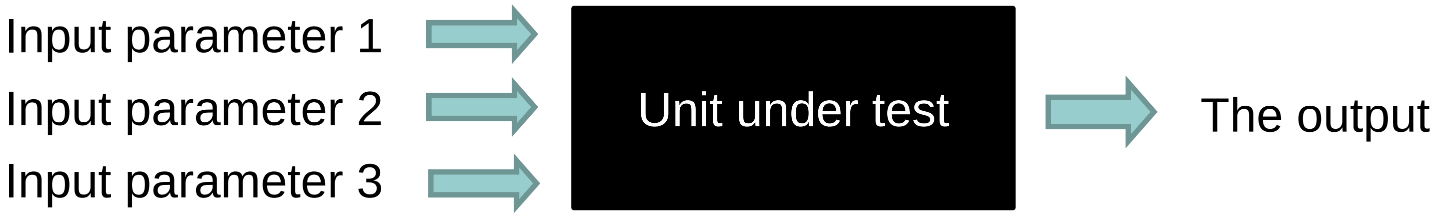


The input parameter

Unit under test

The output

Level 2



Input parameter 1

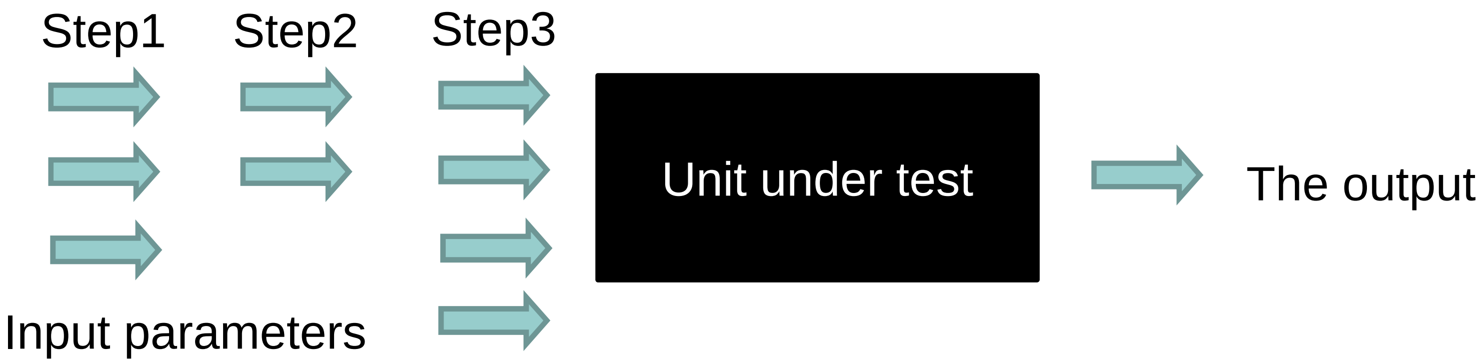
Input parameter 2

Input parameter 3

Unit under test

The output

Level 3



Step1

Step2

Step3

Input parameters

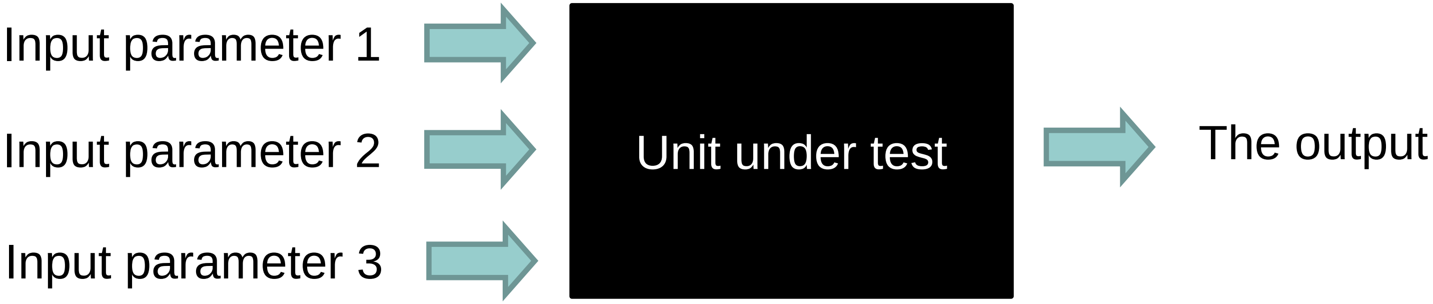
Unit under test

The output

Review: Level-1 Techniques

- **Equivalence class partitioning （ECP, 等价类划分）**
  - Partition the input domain of the parameter into equivalence classes
  - Adequacy criterion: cover each partition at least once
- **Boundary value analysis （BVA, 边界值分析）**
  - Analyze the boundary cases for each equivalence class in ECP.
  - Adequacy criterion: cover each boundary case at least once.
- **Cause-effect graph and decision table（因果图和决策表）**
  - Analyze the causal relation between input and output as *edges*.
  - Adequacy criterion: cover each edge at least once.

Level-2: Adequacy For Combination



Why Combination?

Many faults are triggered by interaction between input parameters of the unit under test.

Example:

```
if (altitude_adj == 0 ) {
      if (volume < 2.2)   {
         faulty code here!
      }else { good code, no problem}
} else {
      ......
}
```

Only test data that satisfies `altitude_adj==0` and `volume <2.2` can trigger the failure.

The Problem

- Level-1 techniques suggests a set of values for each individual parameter
  - Each cover one equivalence class or boundary case
- Now we have K parameters. Can we test all their value combinations?
  - Usually impossible as there are too many.
- This problem is ubiquitous.

Combination Problem: Case 1

- Unit under test is a Java method with 5 parameters:

  `foo( a, b, c, d, e )`

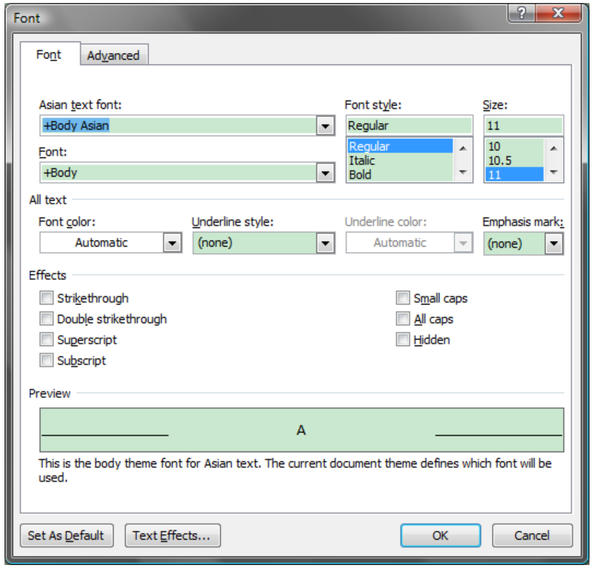
- Determine equivalence classes for each parameter, select a representative value from each equivalence class
  - For **a**, we might have: `[-∞, -4] [-3,-1] [0] [1,12] [13,+∞]`
  - For **b**, we might have: `[0,19] [20] [21,50] [51,+∞]`
  - ……
- To test the method, we need to select combinations of equivalence classes.
  - Suppose every parameter has 10 equivalence classes & boundary cases
  - Total combinations will be 5^{10} = 9765625 – too many to test them all.
- Test adequacy for parameter combination: Which combinations shall we cover?

Combination Problem: Case 2

- Unit under test is a feature or a sub-system
- Office font setting dialog parameters:
  - **font type**: Arial, `Courier`, Tahoma,...
  - **font style**: Regular, Italic, Bold
  - **font size**: 1, [5~10], [10, 20]
  - **font color**: red, green, blue
  - **underline style**: solid, dash, double-dash
  - **underline color**: red, green, blue
  - **emphasis mark**: line, dot
  - **superscript**: none, raise by 3, raise by 5, …
  - **spacing**: normal, expanded, condensed

- Total combinations will be at least: 3*3*17*3*3*3*2*3*3 = 74358

- Again, the total combinations are too many. The question is which combinations shall we cover?

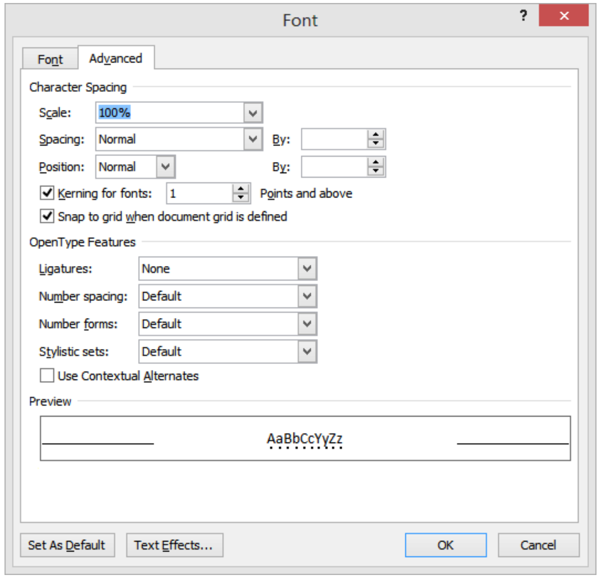


Asian text font: +Body Asian
Font style: Regular
Size: 11
Font: +Body
Regular
Italic
Bold
10
10.5
11

All text
Font color: Automatic
Underline style: (none)
Underline color: Automatic
Emphasis mark: (none)

Effects
☐ Strikethrough
☐ Double strikethrough
☐ Superscript
☐ Subscript
☐ Small caps
☐ All caps
☐ Hidden

Preview
A
This is the body theme font for Asian text. The current document theme defines which font will be used.



Character Spacing
Scale: 100%
Spacing: Normal By:
Position: Normal By:
☑ Kerning for fonts: 1 Points and above
☑ Snap to grid when document grid is defined

OpenType Features
Ligatures: None
Number spacing: Default
Number forms: Default
Stylistic sets: Default
☐ Use Contextual Alternates

Preview
AaBbCcYyZz

Combination Problem: Case 3

Unit under test is a Android application

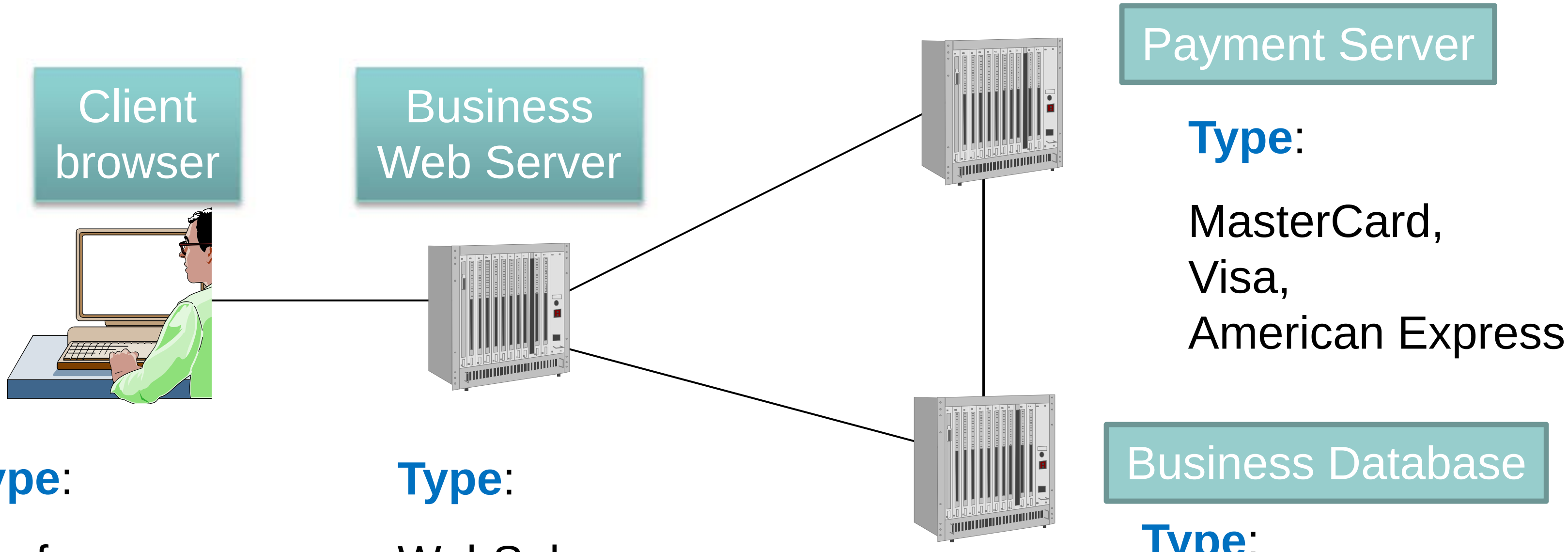
Parameter Name	Values	# Values
HARDKEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARD	12KEY, NOKEYS, QWERTY, UNDEFINED	4
NAVIGATIONHIDDEN	NO, UNDEFINED, YES	3
NAVIGATION	DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL	5
ORIENTATION	LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED	4
SCREENLAYOUT_LONG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

Total device configurations: 3 x 3 x 4 x 3 x 5 x 4 x 4 x 5 x 4 = 172,800

Which combinations shall we cover?

Combination Problem: Case 4

- Unit under test is a B2C system.
- Total system configuration = 4*3*3*3=36
- Which combinations shall we cover?



Type:

Firefox,
Internet Explorer,
Chrome,
Safari

Type:

WebSphere,
Apache,
.NET

Type:

MasterCard,
Visa,
American Express

Type:

DB/2,
Oracle,
MySQL

Combination Problem: Case 5

Unit under test is a radar system with on-off switches. Software must produce the right response for any combination of switch settings:

Total switch settings: $2^{34} = 1.7 \times 10^{10}$

Do you want to try testing all of them?

Level-2 Techniques

- **Combinatorial Coverage （组合覆盖）**
  - Adequacy criterion: cover each t-way interaction at least once
- **Constrained Combinatorial Coverage（带约束的组合覆盖）**
  - Adequacy criterion: combinatorial coverage subject to additional constraints between parameters

Combinatorial Coverage: Definition

- Let p be the number of parameters:
  - Parameters are indexed $1,\ldots, p$.
- For each parameter i, suppose that there are n_i concerned values
  - These values are from the n_i equivalence classes & boundary cases of this parameter.

- **t-way interactions** between parameters $i, i+1,\ldots i+t-1$: the $n_i * n_{i+1} * \ldots * n_{i+t-1}$ value combinations.
  - **Assumption**: parameters are **independent**. That is, the choice of values for any parameter does not affect the choice of values for any other parameter.
  - We will remove this restriction later in *constrained combinatorial coverage*.

- **t-wise (combinatorial) coverage**: cover all the t-way interactions between any set of *t* parameters.
  - Special case: pairwise (combinatorial) coverage -- covering all two-way interactions

Example

- Suppose that we have three parameters, each of which has two possible values.
  - A, B for parameter 1.
  - J, K for parameter 2.
  - Y, Z for parameter 3.
- There are $2^3 = 8$ possible full combinations.
- How about their two-way interaction?

A	J	Y
A	J	Z
A	K	Y
A	K	Z
B	J	Y
B	J	Z
B	K	Y
B	K	Z

Pairwise Coverage

- Cover the C(3, 2) = 12 two-way interactions.

Two-way interactions
between parameter 1 and 2

A	J	
A	K	
B	J	
B	K	

Two-way interactions
between parameter 1 and 3

A		Y
A		Z
B		Y
B		Z

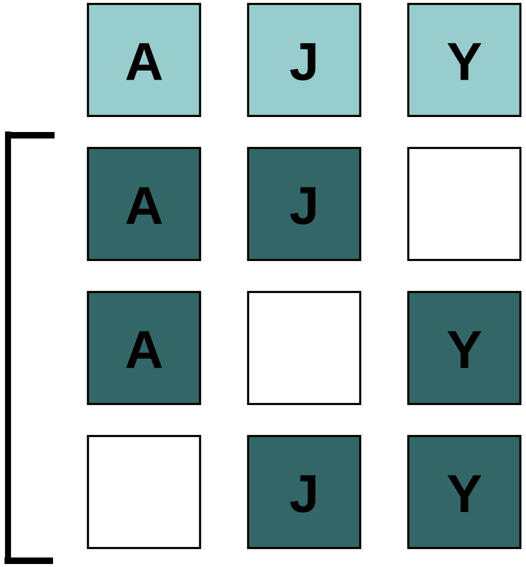
Two-way interactions
between parameter 2 and 3

	J	Y
	J	Z
	K	Y
	K	Z

Full combination vs. 2-way Interaction

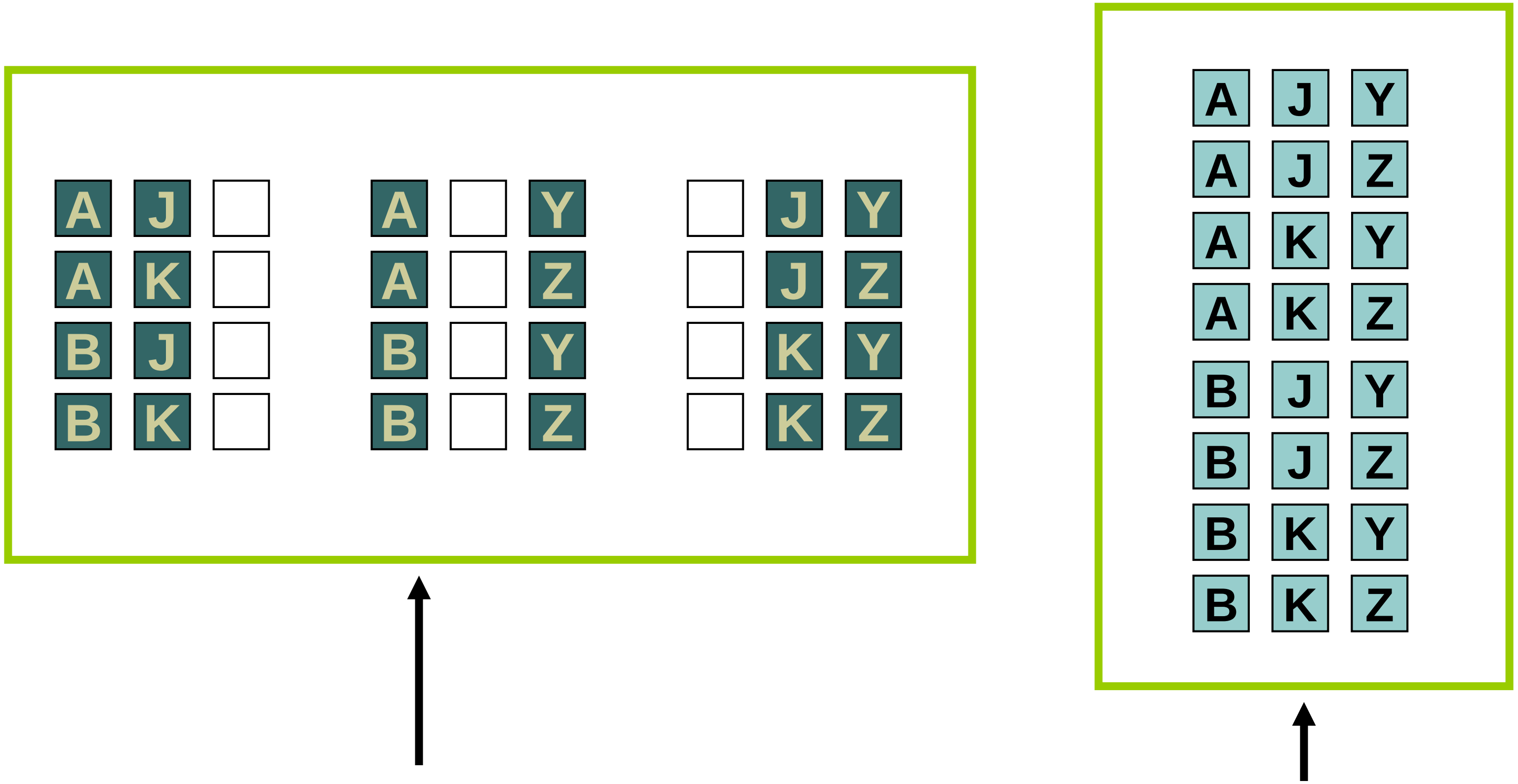
One full combination...

… covers 3 two-way interactions.



A	J	Y
A	J	
A		Y
	J	Y

Pairwise Coverage



A	J	
A	K	
B	J	
B	K	

A		Y
A		Z
B		Y
B		Z

	J	Y
	J	Z
	K	Y
	K	Z

A	J	Y
A	J	Z
A	K	Y
A	K	Z
B	J	Y
B	J	Z
B	K	Y
B	K	Z

Goal: cover all two-way interactions…

…using a subset of full combinations.

Why (only) Two-way Interaction?

- Fault analysis reveals that two-way interaction between parameters is a common source of failure in complex systems.

Meanwhile….

- The number of test cases required to cover all *t*-way interactions grows exponentially with *t*.

Number of combinations needed to cover all t-way interactions: $O(v^t \log n)$

for *v* values, *n* variables, *t*-way interactions

Evidence

- **Case 1**: Evaluation of FDA recall class failures in medical devices: 98% of the problems could have been detected by testing the device with all pairs of parameter settings.

  Reference: D. R. Wallace and D. R. Kuhn, 2001. The Effectiveness of Combinatorial Testing.

- **Case 2**: Mozilla web browser defect study: 75% of the faults can be exposed by covering the two-way interactions between parameters.

  Reference: Pairwise Testing: A Best Practice That Isn't: http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf

- **Case 3**: AT&T email system defect study: the 1000 test cases that satisfy pairwise coverage expose 20% more defects than the 1500 test cases that were designed originally in an ad hoc way.

Real World Example

- F-16 fighter failure caused by the interaction between the location of LANTRN pod and that of ventral fin.

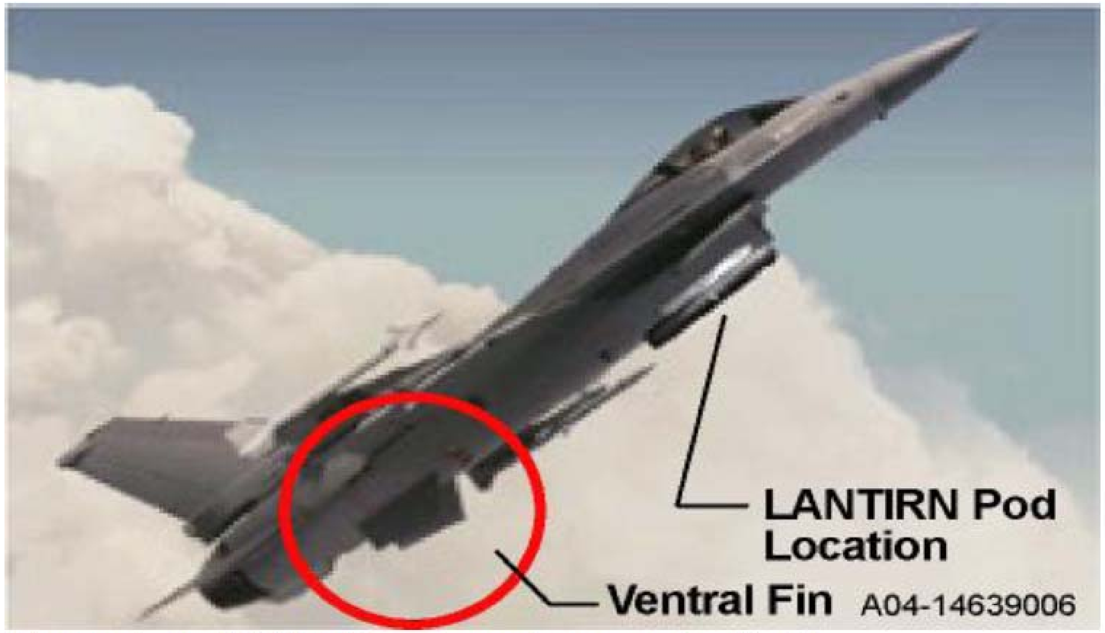


Figure 1. LANTIRN pod carriage on the F-16.

Information from a US AIR FORCE INSTITUTE OF TECHNOLOGY report: http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA506375

Selecting Test Cases for Pairwise Coverage

Covered two-way Interactions

A	J	
A	K	
B	J	
B	K	

A		Y
A		Z
B		Y
B		Z

	J	Y
	J	Z
	K	Y
	K	Z

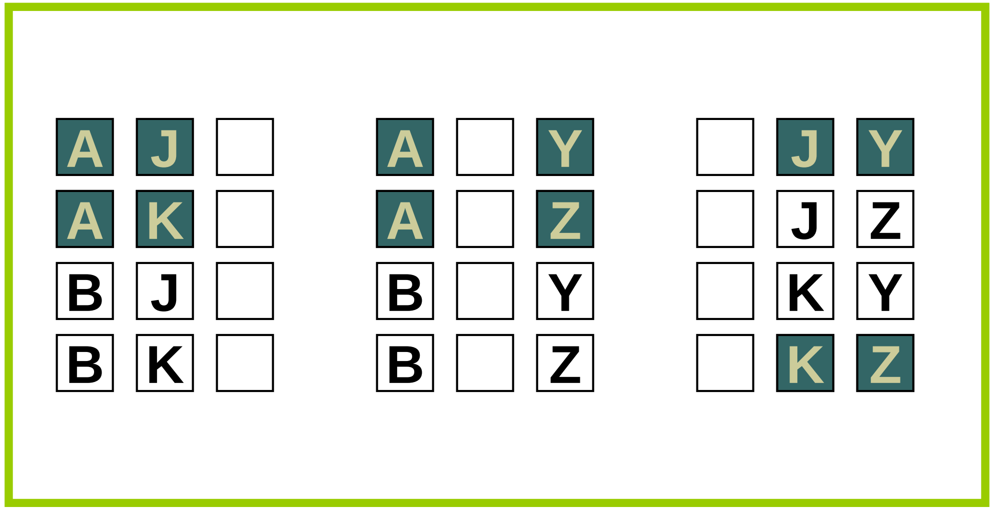
coverage: 3 / 12 = 25%

Full combinations (test cases)

A	J	Y
A	J	Z
A	K	Y
A	K	Z
B	J	Y
B	J	Z
B	K	Y
B	K	Z

Selecting Test Cases for Pairwise Coverage

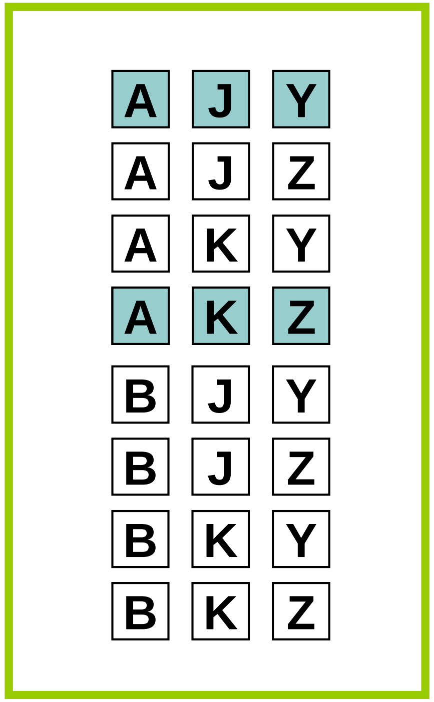
Covered two-way Interactions



A	J		A		Y		J	Y
A	K		A		Z		J	Z
B	J		B		Y		K	Y
B	K		B		Z		K	Z

coverage: 6 / 12 = 50%

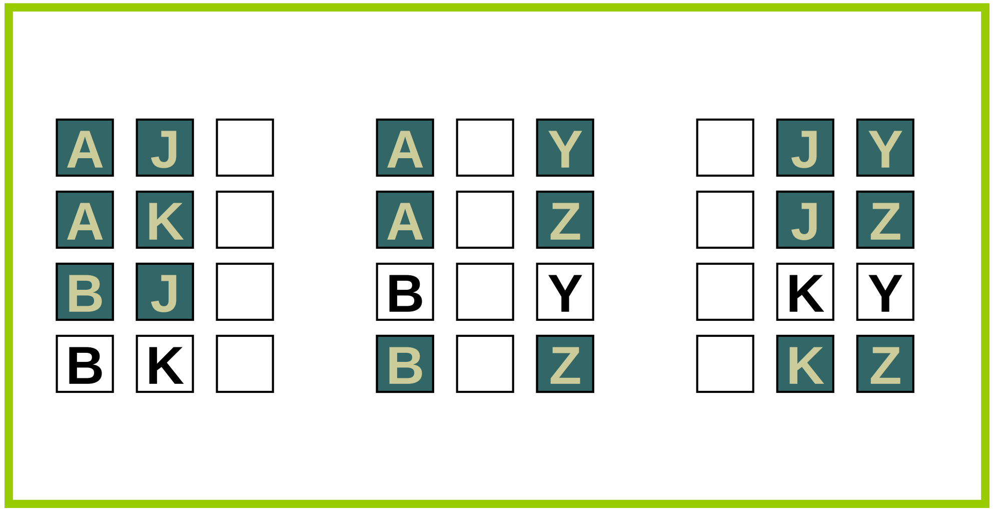
Full combinations (test cases)



A	J	Y
A	J	Z
A	K	Y
A	K	Z
B	J	Y
B	J	Z
B	K	Y
B	K	Z

Selecting Test Cases for Pairwise Coverage

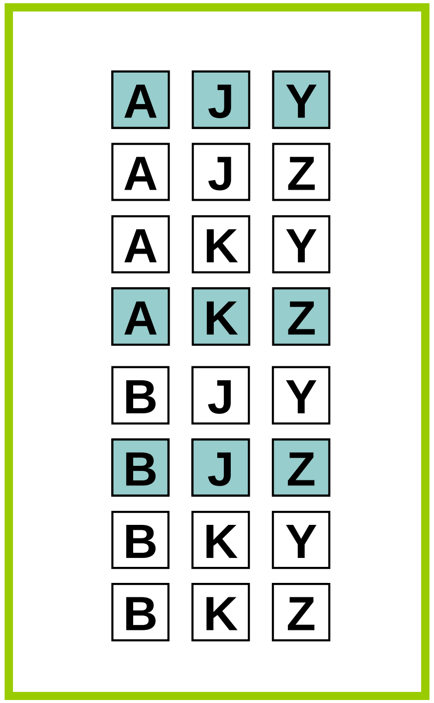
Covered two-ways Interactions



A	J		A		Y		J	Y
A	K		A		Z		J	Z
B	J		B		Y		K	Y
B	K		B		Z		K	Z

coverage: 9 / 12 = 75%

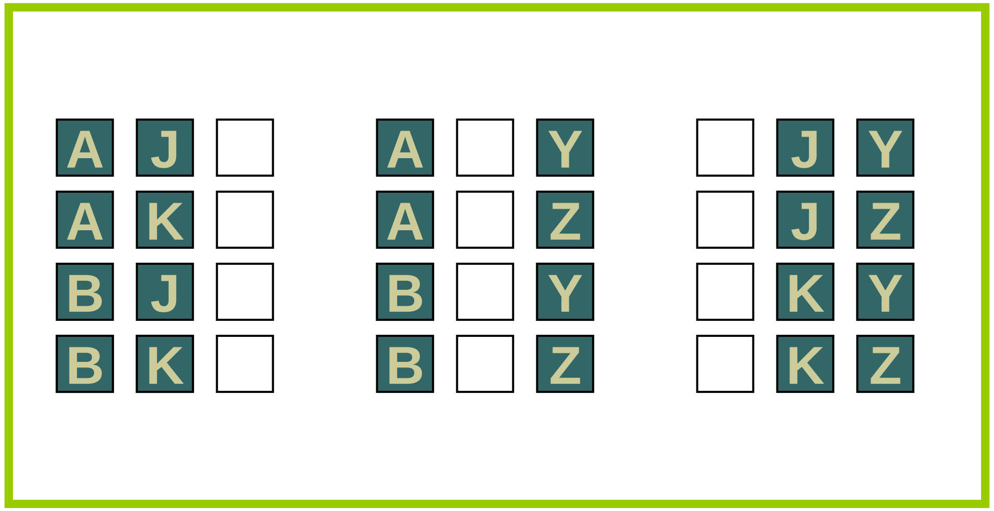
Full combinations
(test cases)



A	J	Y
A	J	Z
A	K	Y
A	K	Z
B	J	Y
B	J	Z
B	K	Y
B	K	Z

Selecting Test Cases for Pairwise Coverage

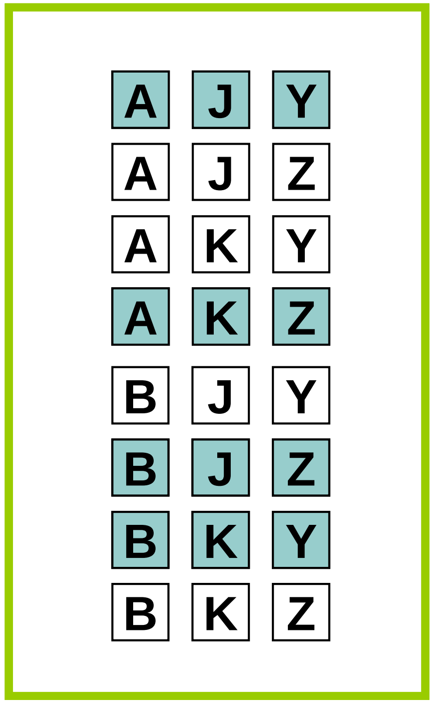
Covered two-ways Interactions



A	J		A		Y		J	Y
A	K		A		Z		J	Z
B	J		B		Y		K	Y
B	K		B		Z		K	Z

coverage: 12 / 12 = 100%

Full combinations
(test cases)



A	**J**	**Y**
A	J	Z
A	K	Y
A	**K**	**Z**
B	J	Y
B	**J**	**Z**
B	**K**	**Y**
B	K	Z

Finding pairwise adequate test cases

Question: how to cover all two-way interactions with the fewest number of test cases?

Transform the Problem

			AJY	AJZ	AKY	AKZ	BJY	BJZ	BKY	BKZ	
A	J		x_1	$+ x_2$							≥ 1
A		Y	x_1		$+ x_3$						≥ 1
	J	Y	x_1				$+ x_5$				≥ 1
A	K				x_3	$+ x_4$					≥ 1
A		Z		x_2		$+ x_4$					≥ 1
	J	Z		x_2				$+ x_6$			≥ 1
B	J						x_5	$+ x_6$			≥ 1
B		Y					x_5		$+ x_7$		≥ 1
	K	Y			x_3				$+ x_7$		≥ 1
B	K								x_7	$+ x_8$	≥ 1
B		Z						x_6		$+ x_8$	≥ 1
	K	Z				x_4				$+ x_8$	≥ 1

Minimize: $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8;$ $x_i \in \{0,1\}$

Classic {0,1}-integer programming

- Solution of {0,1}-integer programming is an NP-complete problem.
- But there are still plenty of (fast enough) tools to address it (e.g. the NEOS online solver: http://www.neos-server.org/neos/ or the bintprog function in Matlab)
- A little experiment for extra solutions:

# Params	# Values per param	# Constraints	# Interactions	Result: # combinations	Run time (s)
3	2	12	8	4	<0.01
4	2	24	16	5	0.01
5	2	40	32	6	0.70
6	2	60	64	6	16.57
7	2	84	128	6	441.21
4	3	54	81	9	0.08
5	3	90	243	*	*

* process killed after running for 6.5 hours

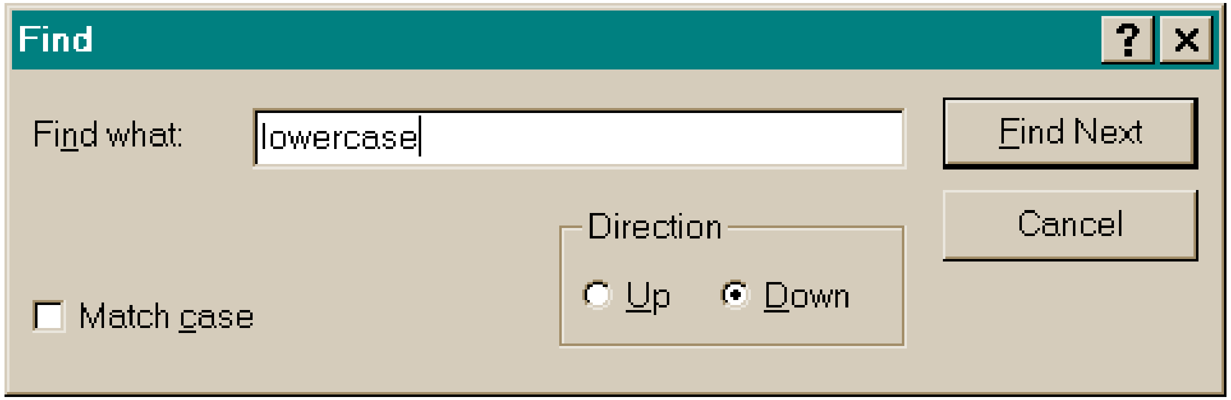
Besides, one or two extra test cases will not kill us – A greedy algorithm is usually good enough.

Greedy Algorithm

- **Input**: N parameters x_1, x_2, …, x_N, with v_1, v_2, … v_N values, respectively.
- **Output**: K combinations of the values of x_1, x_2, …, x_N that achieve two-ways interactions

- Initialize the result set as empty
- For each parameter x_i (i in [0, N-1]),
  - For each of the v_i*v_{i+1} interactions between the v_i values of x_i and v_{i+1} values of x_{i+1},
    - If the interaction has been covered by one of the combinations in the result set, ignore it.
    - Otherwise, create a combination that cover this interaction and the most amount of interactions not yet covered.
- Return the result set.

Quiz 1: Pairwise Coverage



Find what: lowercase

□ Match case

Direction: ○ Up ● Down

- Test a simple Find dialog. It takes three inputs:
  - `Find what:` a text string.
  - `Match case:` yes or no
  - `Direction:` up or down
- For the text string, consider three values: “lowercase” and “Mixed Cases” and “CAPITALS”.
- Total combinations: 2*2*3=12

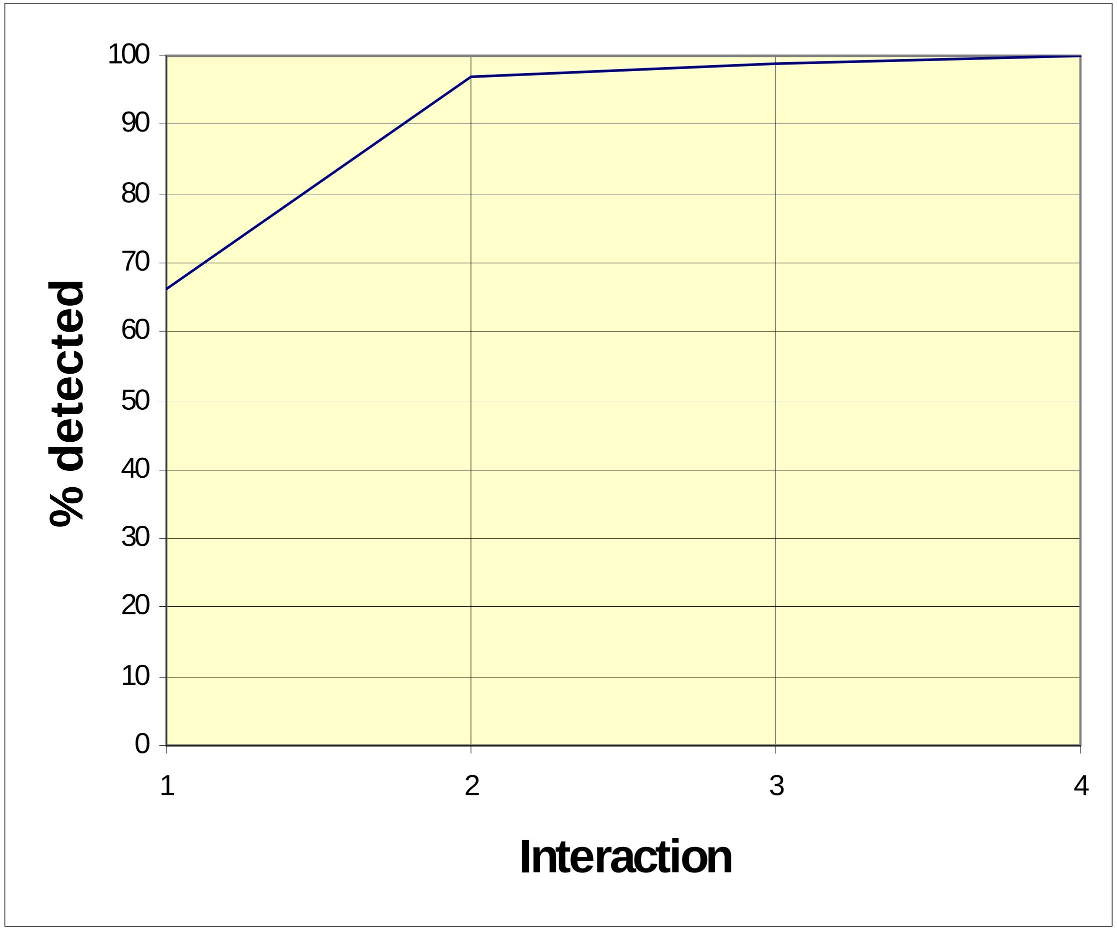
- What are the combinations that satisfy pairwise coverage?
  - LYD, MYU, CYD, LNU, MND, CNU

From Pairwise to T-wise Coverage

- The only obstacle is the *cost.*
    - Recall that the number of test cases needed to cover all *t*-way interactions is $O(v^t \log n)$ for *v* values, *n* variables
    - However, depends on the actual system, the growth on *t* might not be as high as exponential.
- The key issue is whether the benefit justifies the cost.

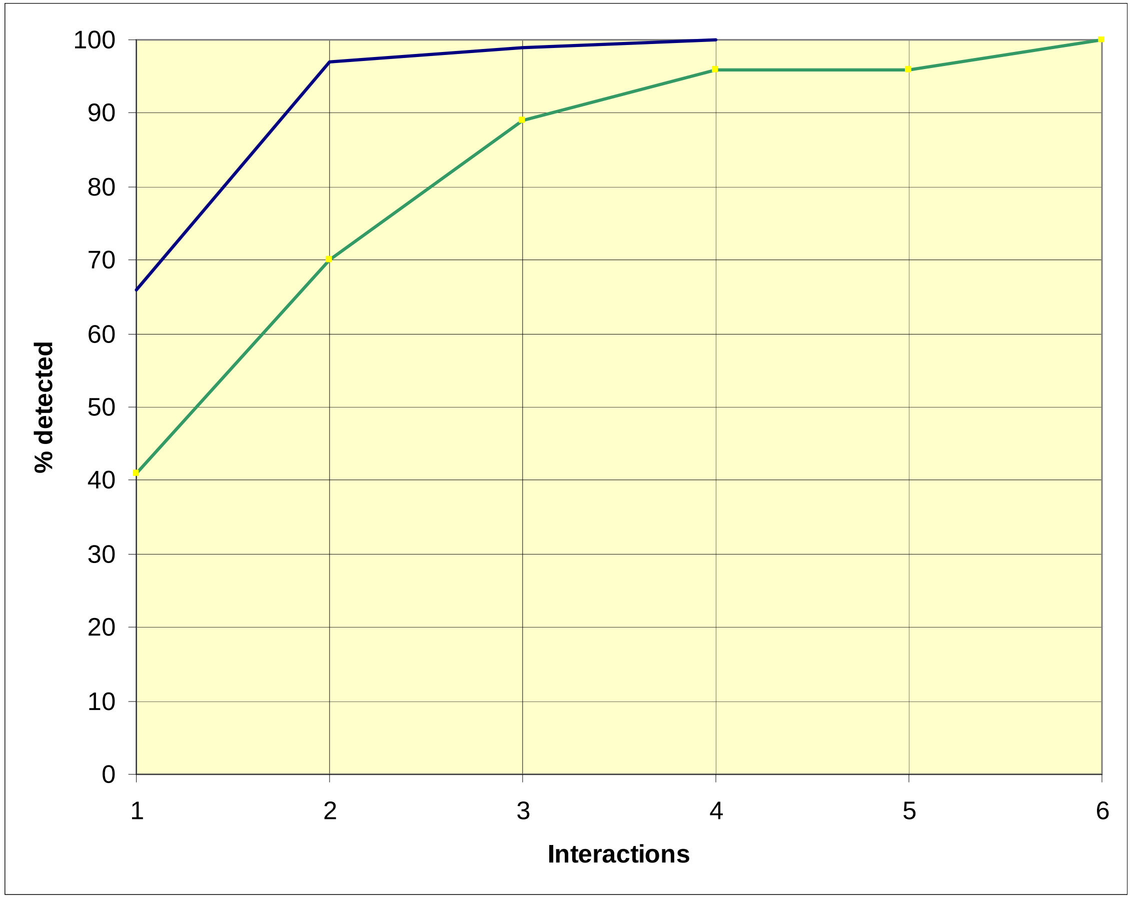
Empirical Study by NIST National Institute of Standards and Technology

Medical device



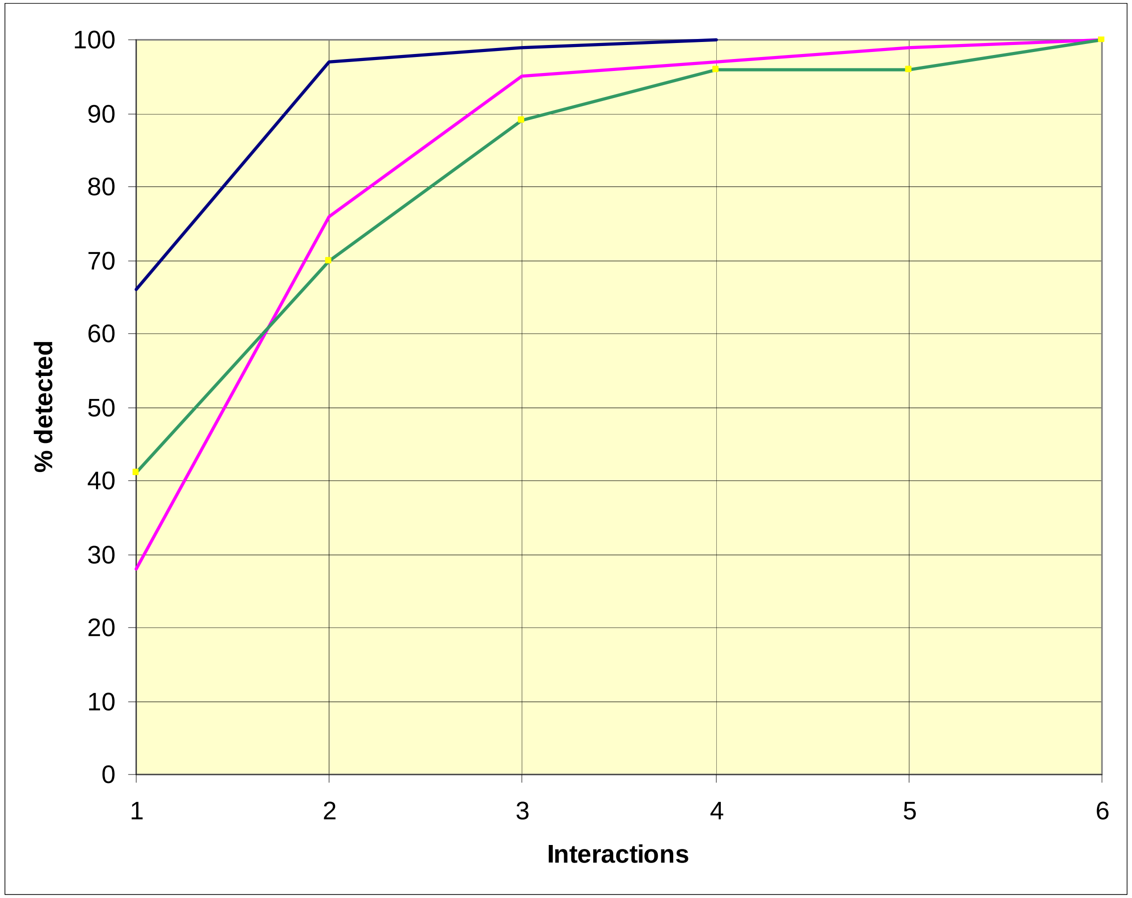
Empirical Study by NIST National Institute of Standards and Technology

Server (green)



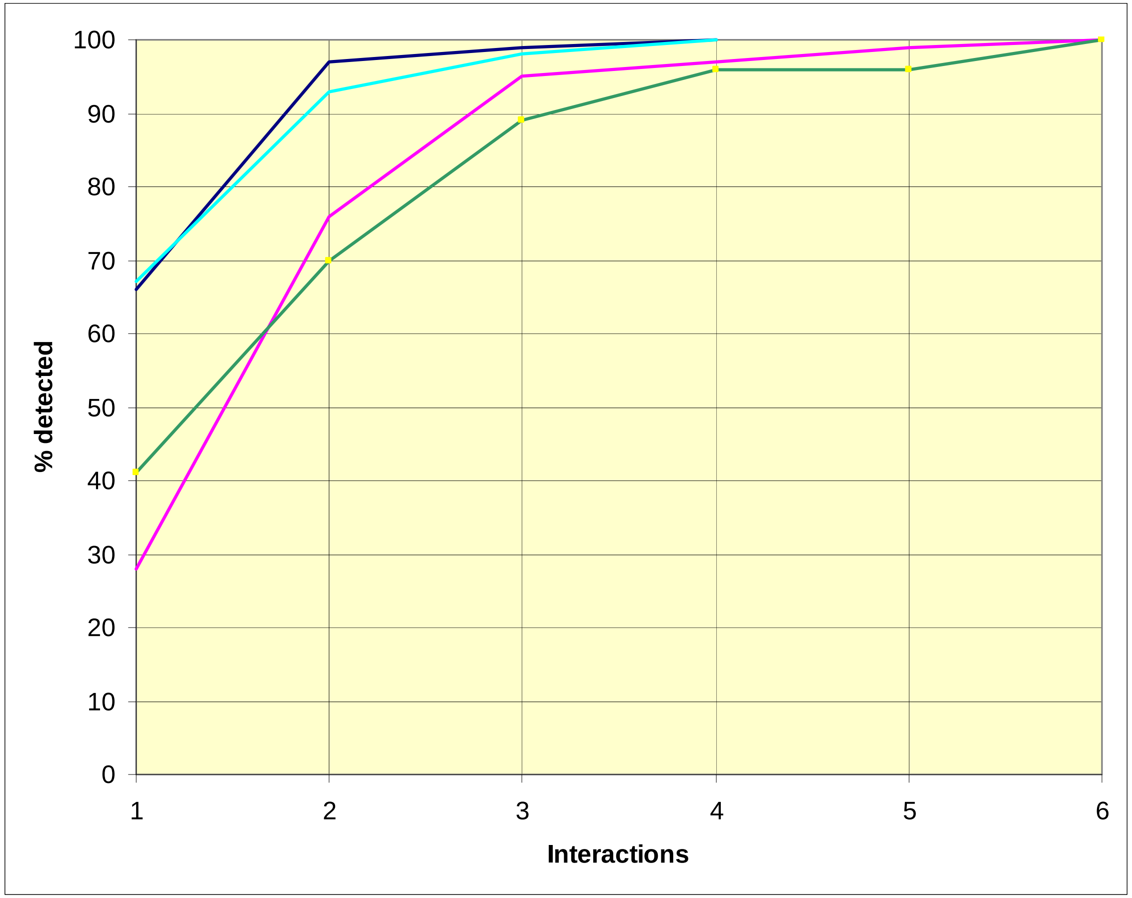
Empirical Study by NIST National Institute of Standards and Technology

Browser (magenta)



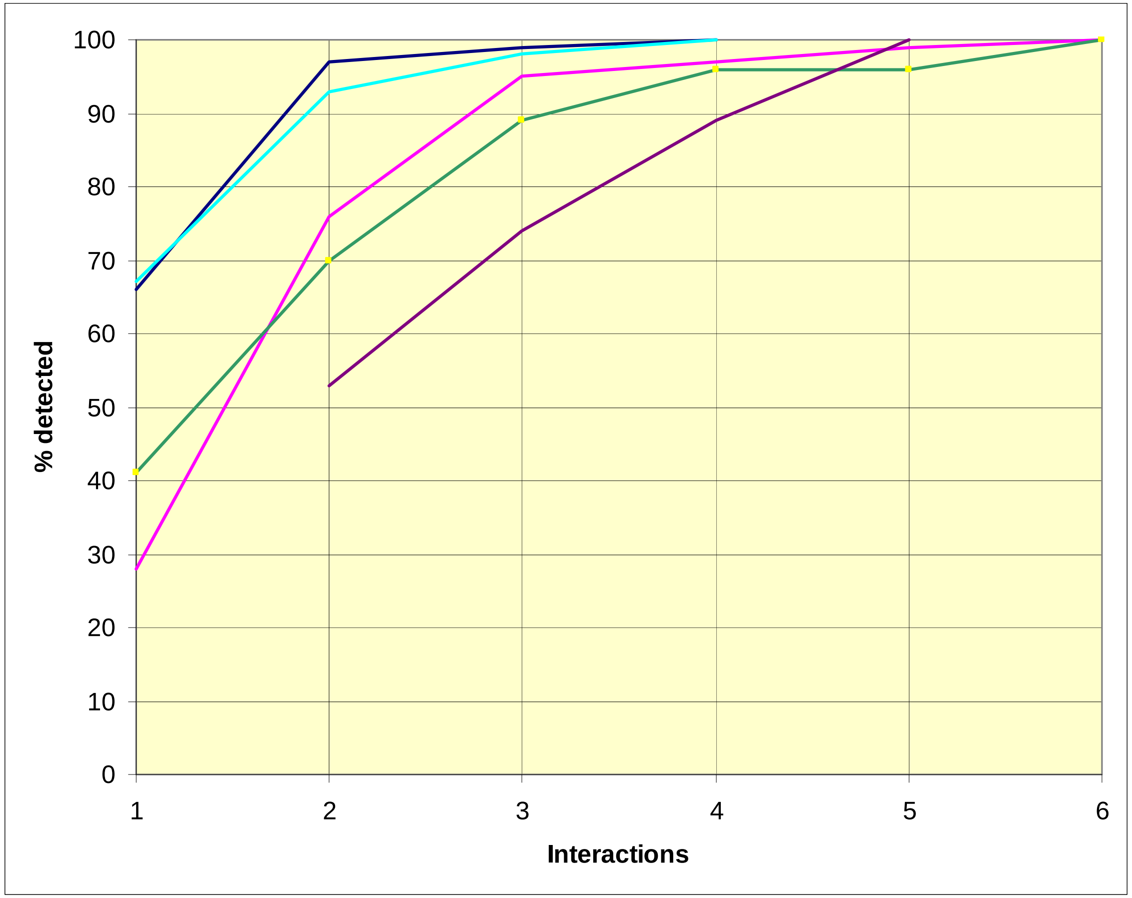
Empirical Study by NIST National Institute of Standards and Technology

NASA Goddard distributed database (light blue)



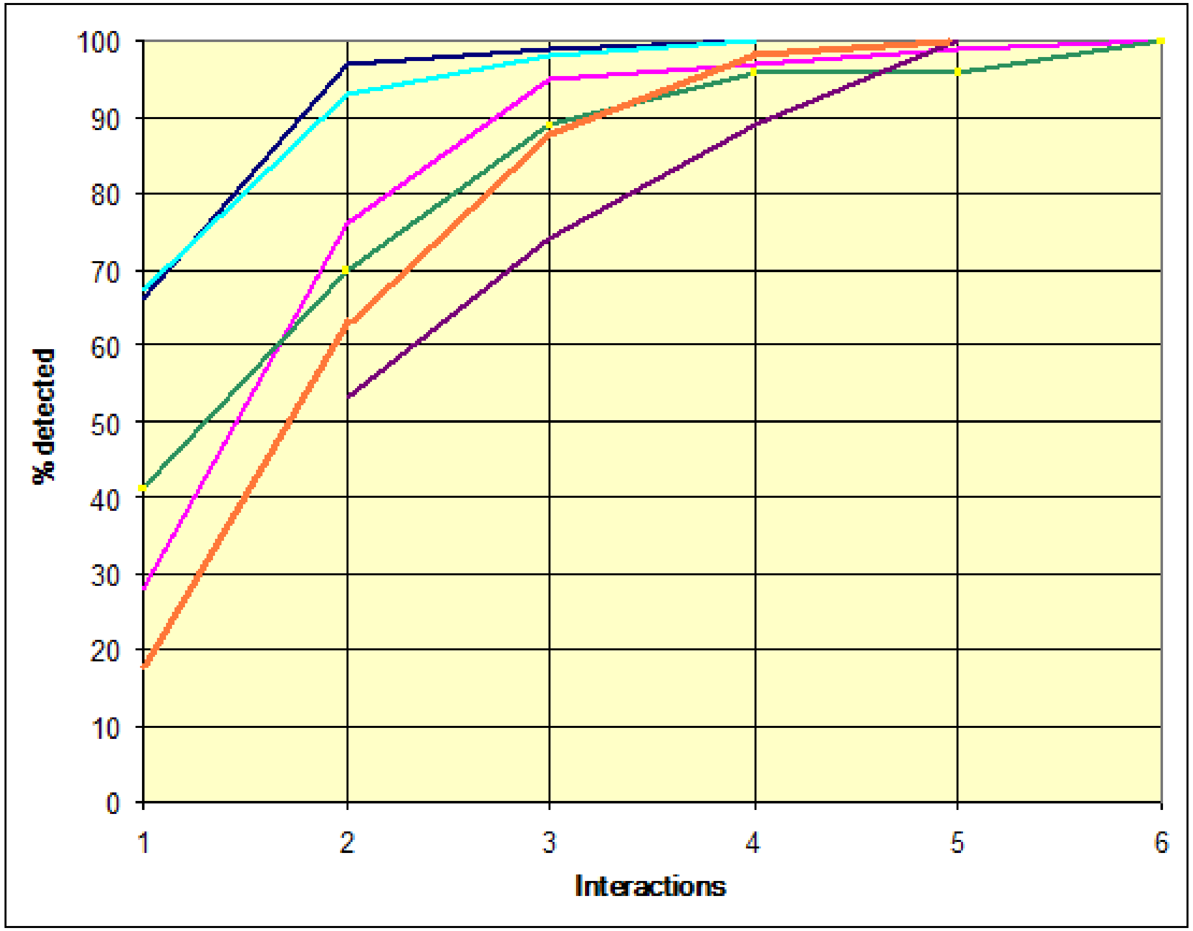
Empirical Study by NIST National Institute of Standards and Technology

FAA Traffic Collision Avoidance System module (purple)



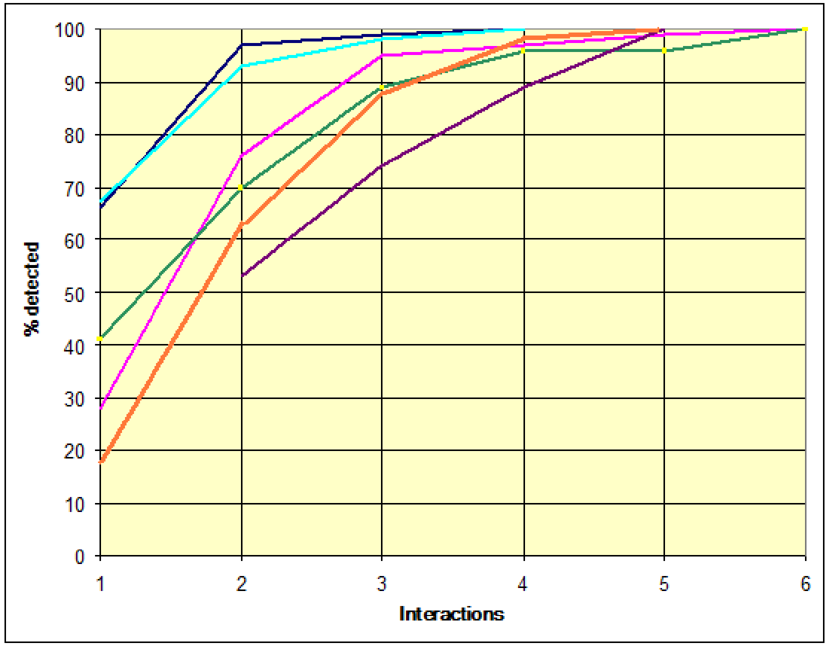
Empirical Study by NIST National Institute of Standards and Technology

Network security (Bell, 2006) (orange)



So what?

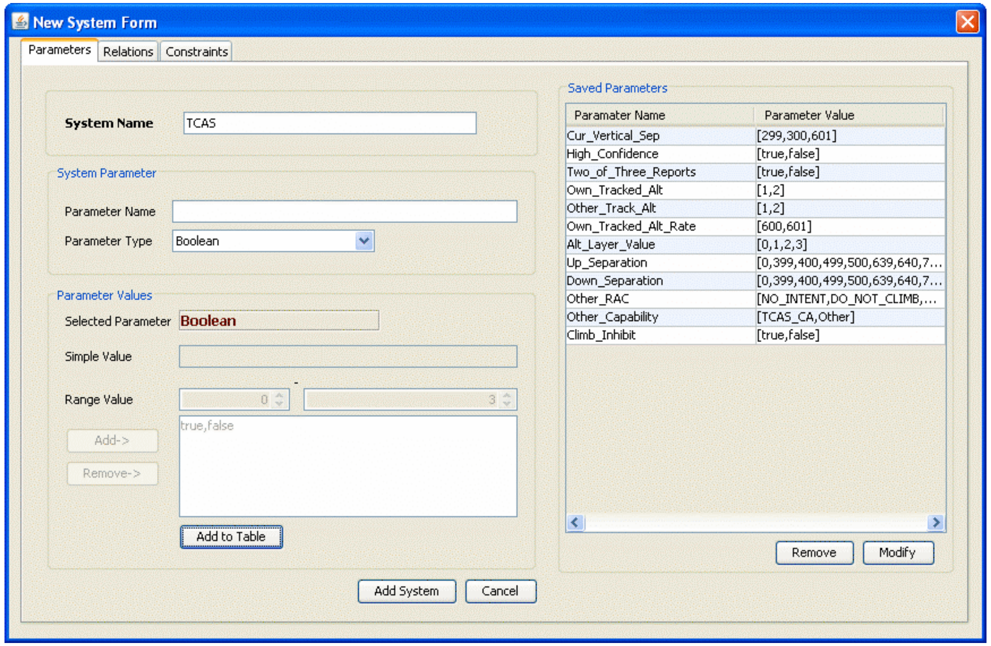
- As suggested from 《How we test software at Microsoft》:
  - Pairwise coverage as the base line.
  - Time allowed, improve the test suite to 3~6-wise combinatorial coverage.
- It is generally agreed that there is little benefit beyond 6-wise combinatorial coverage.



Tool 1: ACTS

http://csrc.nist.gov/groups/SNS/acts/

- Developed by NIST. Free for academic use. But need to write an email to ask for the software. (The course website provides a link to a local copy)
- Support 2~6-way coverage (called ‘strength of interaction’)
- Allow for mixed strength: e.g. cover 2-way for all, 3-way for a selected subset of parameters.



System Name: TCAS

Parameter Name:

Parameter Type: Boolean

Selected Parameter: Boolean

Simple Value:

Range Value: 0 - 3

true,false

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

Tool 2: PICT from Microsoft

http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi

- Command line tool to generate T-way coverage test suite.
  - If T=2 (default) then it is pairwise coverage.
- Example input: test.txt

```
Type:           Primary, Logical, Single, Span, Stripe, Mirror, RAID-5
Size:           10, 100, 500, 1000, 5000, 10000, 40000
Format:         quick, slow
File system:   FAT, FAT32, NTFS
Cluster size:  512, 1024, 2048, 4096, 8192, 16384, 32768, 65536
Compression:  on, off
```

Also support mixed strength with ***Sub-Models***

- Output of “pict /o:2 test.txt”

```
Type        Size        Format      File system  Cluster size  Compression
Primary     40000       quick       NTFS          1024         off
Span        100         slow        FAT32         16384        on
RAID-5      500         quick       FAT           4096         on
Primary     5000        slow        FAT32         4096         off
RAID-5      1000        slow        FAT           32K          on
Stripe      500         slow        NTFS          2048         on
RAID-5      10          quick       FAT32         2048         on
Primary     10          slow        FAT           1024         on
Logical     100         quick       NTFS          32768        on
Primary     5000        quick       NTFS          512          off
Span        500         slow        FAT32         32768        off
Mirror      5000        slow        NTFS          64096        on
RAID-5      100         slow        NTFS          4096         off
RAID-5      10          slow        FAT           512          on
Logical     1000        quick       FAT32         8192         on
Logical     100         quick       NTFS          4096         off
Single      10000       quick       FAT           65536        off
```

Tool 3: Category-Partition Testing

- Proposed by Ostrand and Balcer in a 1988 CACM paper.
  - The input language is called TSL (Test Specification Language). We will use TSL to refer to category-partition testing and the tool.
  - The resource page on our SE-307 course website provides the source code of the tool.
  - We will use this tool in assignment #3.
- Why not ATCS or PICT in assignment #3? Because:
  - 1) we have the source code of the tool, so that you might modify it for your own tasks in the future (the importance of open source).
  - 2) it supported more sophisticated constraints on the parameters, as we will see later.
- One problem, though, is that the tool only generates **all** feasible combinations. It is up to you to select a subset to achieve t-wise combinatorial coverage.

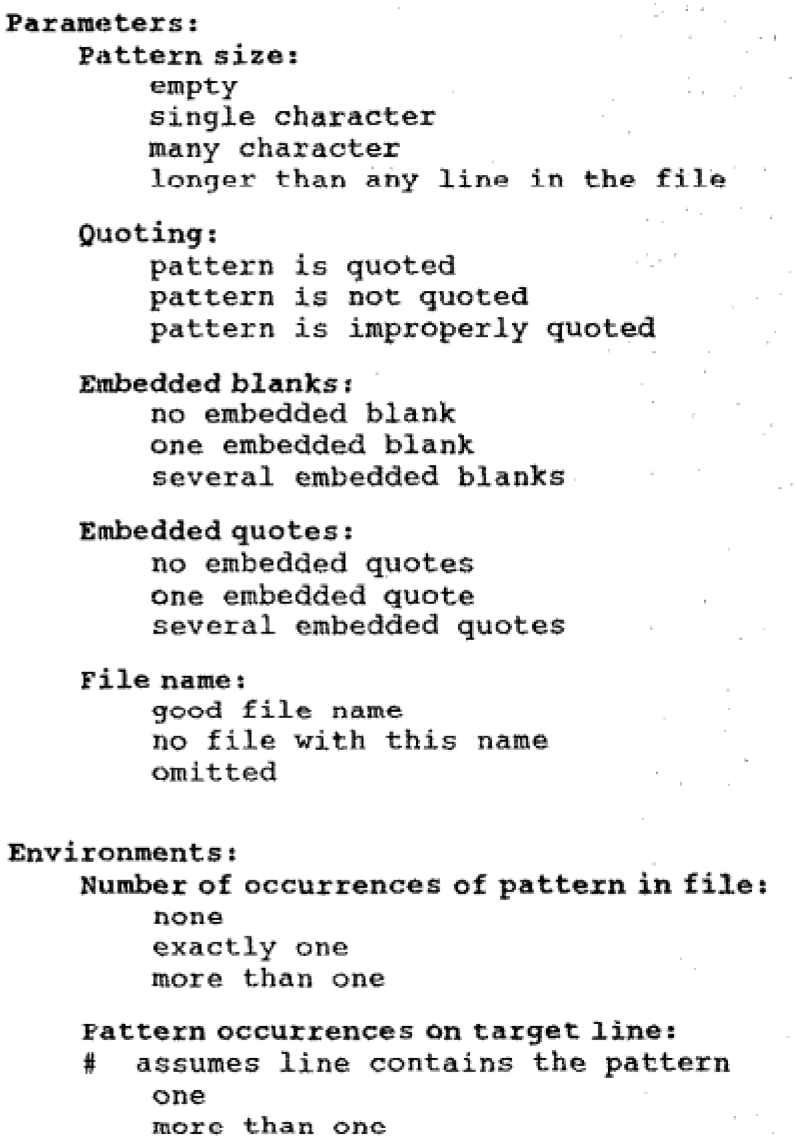
TSL in a nutshell

- Parameters are called **categories** in TSL, while the values of parameter are grouped into **choices**.
  - The original paper discuss two kinds of categories: *parameters* and *environments*. They are handled in the exactly same way in the tool.
  - Choices are strings that describe equivalence classes & boundary cases of the parameters.
    - This is different from ACTS and PICT. They use concrete values. This is not an essential difference, though.

Example: “find” command in Linux

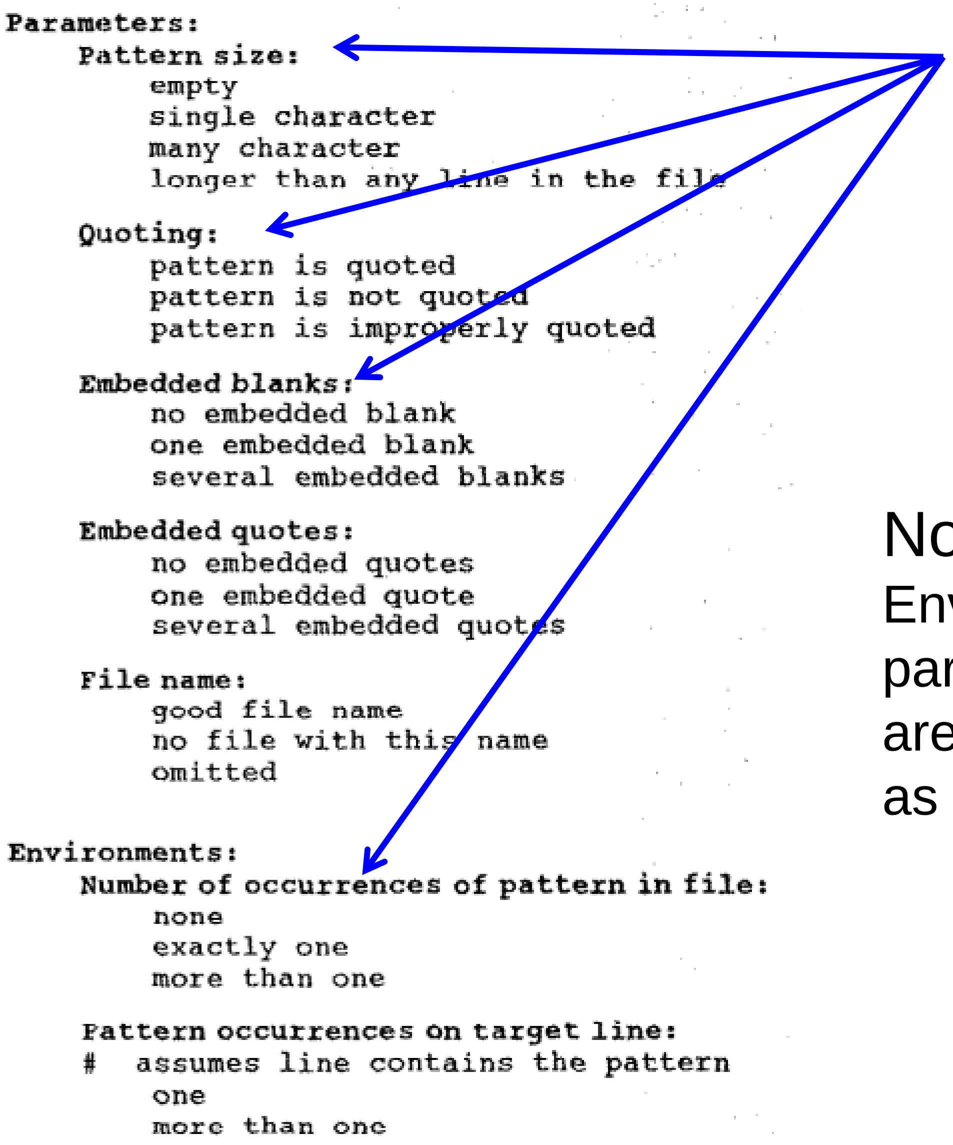
- Syntax: `find <Pattern> <file>`
- Function:
  - Locate one or more instances of a given pattern in a text file. All lines in the file that contains the pattern are printed. A line containing the pattern is written only once, regardless of the number of occurrence.
- Pattern:
  - The pattern is any string whose length does not exceed the maximum length of a line in the file. It can be quoted or unquoted. To include a blank in the pattern, the entire pattern must be enclosed in quotes “”. To include the quotation mark in the pattern, two consecutive quotes must be used.
- Examples:
  - `find john myfile` displays lines that contains `john`
  - `find "john smith" myfile` displays lines that contains `john smith`
  - `find "john"" smith" myfile` displays lines that contain `john" smith`

TSL for **find**



```
Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one
        more than one
```

TSL for **find**



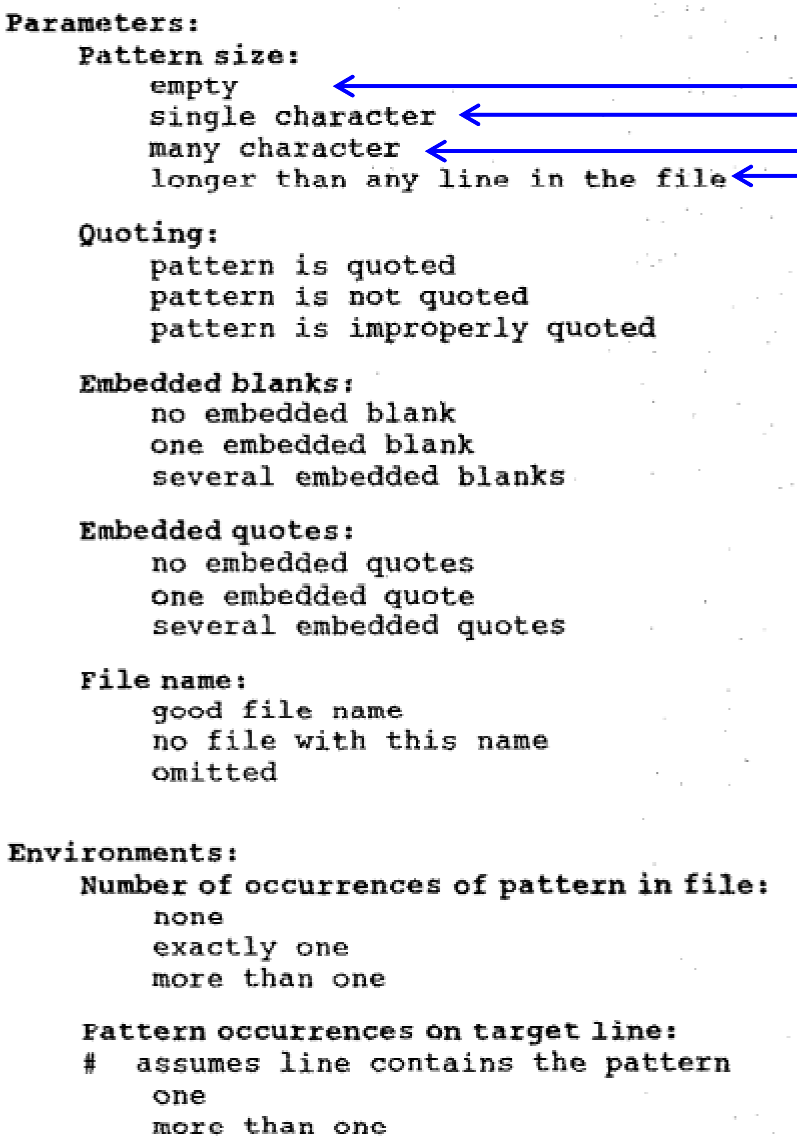
```
Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one
        more than one
```

parameter **(category)**

Notes:

Environments describe “indirect” parameter. In the tool they are handled in exactly the same way as the “direct” parameter.

TSL for **find**



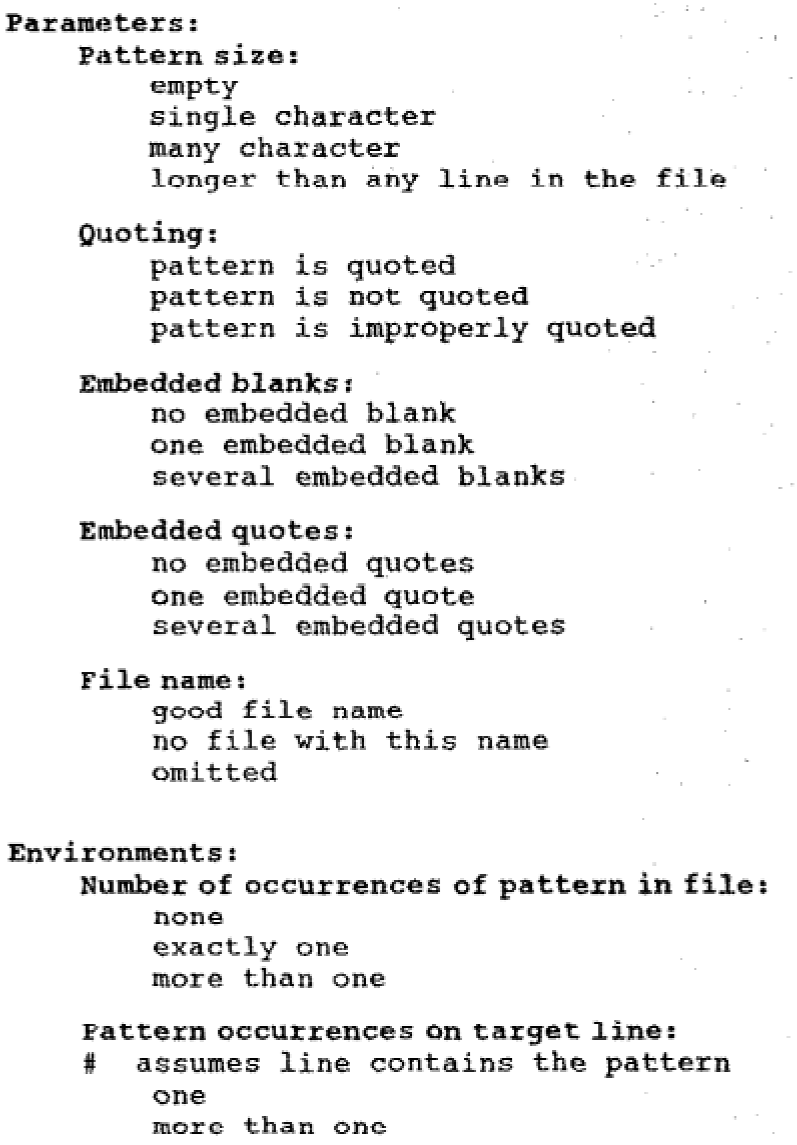
```
Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #   assumes line contains the pattern
        one
        more than one
```

equivalence classes and boundary cases of the parameter **(choices)**

Notes:
ACTS and PICT use *actual values*, while TSL use *classes*.

This is a fundamental difference: by using classes as choices for parameter, TSL allow us to describe the real parameter with multiple “virtual parameters”.

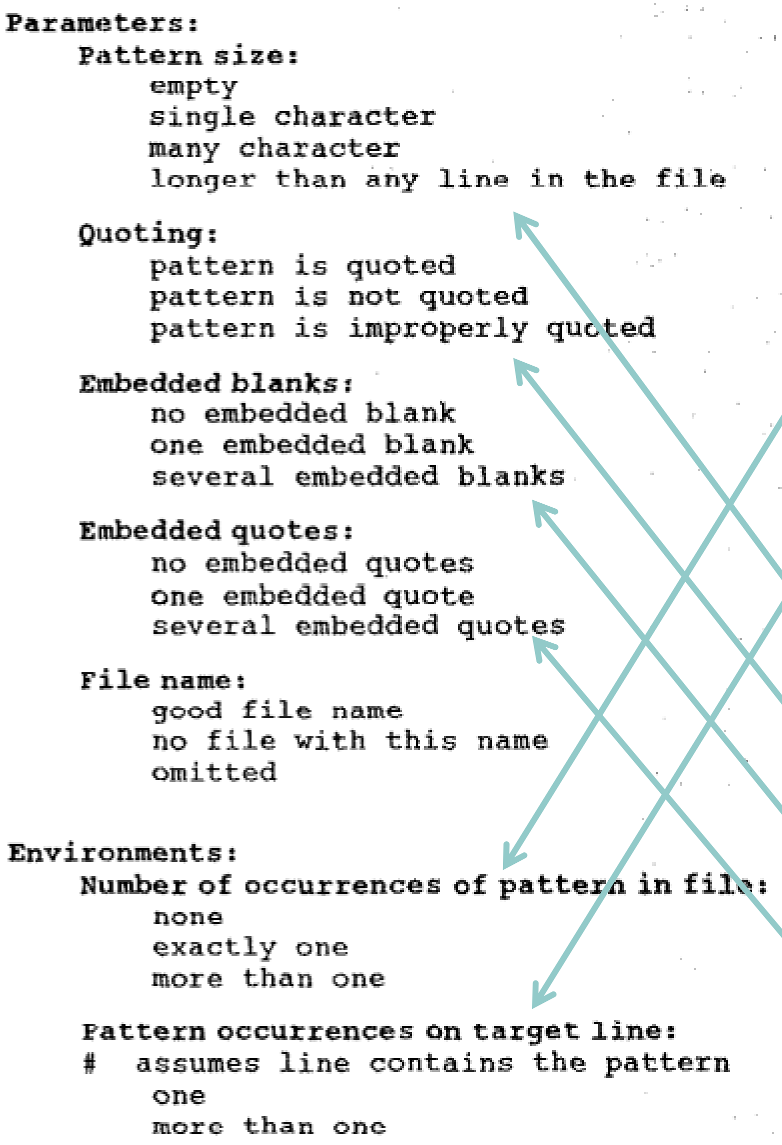
TSL for **find**



```
Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #   assumes line contains the pattern
        one
        more than one
```

- **Note**: a complex parameter can be described by multiple "virtual parameters".
  - The real parameter "pattern" is described by "pattern size", "quoting", etc.
  - The real parameter "file" is described by "filename", "number of pattern occurrence in the file", etc.
- Another possible way to handle complex parameter: use cause-effect graph to generate equivalence classes:
  - Try it yourselves and think about the pros and cons of each method.

Where does TSL come from?



```
Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one
        more than one
```

From the requirement

Function:

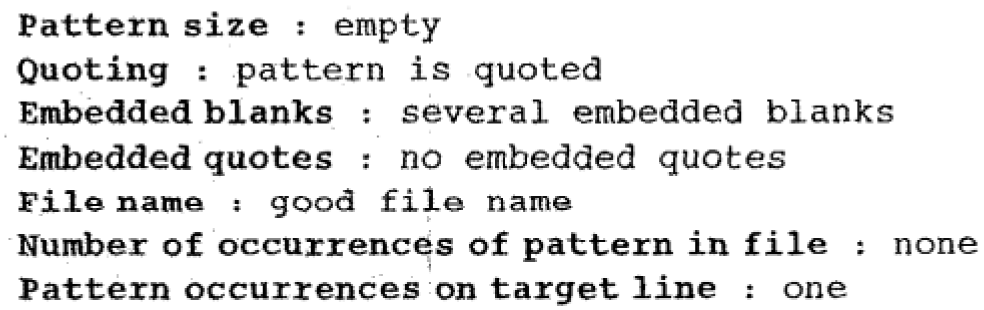
Locate one or more instances of a given pattern in a text file. All lines in the file that contains the pattern are printed. A line containing the pattern is written only once, regardless of the number of occurrence.

Pattern:

The pattern is any string whose length does not exceed the maximum length of a line in the file. It can be quoted or unquoted. To include a blank in the pattern, the entire pattern must be enclosed in quotes “”. To include the quotation mark in the pattern, two consecutive quotes must be used.

Running the tool on TSL

- Generated all feasible combinations of choices. Each combination is called a “test frame”
- A test frame for `find`:



```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

- Test frame is not test case. You need another tool to generate a test case from a test frame.
- In the original category-partition method, it is done manually as it is domain-specific.

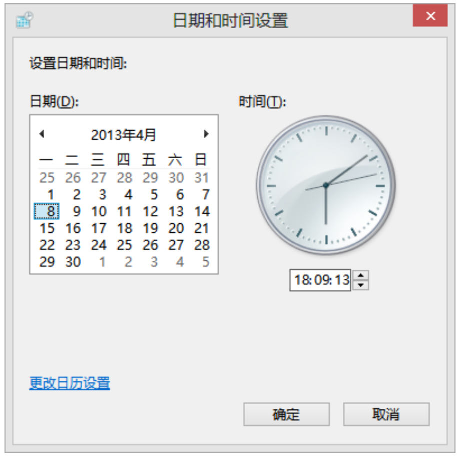
Level-2 Techniques

- **Combinatorial Coverage （组合覆盖）**
  - Adequacy criterion: cover each t-way interaction at least once
- **Constrained Combinatorial Coverage（带约束的组合覆盖）**
  - Adequacy criterion: combinatorial coverage subject to additional constraints between parameters

Why Constraints?

- **Reason 1**: rule out *infeasible* (不可能) combinations
  - They are impossible to input through user interface.
  - Need to differentiate *infeasible* (不可能) from *invalid* (非法).
    - e.g. the combination year=2013, month=2, day=30.

infeasible



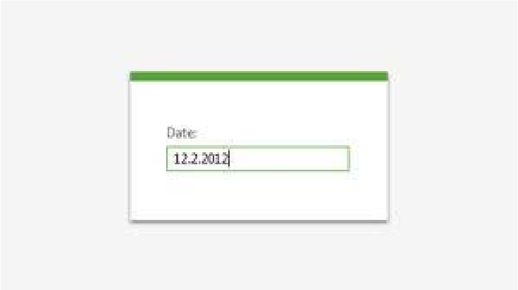
2013年4月

一	二	三	四	五	六	日
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

18:09:13

vs.

invalid



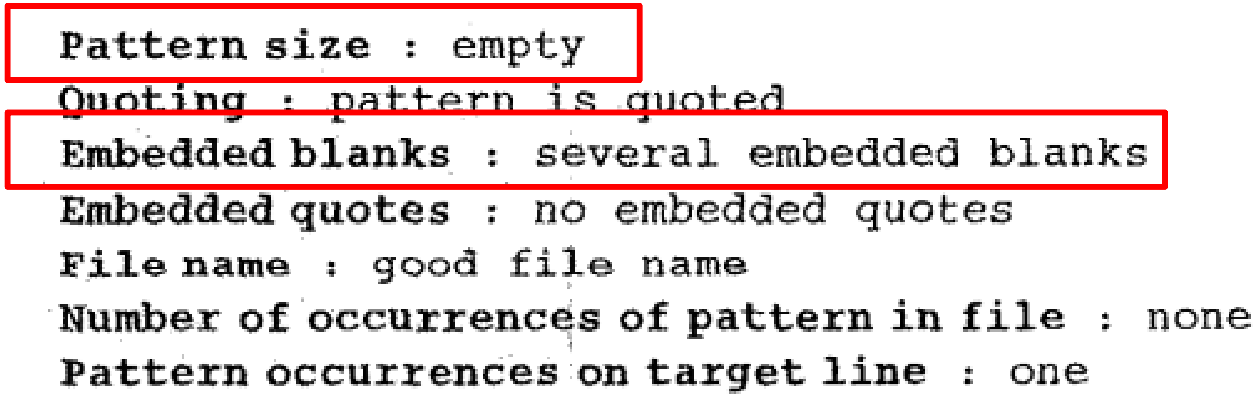
Date: 12.2.2012

- Rule out *infeasible combinations*, not *invalid combinations*.
  - On the contrary, we shall deliberately insert invalid combination into the pairwise-coverage test suite, it they have not been covered.

Why Constraints?

- **Reason 1**: rule out *infeasible* (不可能) combinations
  - Especially important for TSL.

Consider this test frame:



Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one

Why Constraints?

- **Reason 2**: rule out *redundant* (冗余) combinations
  - Some values of one parameter might trigger the system to bypass whatever values the other parameters take.

Example:

```
public static void main (String args[]) {
   if (args.length > 2) {
      Quit.now("Usage: java STS attributes.txt [s|st|ts|t]");
   }
   ...
```

We only need one parameter combination with args.length>2. Others are all redundant, as they exercise exactly the same behavior of the program.

- Constraints can be used to rule out redundant combinations and reduce the size of test suite.

Introducing Constraints

- Be careful!
  - Some combinations might NOT be really infeasible or redundant, as you believe.
  - Suggestion: think twice before introducing constraints.
- Types of constraints:
  - Property constraints: rule out infeasible combinations
  - Error/single constraints: rule out redundant combinations

Property Constraints

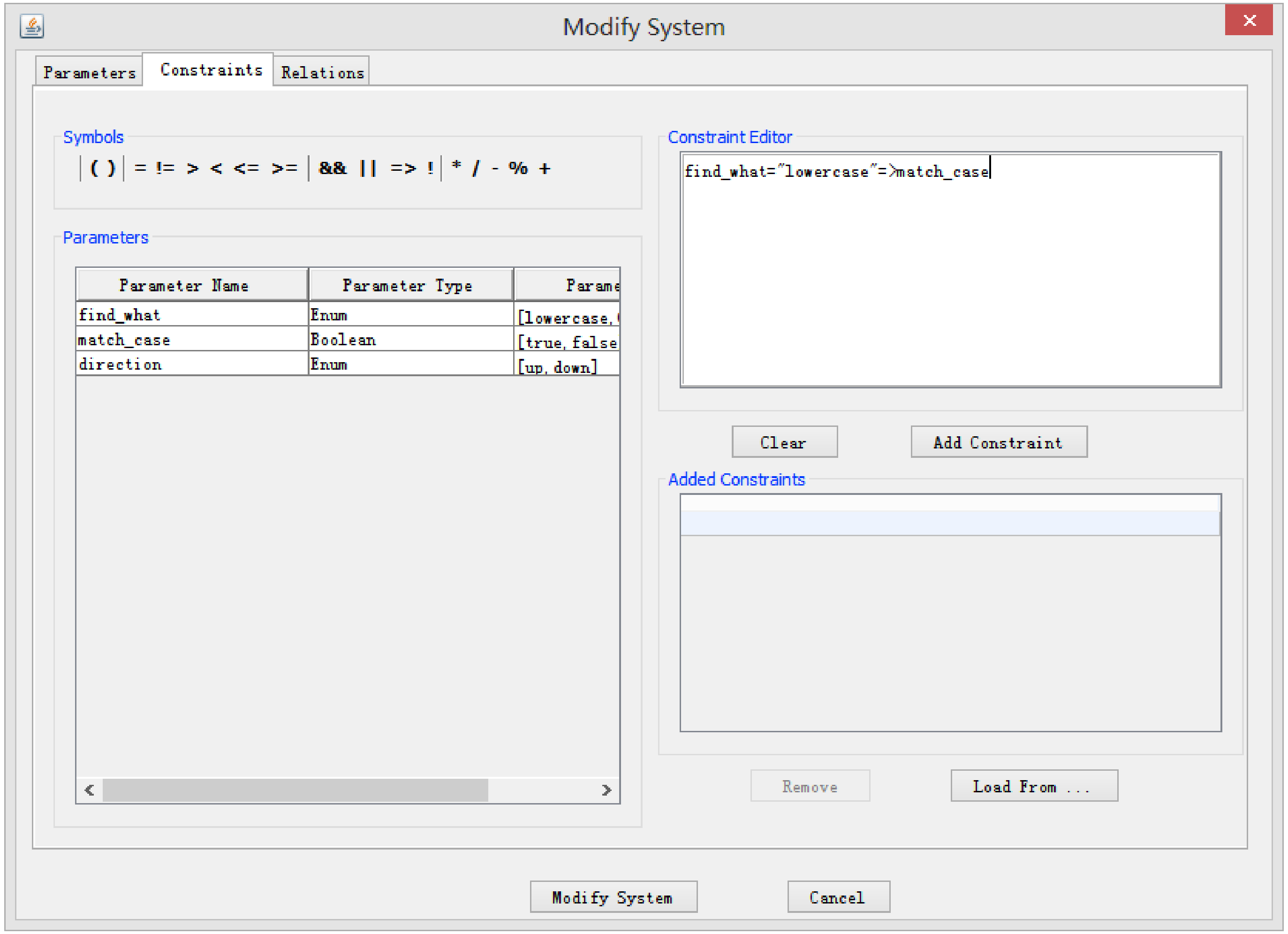
- Typically expressed using first-order logic
  - The tools will not generate combinations that make these constraints false.
  - e.g. `not (month=February and day>=30)`
  - e.g. if `find_what` = `lowercase` then `match_case` = `true`
  - ACTS, PICT, and TSL all support property constraints

Property Constraints in ATCS

```
<Constraint> ::= <Simple_Constraint>|
                 <Constraint><Boolean_Op><Constraint>

<Simple_Constraint>::=  <Term><Relational_Op><Term>

<Term> ::= <Parameter>|
           <Parameter><Arithmetic_Op><Parameter>|
           <Parameter><Arithmetic_Op><Value>

<Boolean_Op>    :=“&&” |“||” |“=>”
<Relational_Op> := “=” |“!=” | “>” | “<” | “>=” | “<=”
<Arithmetic_Op> := “+” | “-” | “*” | “/” | “%”
```

Property Constraints in ATCS

- Example: `find_what="lowercase"=>match_case`



Parameter Name	Parameter Type	Parame
find_what	Enum	[lowercase,
match_case	Boolean	[true, false
direction	Enum	[up, down]

```
find_what="lowercase"=>match_case
```

Property Constraints in PICT

- Essentially the same with ATCS

- Minor differences:
  - ACTS: `a!=b` PICT: `[a]<>[b]`
  - ACTS: `A => B` PICT: `IF A THEN B`
  - PICT additionally supports `IF A THEN B ELSE C`
  - PICT directly supports set operation.
    e.g. `IF [a] in {12, 24} THEN [compress]="true"`
  - In ACTS, you need to use `a=12||a=24` instead.

Property Constraints in TSL

- Use `[`**`property`**` ...]` and `[`**`if`**` ...]` to express property constraints

Example 1:

```
P1:
  c1.  [property p1]
P2:
  c2.  [if p1]
```

mean P2=c2 => P1=c1

That is, a combination with P2=c2 but P1!=c1 is infeasible

Property Constraints in TSL

- Use `[`**`property`**` ...]` and `[`**`if`**` ...]` to express property constraints

Example 2:

```
P1:
    c1.  [property p1]
P2:
    c2.  [property p1]
P3:
    c3.  [if p1]
```

mean `P3=c3 => ( P1=c1 || P2=c2 )`

That is, a combination with P3=c3 but P1!=c1 and P2!=c2 is infeasible

Property Constraints in TSL

- Use `[property ...]` and `[if ...]` to express property constraints

Example 3:

```
P1:
    c1.  [property p1]
P2:
    c2.  [property p2]
P3:
    c3.  [if p1 && p2]
```

mean `P3=c3`=> (`P1=c1 && P2=c2`)

That is, a combination with P3=c3 but P1!=c1 or P2!=c2 is infeasible

Property Constraints in TSL

- Use `[property ...]` and `[if ...]` to express property constraints

Example 4:

```
P1:
   c1.  [property p1]
P2:
   c2.  [property p2]
P3:
   c3.  [if p1 || p2]
```

mean `P3=c3=>` (`P1=c1 || P2=c2)`

The same with example 2!

Error/Single Constraints

- Error/single constraint is usually expressed as a *label* on a value or a class of the parameter.
- It means that we need only one of the combinations that covers this value/class, regardless of interaction strength.
    - The difference between error constraint and single constraint is conceptual only – the former rules out redundancy caused by invalid input, the latter by valid input.
    - The tools handle them in the same way.
- Example:

```
public static void main (String args[]) {
   if (args.length > 2) {
      Quit.now("Usage: java STS attributes.txt [s|st|ts|t]");
   }
```

We shall label the class “args.length>2” with an error constraint.

Error/Single Constraints in PICT

- Use '~' as the label

- Example:

```
parameter1: ~-1, 0, 1, 2
```

means that whenever parameter1 takes the value of -1, the system will report an error and stop. Therefore, we only need to cover one of the combinations in which parameter1 is -1 – it is pointless to try more as the system does not use the other parameters when parameter1 is -1.

- **Note**: ATCS does not support error/single constraints

Error/Single Constraints in TSL

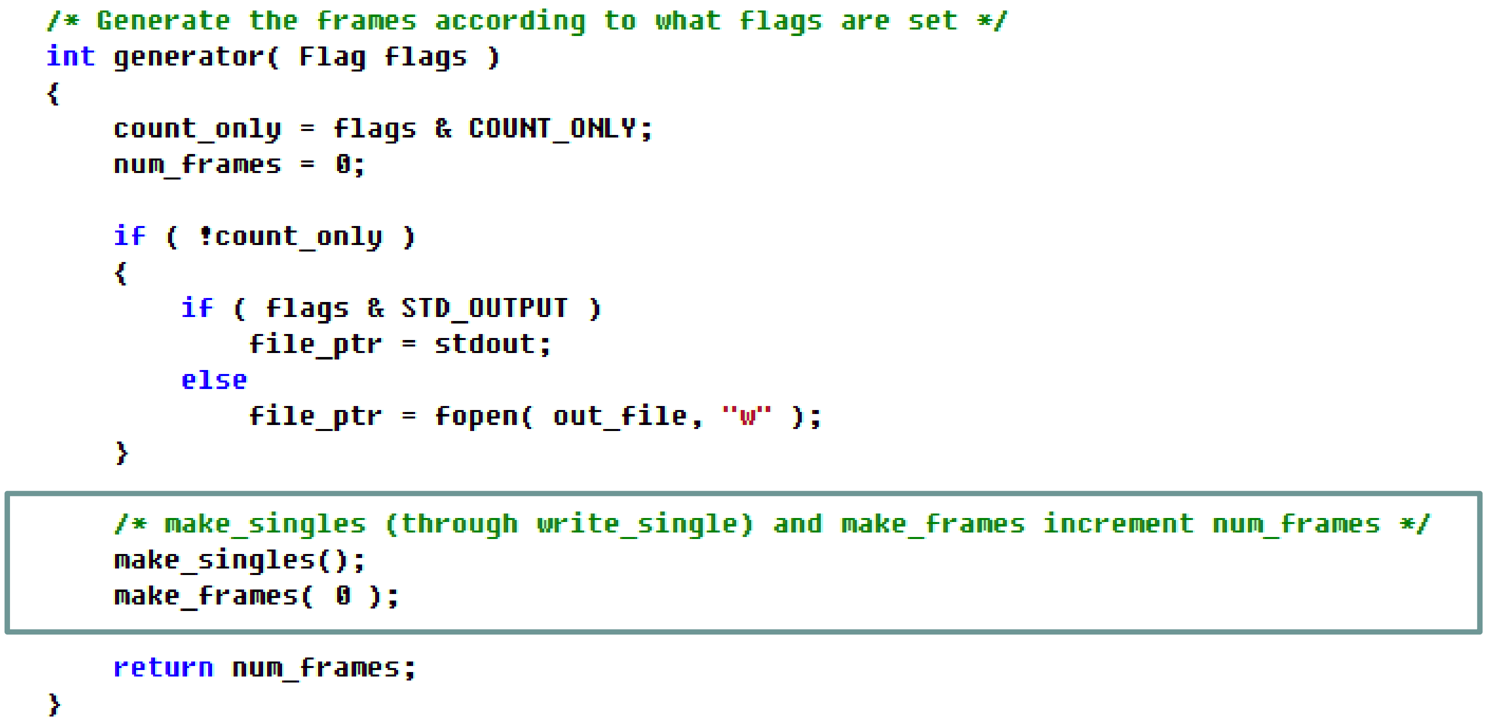
- Use `[`**`single`**`]` and `[`**`error`**`]` as the label.
- Much more sophisticated, as they can be used with property constraints to express *conditional error/single constraints*
  - Not possible with PICT and ATCS.
- Example:

```
P1:
   c1. [property p1]
P2:
   c2. [if p1] [else] [error]
```

means combinations with P1!=c1 and P2=c2 are invalid combinations (but feasible!) that trigger the system to report error and stop. We only need one of them in the test suite.

How the Tool Works

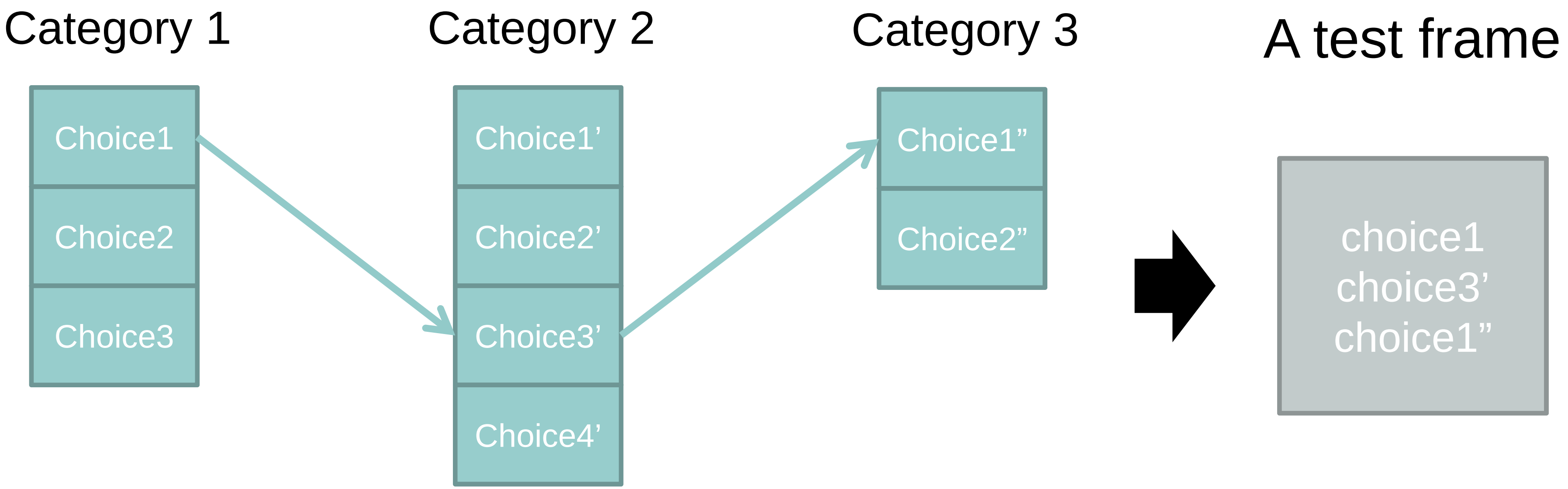
- **Step 1**: Read the TSL specification.
- **Step 2**: Generate test frames that satisfy [single]/[error] constraints.
- **Step 3**: Generate test frames that do not satisfy any [single]/[error] constraints



```
/* Generate the frames according to what flags are set */
int generator( Flag flags )
{
    count_only = flags & COUNT_ONLY;
    num_frames = 0;

    if ( !count_only )
    {
        if ( flags & STD_OUTPUT )
            file_ptr = stdout;
        else
            file_ptr = fopen( out_file, "w" );
    }

    /* make_singles (through write_single) and make_frames increment num_frames */
    make_singles();
    make_frames( 0 );

    return num_frames;
}
```

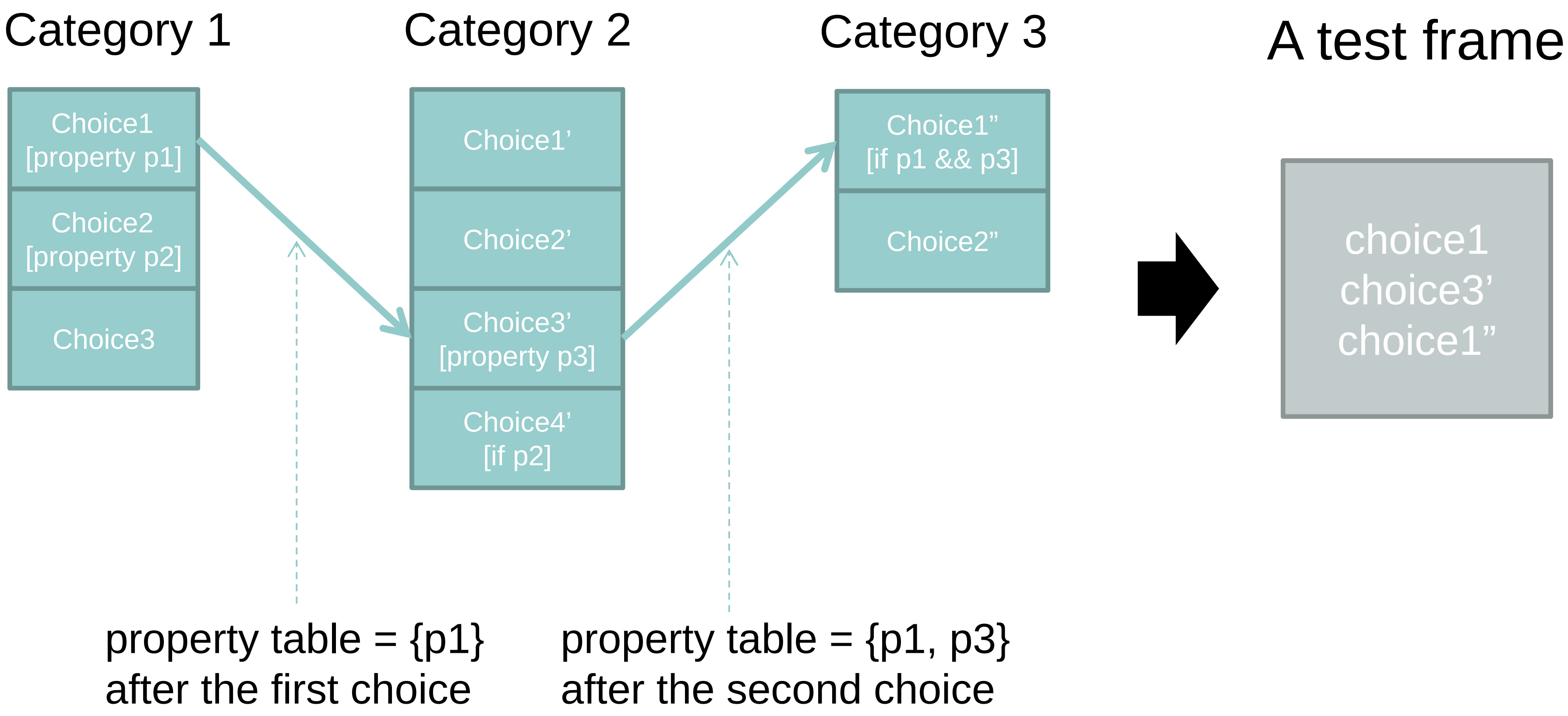
How the Tool Works

- Without considering constraints, generate test frames are easy
  - Backtracking



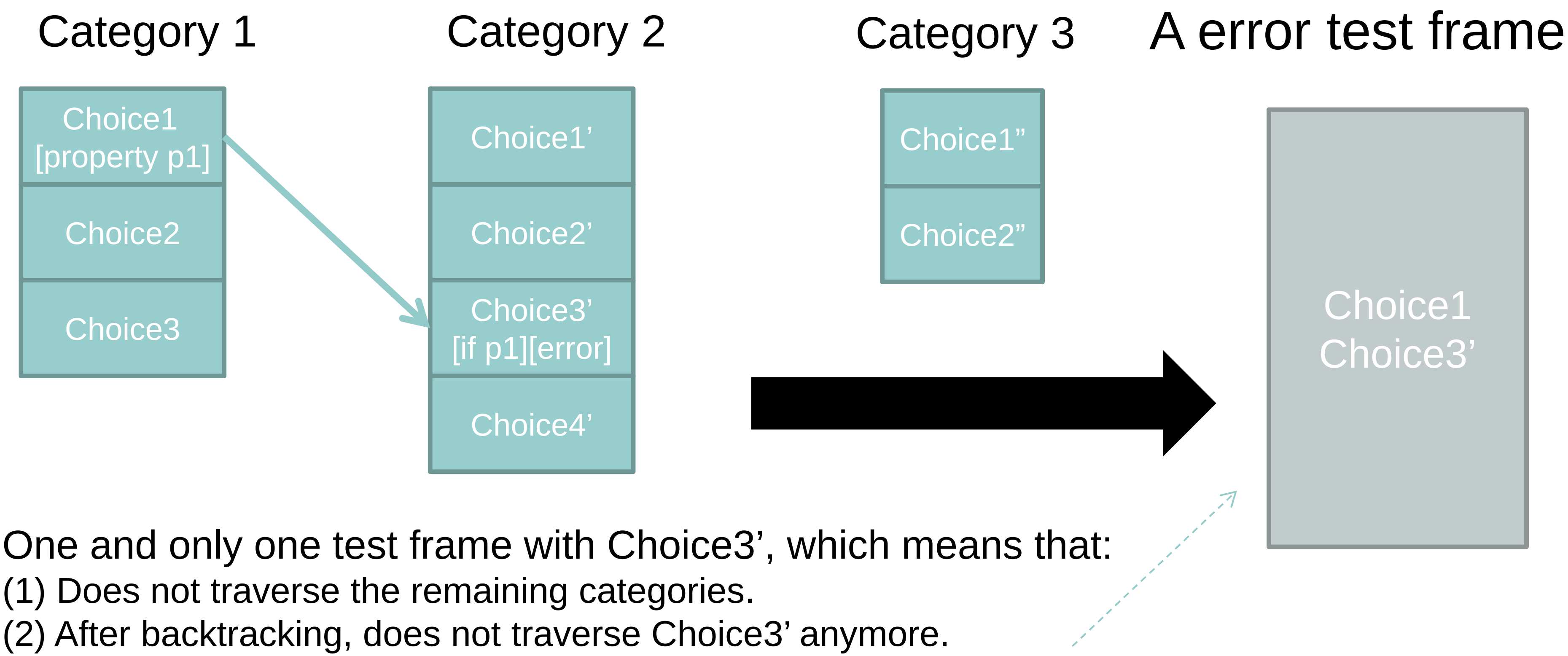
Considering Property Constraints

- Maintaining a property table
  - Add property to the table when selecting a choice
  - Remove it from the table when backtracking



Considering Error/Single Constraints

- Stop when encountering a choice with [error]/[single] constraint
  - If [error]/[single] is conditional, then the condition needs to be checked first against the property table.



One and only one test frame with Choice3’, which means that:
(1) Does not traverse the remaining categories.
(2) After backtracking, does not traverse Choice3’ anymore.

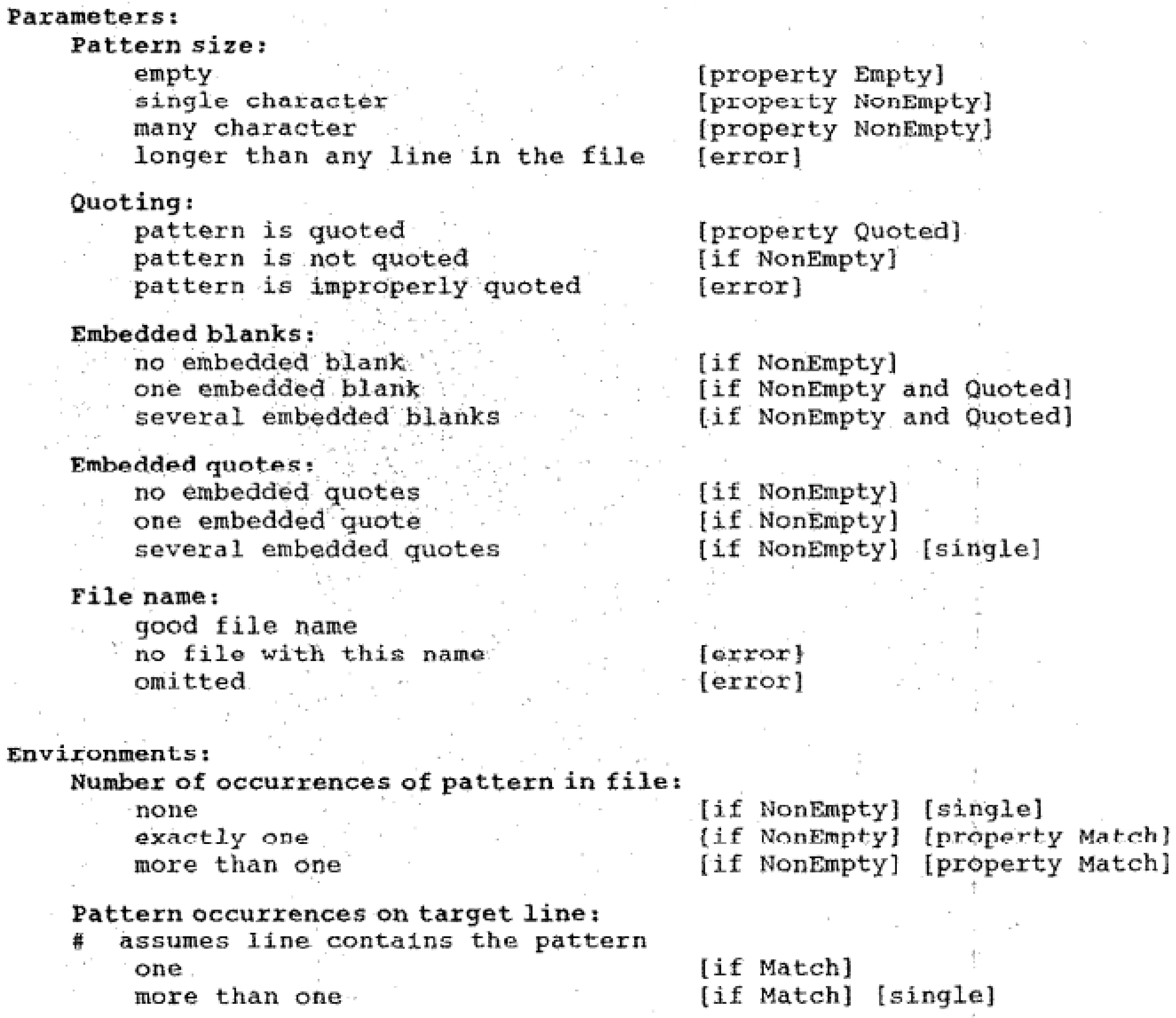
How Choices Are Selected

- Examples:
  - `[if NonEmpty]`
    Reads: if `NonEmpty` is true combine this choice with others.
  - `[if Hmmmmm] [single]`
    Reads: if `Hmmmmm` is true make a single frame with this choice.
  - `[if Radical] [property Cool] [else] [error]`
    Reads: if Radical is true set Cool to true and combine this choice with others, else make an error frame with this choice.
  - `[if NoWay] [single] [else]`
    Reads: if NoWay is true make a single frame with this choice, else combine this choice with others.

How Choices Are Selected

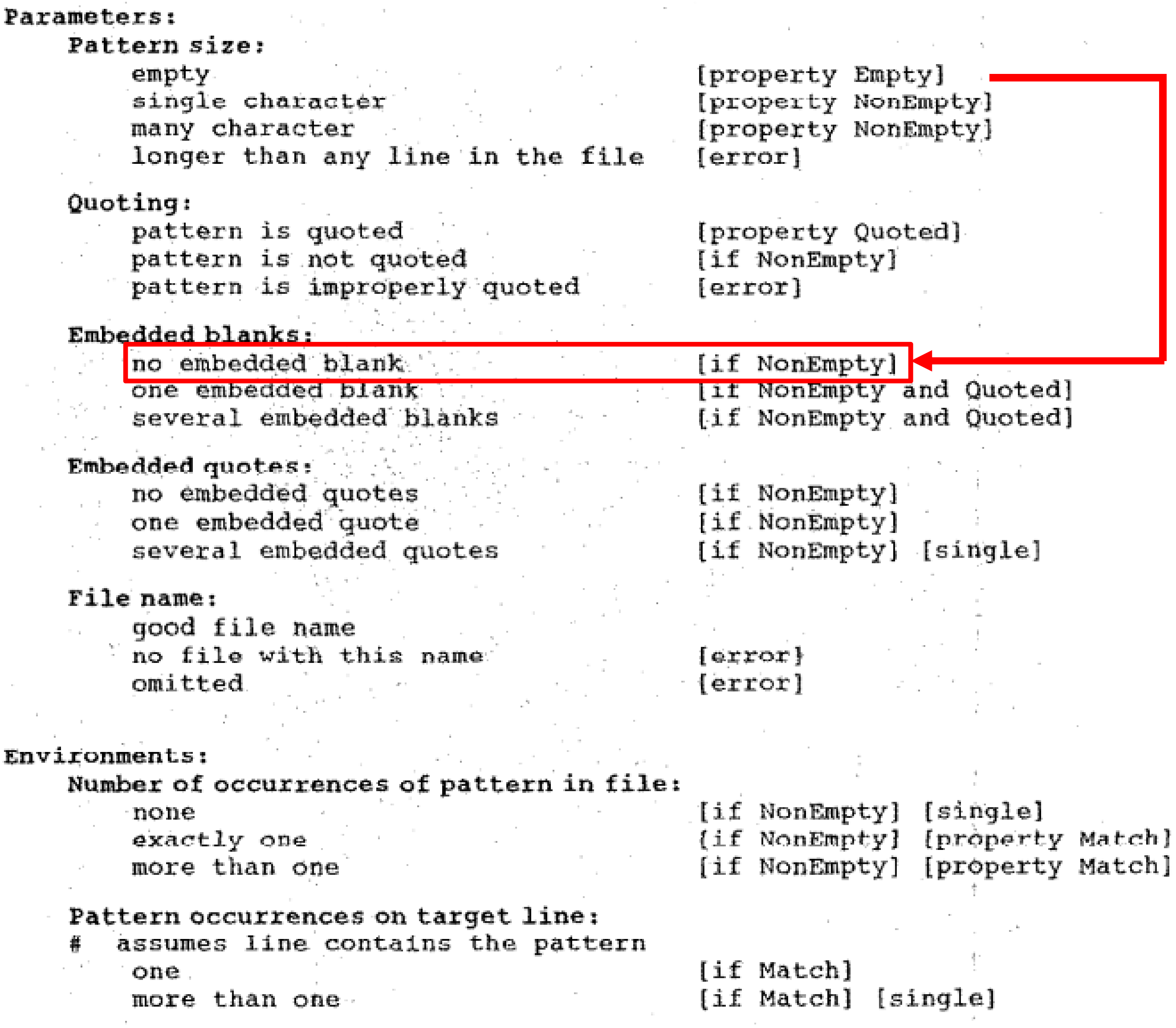
- Examples:
  - `[property IC] [single]`
    Reads: make a single frame with this choice. (the property list is ignored)
  - `[single] [if Random] [property RandQuoted]`
    Reads: make a single frame with this choice. (the selector expression is ignored)
  - `[property Oh, Yeah]`
    Reads: set `Oh` and `Yeah` to true and combine this choice with others.
  - `[property Long] [if Unquoted] [error] [else] [property Zero]`
    Reads: if Unquoted is true make an error frame with this choice, else set Zero to true and combine this choice with others. (the first property list is ignored)

Complete TSL for **find**



```
Parameters:
    Pattern size:
        empty                                  [property Empty]
        single character                       [property NonEmpty]
        many character                         [property NonEmpty]
        longer than any line in the file       [error]

    Quoting:
        pattern is quoted                      [property Quoted]
        pattern is not quoted                  [if NonEmpty]
        pattern is improperly quoted           [error]

    Embedded blanks:
        no embedded blank                      [if NonEmpty]
        one embedded blank                     [if NonEmpty and Quoted]
        several embedded blanks                [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                     [if NonEmpty]
        one embedded quote                     [if NonEmpty]
        several embedded quotes                [if NonEmpty] [single]

    File name:
        good file name
        no file with this name                 [error]
        omitted                                [error]


Environments:
    Number of occurrences of pattern in file:
        none                                   [if NonEmpty] [single]
        exactly one                            [if NonEmpty] [property Match]
        more than one                          [if NonEmpty] [property Match]

    Pattern occurrences on target line:
    #   assumes line contains the pattern
        one                                    [if Match]
        more than one                          [if Match] [single]
```

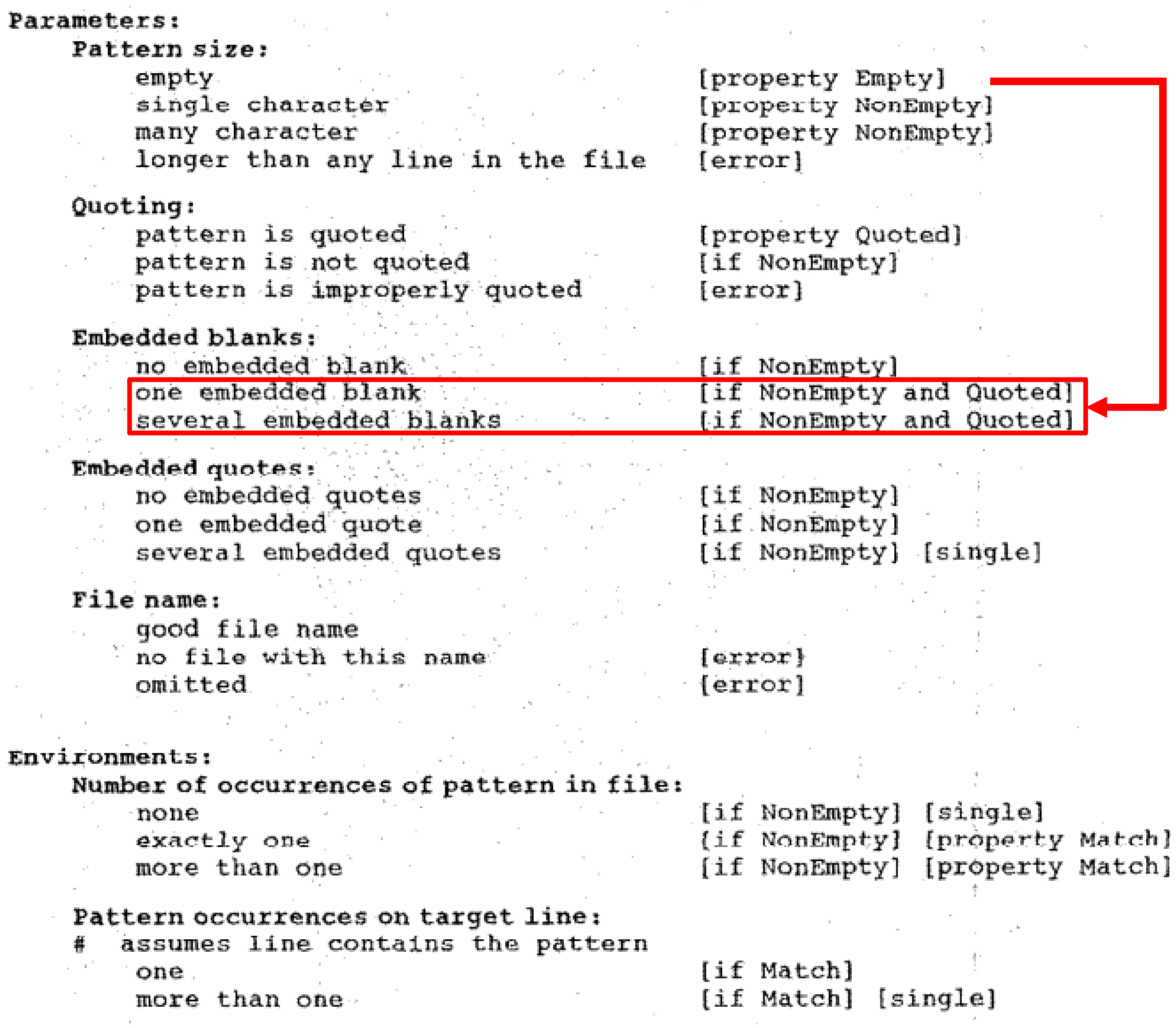
Complete TSL for `find`



```
Parameters:
    Pattern size:
        empty                               [property Empty]
        single character                    [property NonEmpty]
        many character                      [property NonEmpty]
        longer than any line in the file    [error]

    Quoting:
        pattern is quoted                   [property Quoted]
        pattern is not quoted               [if NonEmpty]
        pattern is improperly quoted        [error]

    Embedded blanks:
        no embedded blank                   [if NonEmpty]
        one embedded blank                  [if NonEmpty and Quoted]
        several embedded blanks             [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                  [if NonEmpty]
        one embedded quote                  [if NonEmpty]
        several embedded quotes             [if NonEmpty] [single]

    File name:
        good file name
        no file with this name              [error]
        omitted                             [error]


Environments:
    Number of occurrences of pattern in file:
        none                                [if NonEmpty] [single]
        exactly one                         [if NonEmpty] [property Match]
        more than one                       [if NonEmpty] [property Match]

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one                                 [if Match]
        more than one                       [if Match] [single]
```

Reduce the redundancy: if the pattern is empty then it surely does not include embedded blank

Complete TSL for `find`



```
Parameters:
    Pattern size:
        empty                                [property Empty]
        single character                     [property NonEmpty]
        many character                       [property NonEmpty]
        longer than any line in the file     [error]

    Quoting:
        pattern is quoted                    [property Quoted]
        pattern is not quoted                [if NonEmpty]
        pattern is improperly quoted         [error]

    Embedded blanks:
        no embedded blank                    [if NonEmpty]
        one embedded blank                   [if NonEmpty and Quoted]
        several embedded blanks              [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                   [if NonEmpty]
        one embedded quote                   [if NonEmpty]
        several embedded quotes              [if NonEmpty] [single]

    File name:
        good file name
        no file with this name               [error]
        omitted                              [error]


Environments:
    Number of occurrences of pattern in file:
        none                                 [if NonEmpty] [single]
        exactly one                          [if NonEmpty] [property Match]
        more than one                        [if NonEmpty] [property Match]

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one                                  [if Match]
        more than one                        [if Match] [single]
```

Avoid infeasible combination:
if the pattern is not quoted, then it is impossible to include embedded blanks

Case Study with TSL: 电脑销售系统

Direct Input: a description on the computer system that is selected by the customer.

Indirect Input (environment): a database that contains all computer models available for sale.

Case Study with TSL: 电脑销售系统

- Requirement on the input
  - The computer system description consists of a model number and a set of (slot, component) pairs.
  - The model number determines a set of constraints on available components and the available logic slots for components.
  - Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs.

- Example:
  - *The required “slots” of the ThinkPad E430 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. The optional slots include external storage devices such as a CD/DVD writer.*

Case Study with TSL: 电脑销售系统

- Requirement on the input
  - The set of (slot, component) pairs are corresponding to the required and optional slots of the model.
  - A component is a choice that can be varied within a model. Valid components is determined by the model. The special value empty is allowed for optional slots.
  - In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

- Example:
  - *The default configuration of the ThinkPad E430 includes 2 gigabytes of memory; 4 and 8 gigabyte memory are also available. The default operating system is Windows 8, home edition, but Windows 8, professional edition may also be selected. The professional edition requires at least 4 gigabytes of memory.*

Step 1: Parameters and Choices

- The requirement:
  - *The computer system description consists of a model number and a set of (slot, component) pairs.*

- **Parameter**: model number of the computer system
  - Malformed
  - Not in database
  - Valid

Step 1: Parameters and Choices

- The requirement:
  - The model number determines a set of constraints on available components and the available logic slots for components. Slots may be required or optional.

- **Parameter**: number of required slots specified by the model
  - 0
  - 1
  - Many #more than one
- **Parameter**: number of optional slots specified by the model
  - 0
  - 1
  - Many #more than one

Step 1: Parameters and Choices

- The requirement:
  - The set of (slot, component) pairs are corresponding to the required and optional slots of the model.

- **Parameter**: correspondence between slots specified by the model and slots described by the system
  - Omitted slots #e.g. model specifies a DVD slot but system omits it.
  - Extra slots
  - Mismatched slots #that is, have both omitted slots and extra slots
  - Complete correspondence

Step 1: Parameters and Choices

- The requirement:
  - The computer system description consists of a model number and a set of (slot, component) pairs. Required slots must be assigned with a suitable component to obtain a legal configuration
- **Parameter**: how many required slots are non-empty in the system
  - 0
  - < total number of required slots specified by the model
  - = total number of required slots specified by the model

Step 1: Parameters and Choices

- The requirement:
  - The model number determines a set of constraints on available components
  - A component is a choice that can be varied within a model. Valid components for each slot is determined by the model.
  - In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

- **Parameter**: Selection of components for required slots in the system
  - All valid choices.
  - ≥ 1 incompatible with model
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection

Step 1: Parameters and Choices

- The requirement:
  - The computer system description consists of a model number and a set of (slot, component) pairs.
  - Optional slots may be left empty or filled depending on the customers' needs.
- **Parameter**: how many optional slots are non-empty in the system
  - 0
  - < total number of optional slots specified by the model
  - = total number of optional slots specified by the model

Step 1: Parameters and Choices

- The requirement:
  - The special value empty is allowed for optional slots.
  - In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

- **Parameter**: selection of components for optional slots in the system
  - All valid with empty choices.
  - All valid with at least one non-empty choice.
  - ≥ 1 incompatible with model
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection

Step 2: Identify Constraints

Number of required slots specified by the model:

0	
1	[property RSNE]
Many	[property RSNE]

How many required slots are non-empty in the system:

0	[if RSNE] [error]
< number required slots	[error]
= number required slots	

Number of optional slots specified by the model:

0	
1	[property OSNE]
Many	[property OSNE]

How many optional slots are non-empty in the system:

0	
< number optional slots	[if OSNE]
= number optinonal slots	[if OSNE]

Step 2: Identify Constraints

Model number:

Malformed	[error]
Not in database	[error]
Valid	

Correspondence between slots specified by the model and slots described by the system:

Omitted slots	[error]
Extra slots	[error]
Mismatched slots	[error]
Complete correspondence	

Step 2: Identify Constraints

- Selection of components for required slots in the system
  - All valid choices.
  - ≥ 1 incompatible with model [if RSNE] [error]
  - ≥ 1 incompatible with slots [if RSNE] [error]
  - ≥ 1 incompatible with another selection [if RSNE] [error]

- selection of components for optional slots in the system
  - All valid with empty choices.
  - All valid with at least one non-empty choice.
  - ≥ 1 incompatible with model [if OSNE] [error]
  - ≥ 1 incompatible with slots [if OSNE] [error]
  - ≥ 1 incompatible with another selection [if OSNE] [error]

The Complete TSL Specification

- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number of required slots specified by the model
  - 0
  - 1 [property RSNE]
  - Many [property RSNE]
- Number of optional slots specified by the model
  - 0
  - 1 [property OSNE]
  - Many [property OSNE]
- Correspondence between slots specified by the model and slots described by the system
  - Omitted slots [error]
  - Extra slots [error]
  - Mismatched slots [error]
  - Complete correspondence
- # of required slots that are non-empty in the system
  - 0 [if RSNE] [error]
  - < number required slots [if RSNE] [error]
  - = number required slots
- Selection of components for required slots in the system
  - All valid choices.
  - ≥ 1 incompatible with slots [if RSNE][error]
  - ≥ 1 incompatible with another selection [if RSNE][error]
  - ≥ 1 incompatible with model [if RSNE][error]
- # of optional slots that are non-empty in the system
  - 0
  - < number optional slots [if OSNE]
  - = number optional slots [if OSNE]
- Selection of components for optional slots in the system
  - All valid with empty.
  - All valid with at least one non-empty choice. [if OSNE]
  - ≥ 1 incompatible with slots [if OSNE][error]
  - ≥ 1 incompatible with another selection [if OSNE][error]
  - ≥ 1 incompatible with model [if OSNE][error]

An Example Test Frame

- Model number: **Valid**
- Number of required slots specified by the model: **Many**
- Number of optional slots specified by the model: **0**
- Correspondence between slots specified by the model and slots described by the system: **complete**
- # of required slots that are non-empty in the system: **=**
- Selection of components for required slots in the system: **all valid**
- # of optional slots that are non-empty in the system: **0**
- Selection of components for optional slots in the system: **all valid with empty.**

Thank you!