

# Functional testing

## Testing in-the-large

(Chapter 6)

# Outline

- Black-box testing
- Testing in-the-large

# Functional vs Structural Testing

- WHITE BOX (structural) testing
  - partitioning criteria based on module's internal code
  - tests *what the program does*
  - derives test cases from program code
- BLACK BOX (functional) testing
  - partitioning criteria based on the module's specification
  - tests *what the program is supposed to do (derive test cases from specifications)*

# Systematic black-box techniques

- Testing driven by logic specifications (pre and postconditions)
- Syntax-driven testing
- Decision table based testing
- Equivalence class testing
- Random/fuzz testing

# Test driven by logic specification

```
{n > 0} -- n is a constant value
procedure search (table: in integer_array;
                  n: in integer;
                  element: in integer;
                  found: out Boolean);
{found ≡ (exists i (1 ≤ i ≤ n and table (i) = element))}
```

rewrite postcondition as

$$\begin{aligned} &(\text{exists } i (1 \leq i \leq n \text{ and table } (i) = \text{element})) \rightarrow \text{found} \\ &\quad \wedge \\ &\neg (\text{exists } i (1 \leq i \leq n \text{ and table } (i) = \text{element})) \rightarrow \neg \text{found} \end{aligned}$$

use **condition coverage** to derive two test cases

1.  $(\text{exists } i (1 \leq i \leq n \text{ and table } (i) = \text{element})) \rightarrow \text{found}$
2.  $\neg (\text{exists } i (1 \leq i \leq n \text{ and table } (i) = \text{element})) \rightarrow \neg \text{found}$

# Test driven by logic specification

```
{n > 0}  
procedure sort (a: in out integer_array;  
                n: in integer);  
{ ? }
```

postcondition:  
out\_a a permutation of in\_a  
     $\wedge$   
(forall i ( $1 \leq i < n$  and  $a[i] \leq a[i+1]$ ))

use **condition coverage** to derive two test cases

1. permutation
2. array sorted

# Test driven by logic specification

```
{ length ≥ 0 ∧ start ≥ 0 ∧ start + length < a.len}
procedure substring (a: in char_array;
                      start: in integer;
                      length: in integer
                      b: out char_array);
{ ? }
```

postcondition:

$b.len = length \wedge$

(forall  $i$  ( $0 \leq i < length$  and  $b[i] = a[start+i]$ ))

use **condition coverage** to derive two test cases

1.  $b.len = length$
2.  $b[i] = a[start+i]$

# Syntax-driven testing (1)

- Consider testing an interpreter of the following language

$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle + \langle \text{term} \rangle |$

$\langle \text{expression} \rangle - \langle \text{term} \rangle | \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle | \langle \text{term} \rangle /$

$\langle \text{factor} \rangle | \langle \text{factor} \rangle$

$\langle \text{factor} \rangle ::= \text{ident} | \text{number} | ( \langle \text{expression} \rangle )$

# Syntax-driven testing (2)

- Apply complete coverage principle to all grammar rules
- Generate a test case for each rule of the grammar
  - note, however that the test case might also cover other rules
- Note: the specification is formal, and test generation can be automated

# Decision-table-based testing

- Condition-action table
  - Describe how different combinations of inputs generate outputs
- Example: word processor
  - conditions: user commands to make text plain (P), make boldface (B), make italic (I), emphasize (E), super emphasize (SE)
  - actions: selected text' font set to plain text (p), boldface (b), italic (i)

P	*							
B		*						*
I			*					*
E				*	*			
SE						*	*	*
E = B				*				
E = I					*			
SE = B						*		
SE = I							*	
SE = B + I								*
<b>action</b>	<b>p</b>	<b>b</b>	<b>i</b>	<b>b</b>	<b>i</b>	<b>b</b>	<b>i</b>	<b>b,i</b>

# Decision-table-based testing

- Example: triangle problem

$a + b \leq c$  (triangle inequality) -> *no triangle*

All sides different -> *scalene*

Two sides equal -> *isosceles*

All sides same length -> *equilateral*

from "Software Testing" by Paul Jorgensen

c1: a, b, c form a triangle?	N	Y	Y	Y	Y	Y	Y	Y	Y
c2: a = b?	-	Y	Y	Y	Y	N	N	N	N
c3: a = c?	-	Y	Y	N	N	Y	Y	N	N
c4: b = c?	-	Y	N	Y	N	Y	N	Y	N
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

from "Software Testing" by Paul Jorgensen

# Equivalence class testing

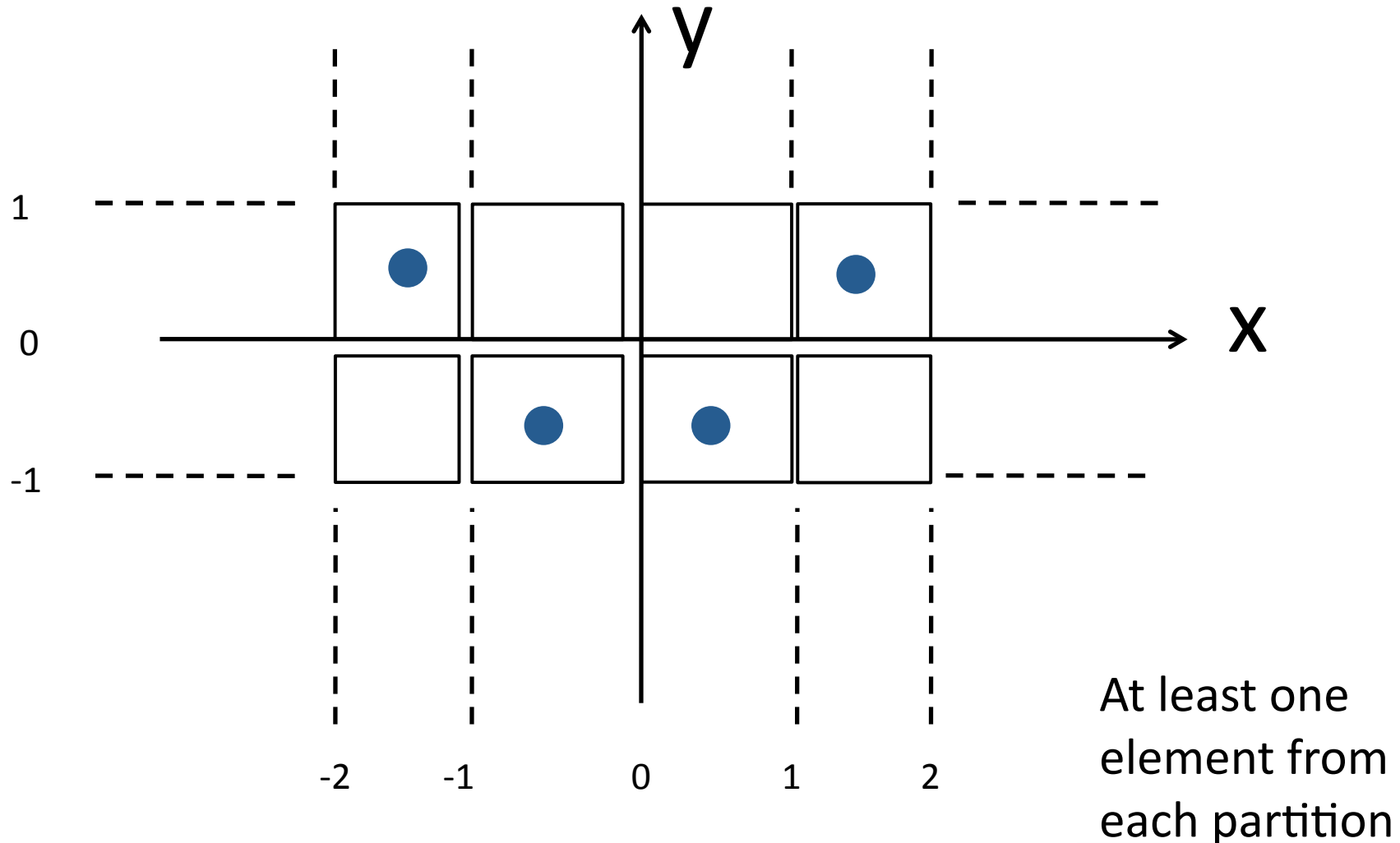
`float f (float x, float y)`

Partitions for input domain

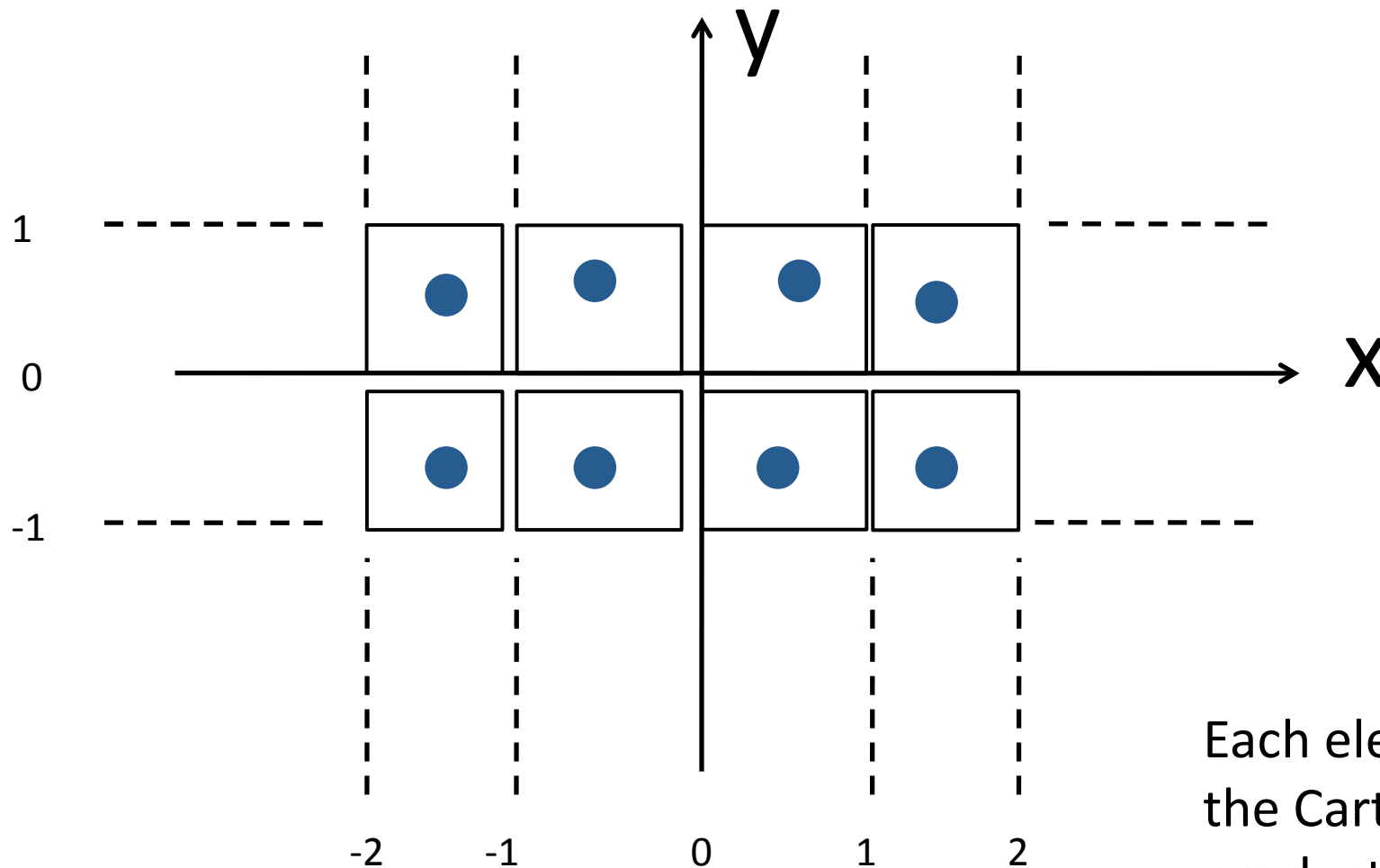
x:  $[-2, -1) \cup [-1, 0) \cup [0, 1) \cup [1, 2)$

y:  $[-1, 0) \cup [0, 1)$

# Weak equivalence class testing

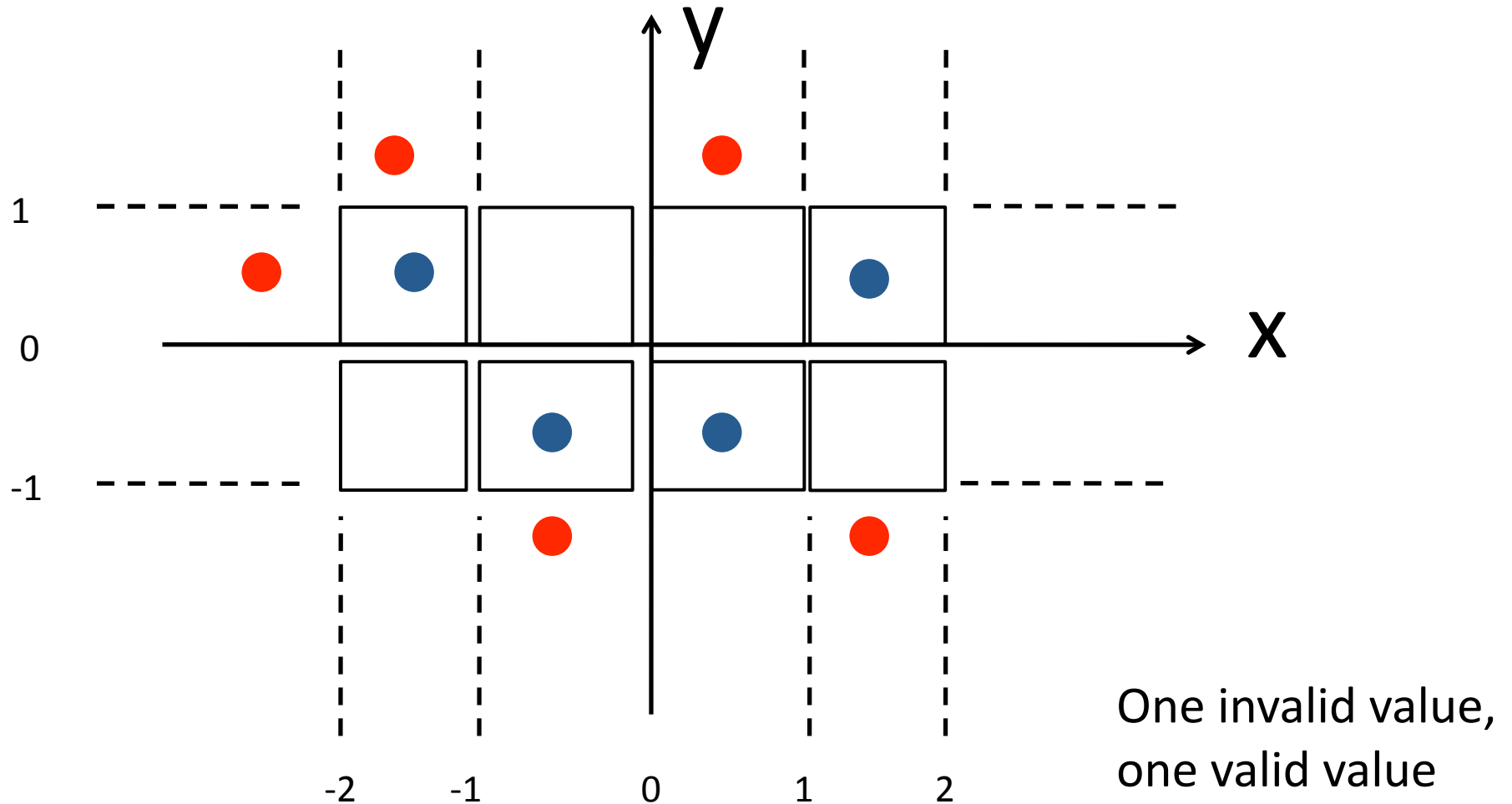


# Strong equivalence class testing

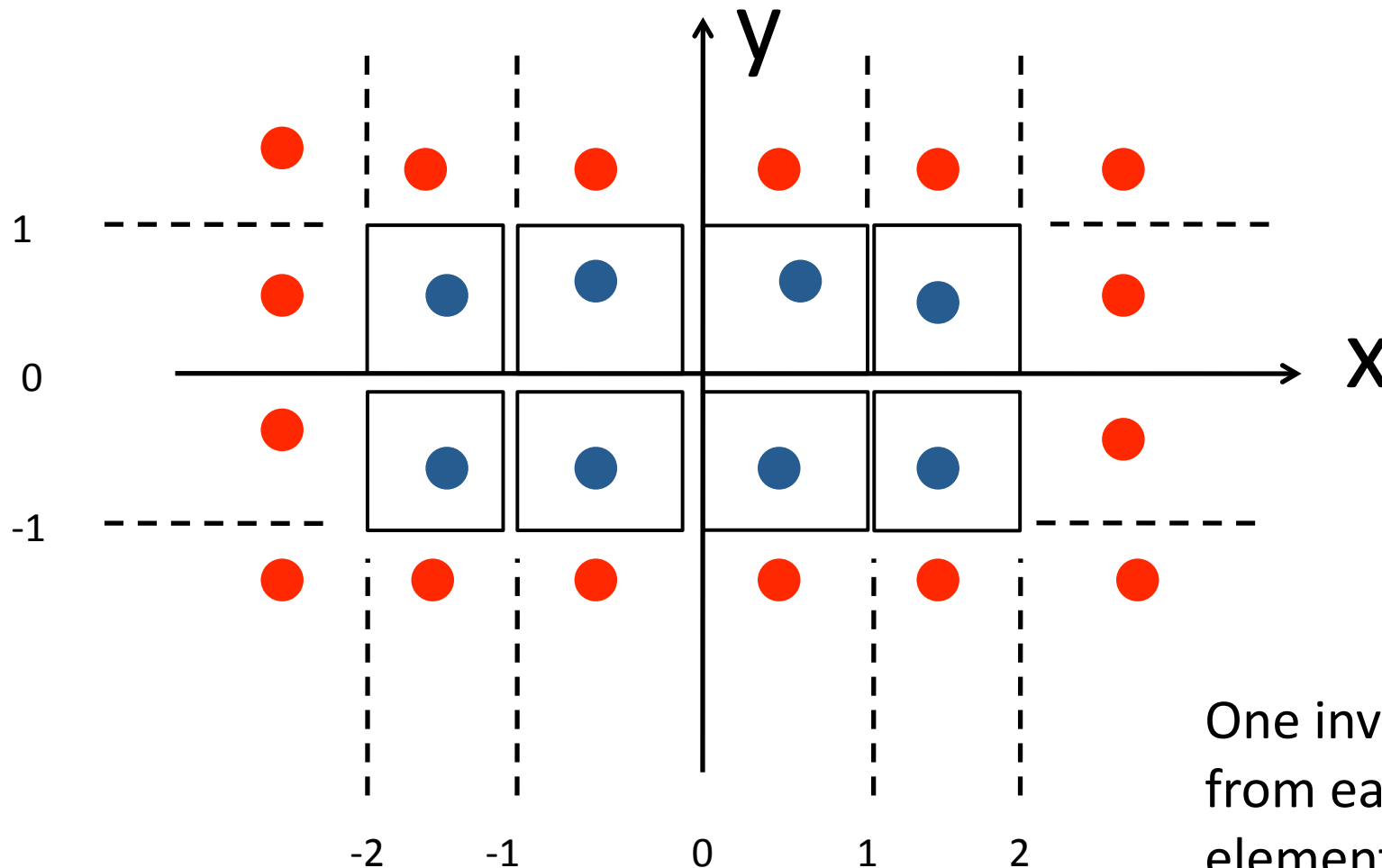


Each element of  
the Cartesian  
product

# Weak robust equivalence class testing



# Strong *robust* equivalence class testing

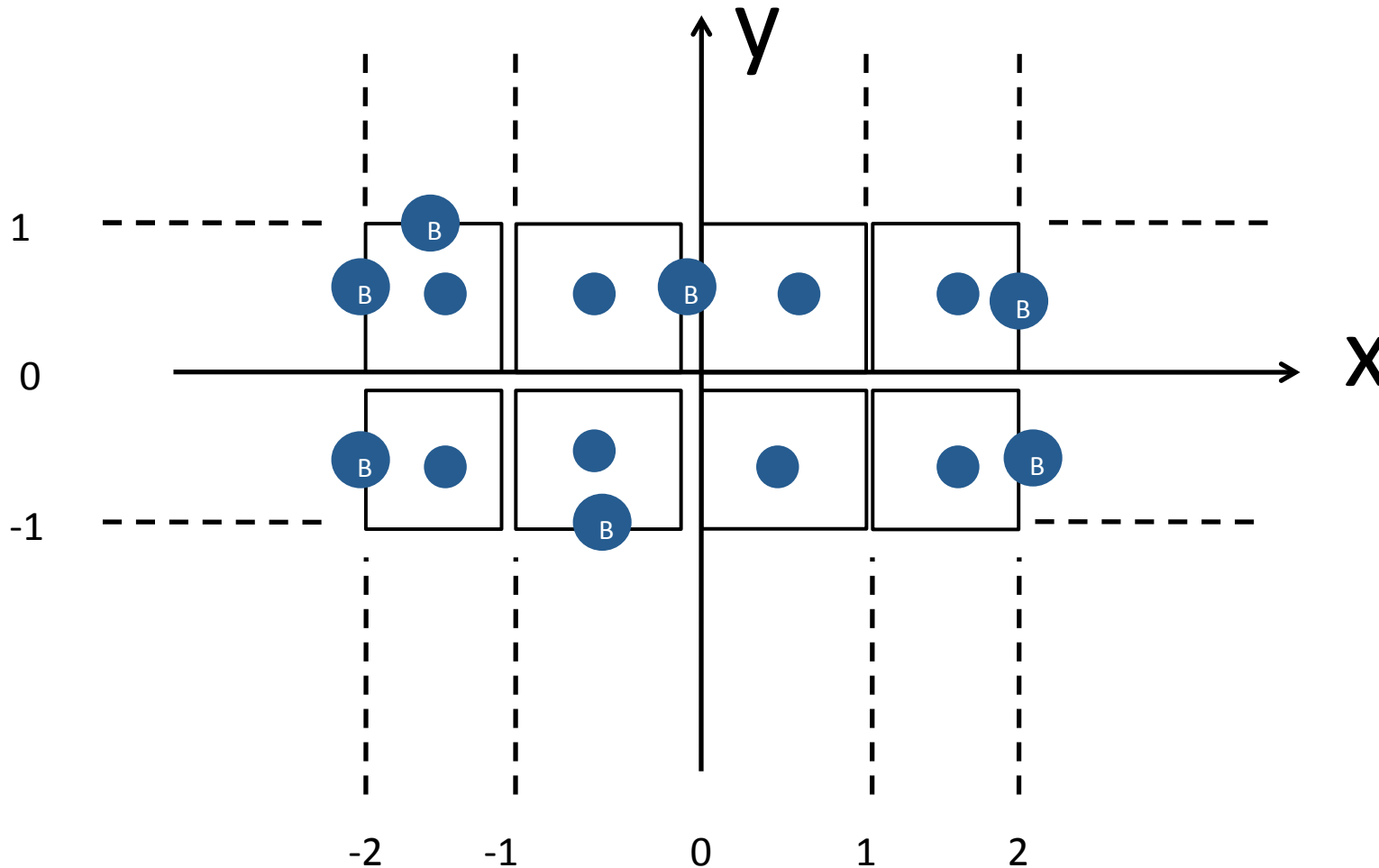


One invalid value  
from each  
element of the  
Cartesian product

# Testing boundary conditions

- Equivalence class testing assumes that behavior is "similar" for all data within a class
- Some typical programming errors, happen at the boundary between different classes
- Test not only “inside” the classes, but also at their boundaries
- Applies to both white-box and black-box testing

# Boundary testing example



# Random & fuzz testing

- Random testing
  - for input partition  $[min, max]$  pick a random value  $r$  such that  $min \leq r \leq max$
- Fuzz testing
  - Feed bad/random data as application input
  - Miller et al. fuzz testing crashes
    - > text: 9% (Linux), 7% (Mac)
    - > graphic: 64% (Windows 2000), 26% (X-Windows), 73% (Mac)
  - Countermeasures: checksums, validity/consistency checks

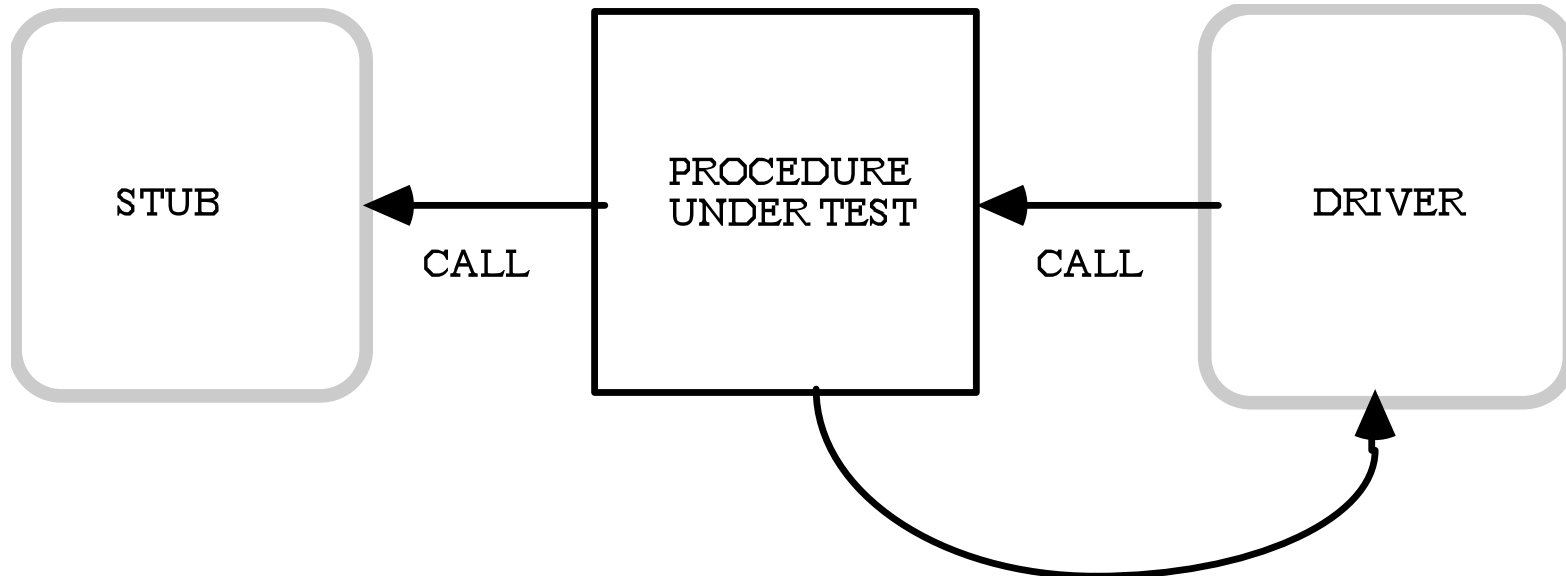
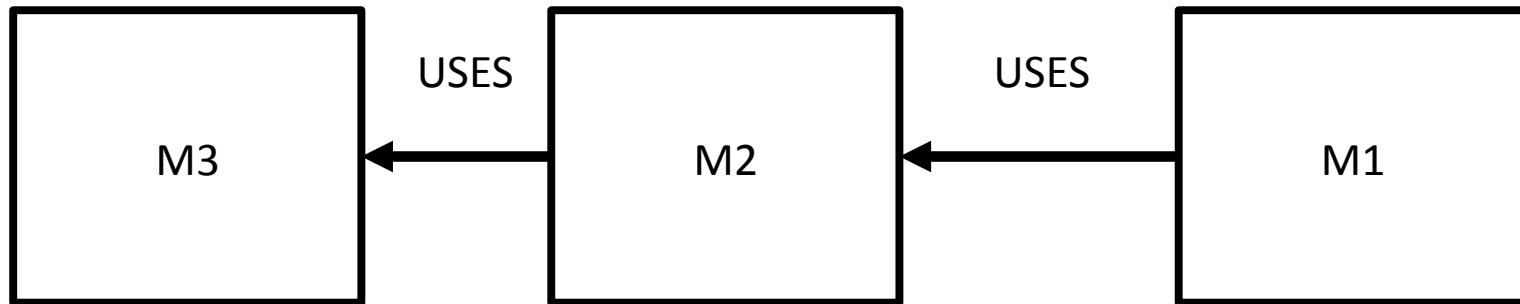
# Testing in the large

- Module testing
  - testing a single module
- Integration testing
  - integration of modules and subsystems
- System testing
  - testing the entire system
- Acceptance testing
  - performed by the customer

# Module testing

- Scaffolding needed to create the environment in which the module should be tested
  - stubs
    - modules used by the module under test
  - driver
    - module activating the module under test

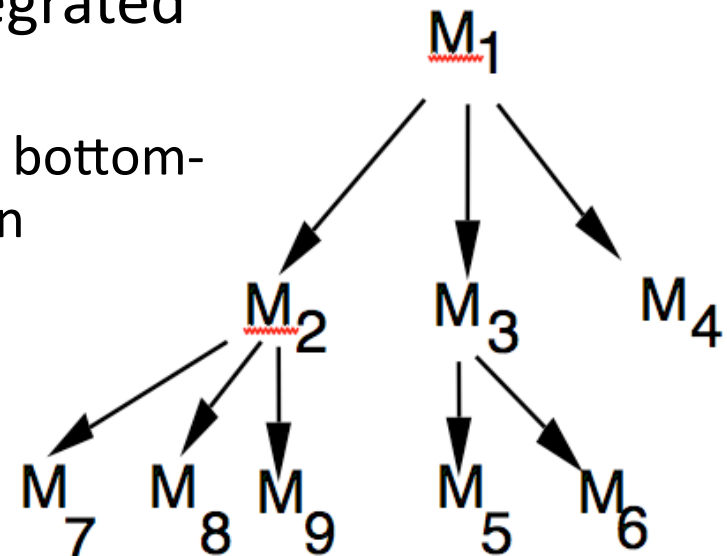
# Testing a functional module



ACCESS TO NONLOCAL VARIABLES

# Integration testing

- Big-bang approach
  - first test individual modules in isolation
  - then test integrated system
- Incremental approach
  - modules are progressively integrated and tested
    - can proceed both top-down and bottom-up according to the USES relation



# Reading for next class

**Section 6.4 (omit 6.4.2.2 and 6.4.2.3)**