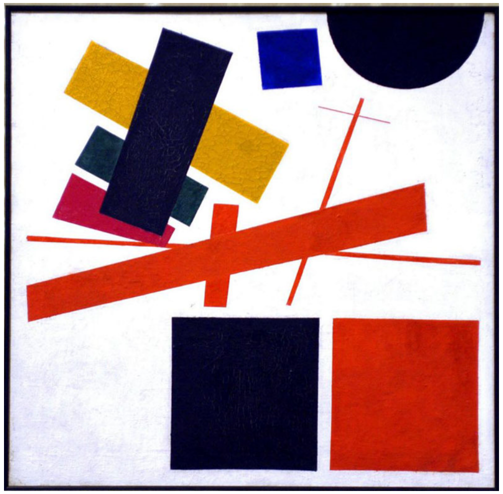
CPEN 422
Software Testing and Analysis



Functional & Combinatorial Testing

Functional vs. Structural Testing

10.2

Functional

- Any level of granularity where spec is available
- Benefit from rich domain & application knowledge
- Can identify *omissions*

Structural

- Tied to specific program structures
- Deep knowledge of actual algorithms and implementation
- Can find *surprises*

Software testing

- Software testing consists of
  1. Select a set of inputs to a program ("test inputs")
  2. Determine if the program behaves correctly on each test input

Test Cases and Inputs

assertEquals(foo(in), exp);

- **foo**: Function to be tested
- **in**: Set of input parameters
- **exp**: Expected output

- How to choose input **in** ???
  - Random is an option.....
  - Systematic is smarter though

Benefits of Random Testing

- Choice of the input data at random
- Cheap to implement
- Easy to understand
- Naive? Useless? Not really.
  - It can detect bugs!

Random testing works

- An attractive error-detection technique
  - Yields lots of test inputs
  - Actually used in industry!
  - Finds errors (empirical studies)
    - Miller et al. 1990: Unix utilities
    - Kropp et al.1998: OS services
    - Forrester et al. 2000: GUI applications
    - Claessen et al. 2000: functional programs
    - Csallner et al. 2005, Pacheco et al. 2005: object-oriented programs
    - Groce et al. 2007: flash memory, file systems

When to use Random Testing

- We have a generic **automated oracle,**
- for example
  - **Java Spec: e.equals(e) -> ALWAYS true**
  - **No crashes**
  - **No exceptions that terminate/halt the program**

- Can we do better than pure random?
  1. Use feedback from test executions

Random testing: pitfalls

1. Useful test

```
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

2. Redundant test

```
Set s = new HashSet();
s.add("bye");
assertTrue(s.equals(s));
```

3. Useful test

```
Date d = new Date(2015, 2, 14);
assertTrue(d.equals(d));
```

4. Illegal test

```
Date d = new Date(2015, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

5. Illegal test

```
Date d = new Date(2015, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Feedback-directed random test generation

- Build test inputs incrementally
  - New test inputs extend previous ones
  - e.g. a test input is a method sequence
- As soon as a test is created, execute it
- Use execution results to guide the search
  - away from redundant or illegal method sequences
  - towards sequences that create new object states

Random testing: pitfalls

1. Useful test

Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));

2. Redundant test

Set s = new HashSet();
s.add("bye");
assertTrue(s.equals(s));

3. Useful test

Date d = new Date(2015, 2, 14);
assertTrue(d.equals(d));

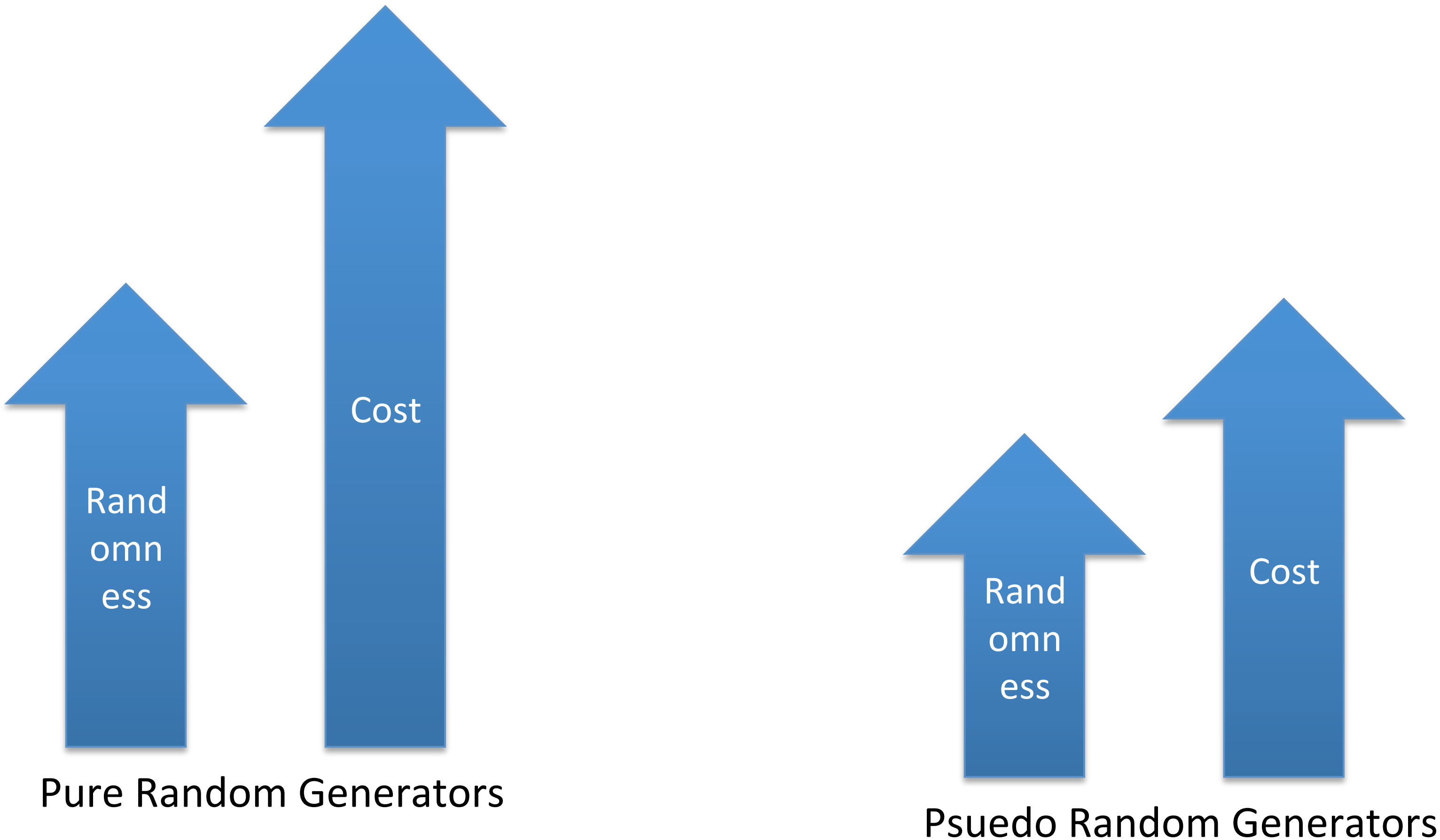
4. Illegal test

Date d = new Date(2015, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));

5. Illegal test

Date d = new Date(2015, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));

Random Generator



2.2
10.2

Random Testing?

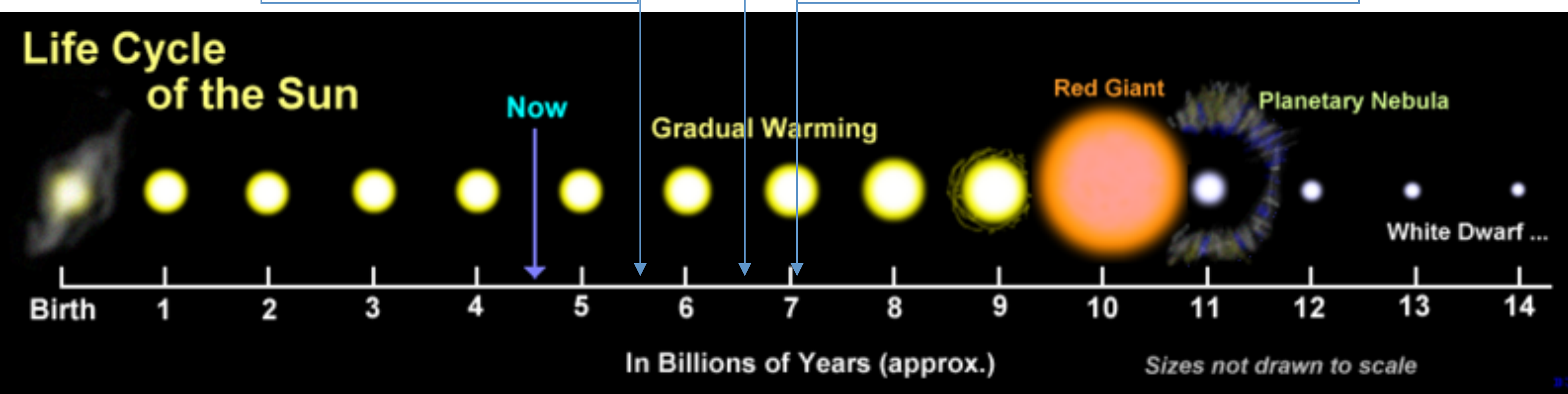
- A simple program:
  3 inputs, 1 output
- a,b,c: 32 bit integers
- trillion test cases / s.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

All oceans dry

All plants dead

Tests done



Systematic Testing: Alternative to Random Testing

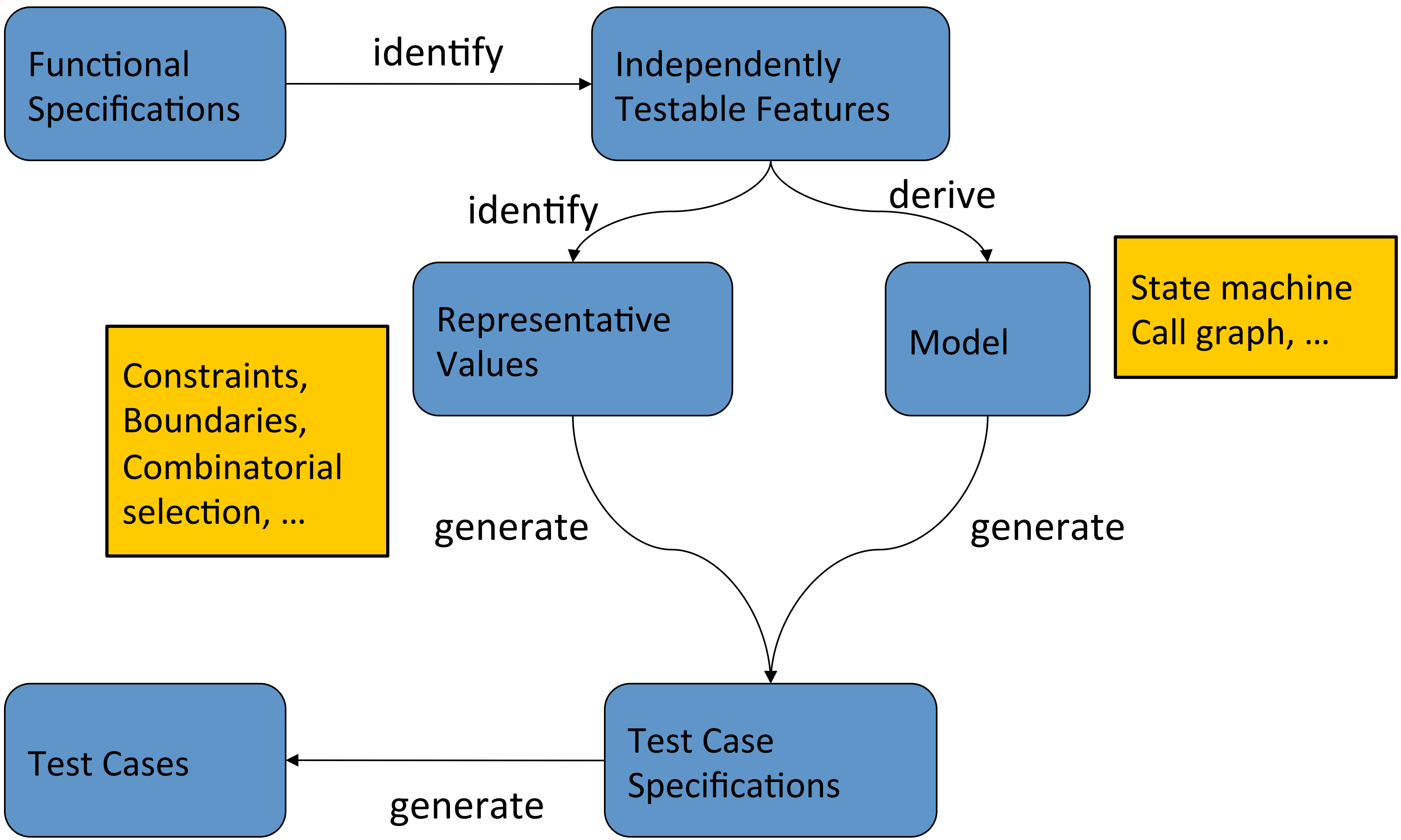
Random versus Systematic Testing

- Random (uniform):
  - Pick possible inputs uniformly
  - Avoids designer bias
    - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
  - But treats all inputs as equally valuable

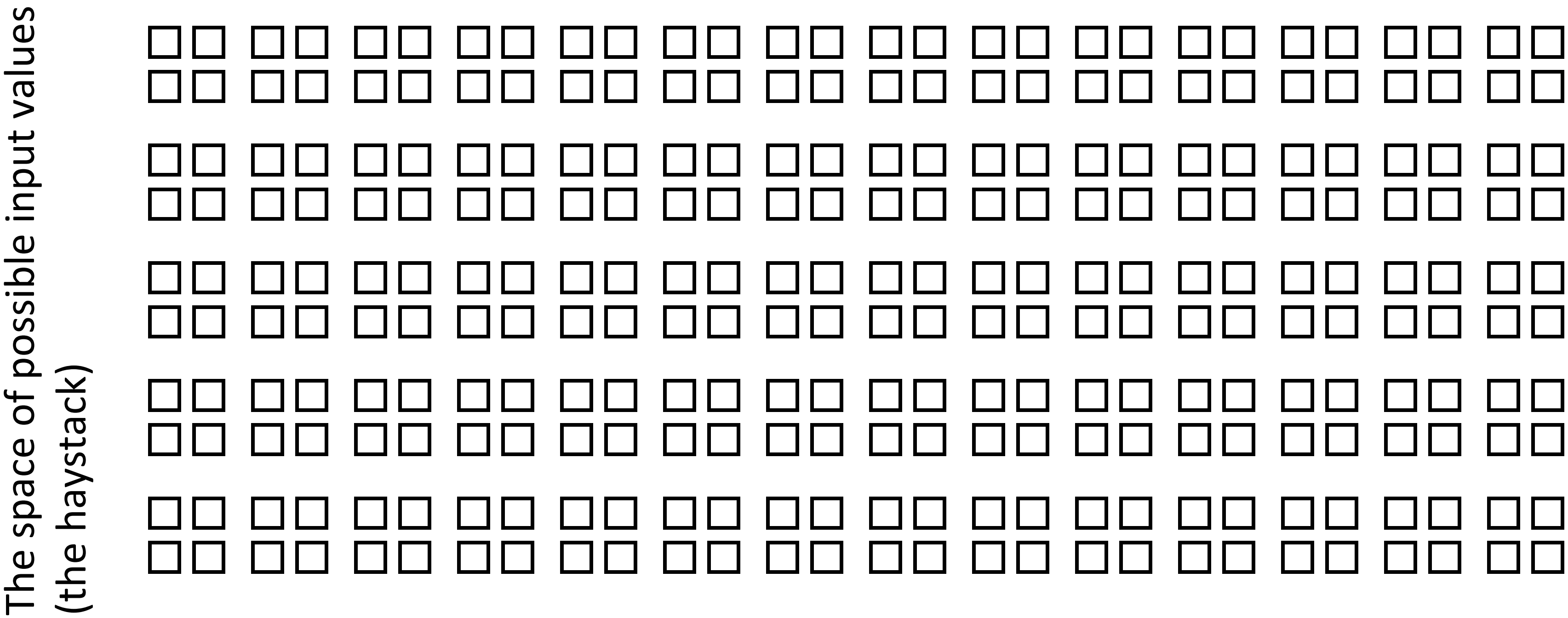
- Systematic (non-uniform):
  - Try to select inputs that are **especially valuable**
  - Usually by choosing representatives of classes that are supposed to fail often or not at all

- Functional testing is systematic testing

Systematic *Functional* Testing



Systematic Partition Testing

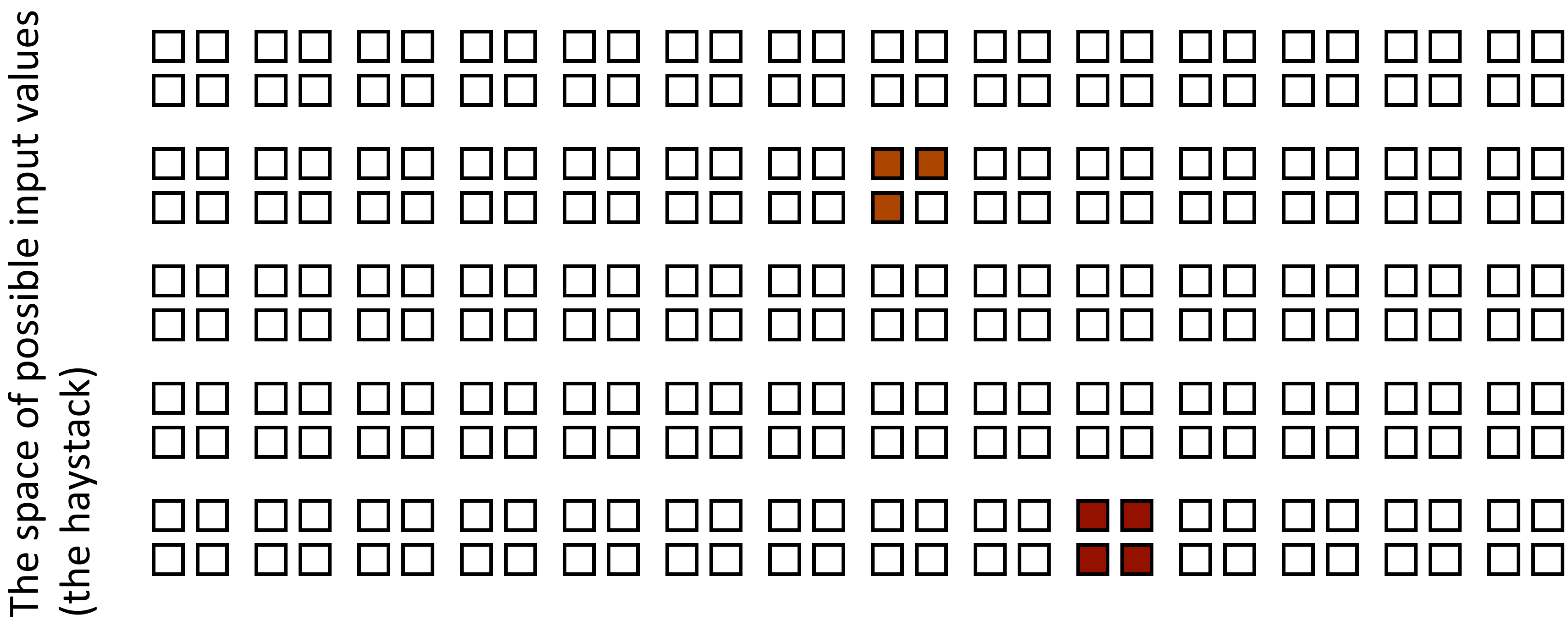


Systematic Partition Testing

10.2

■ Failure (valuable test case)

□ No failure



Systematic Partition Testing

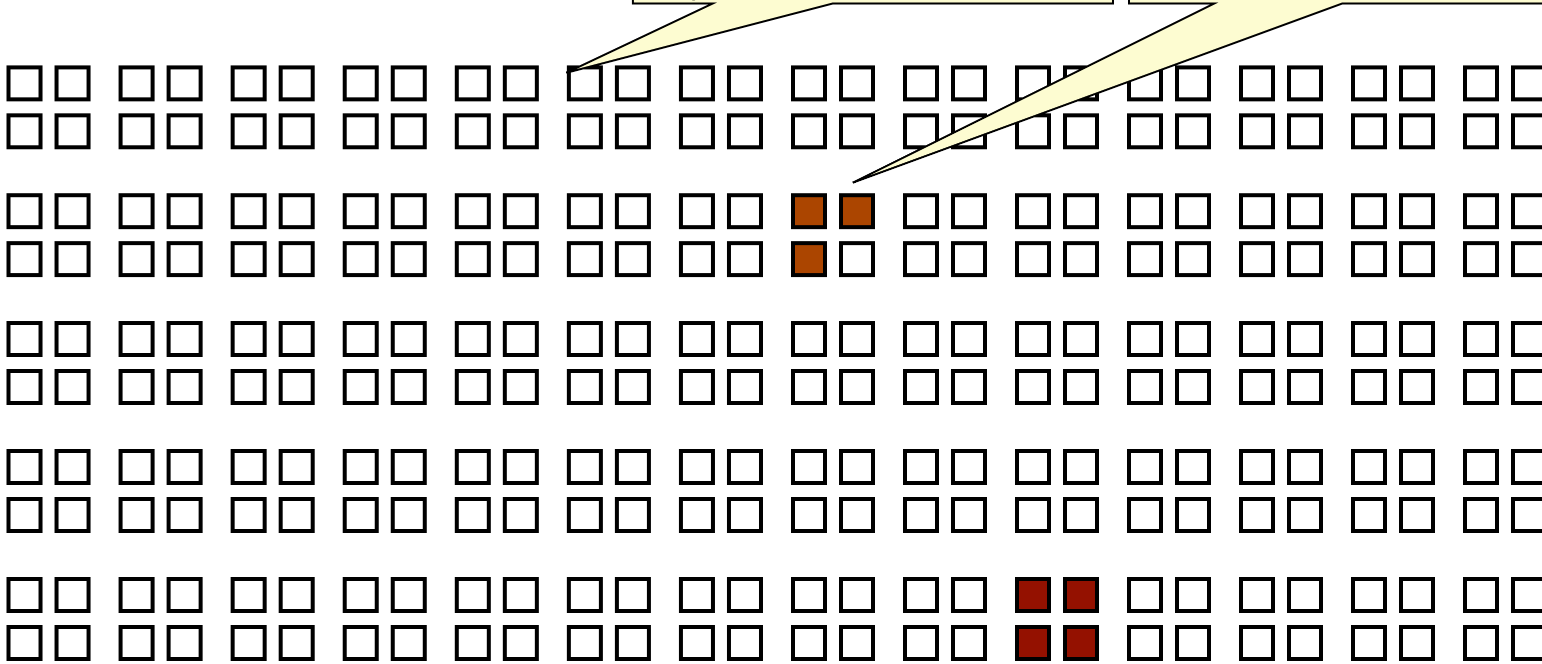
10.2

Failure (valuable test case)

No failure

Failures are sparse in the space of possible inputs ...

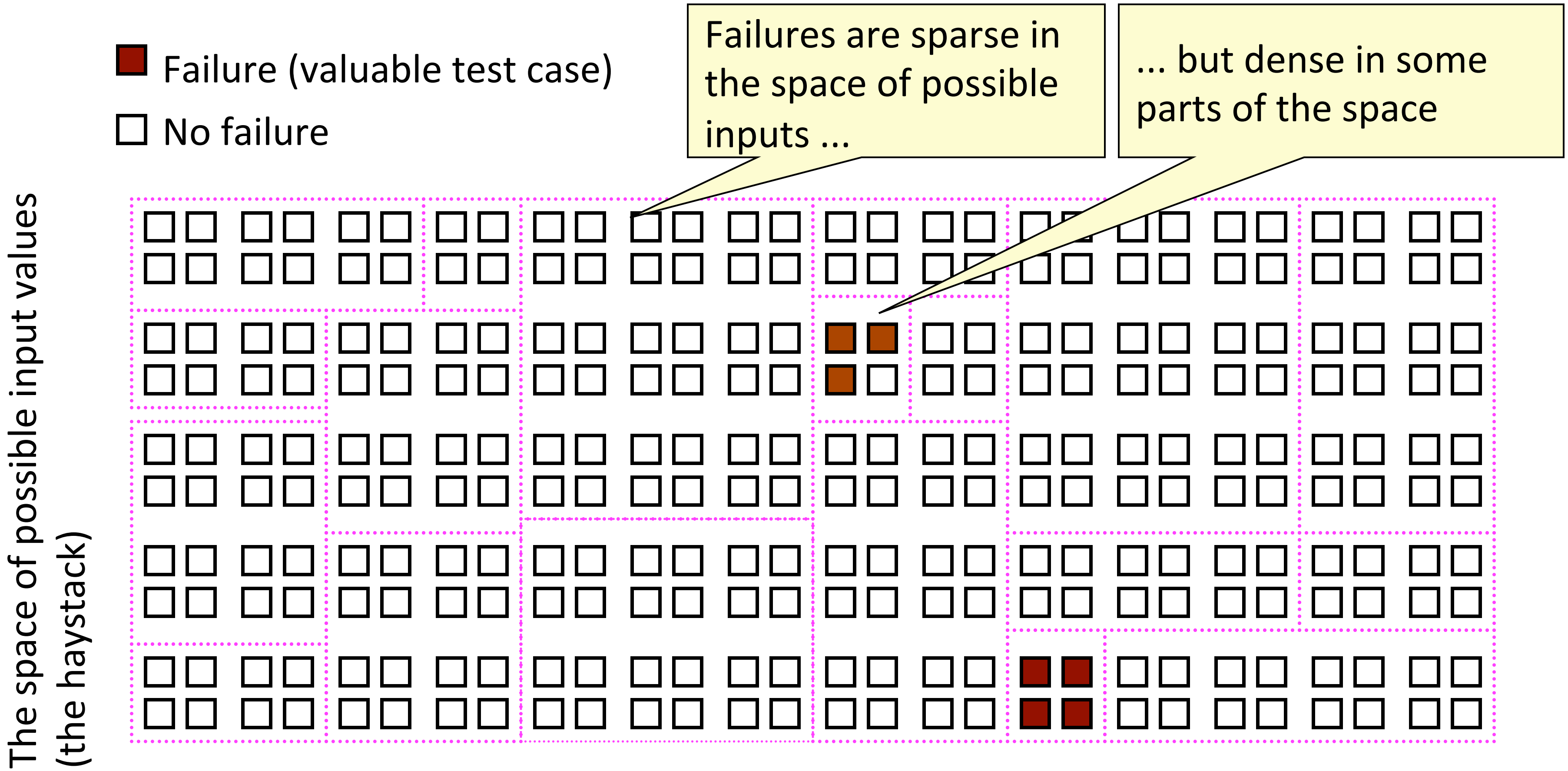
... but dense in some parts of the space



The space of possible input values (the haystack)

10.2

Partition: Equivalence Classes



Functional testing is one way of drawing pink lines to isolate regions with likely failures

10.2

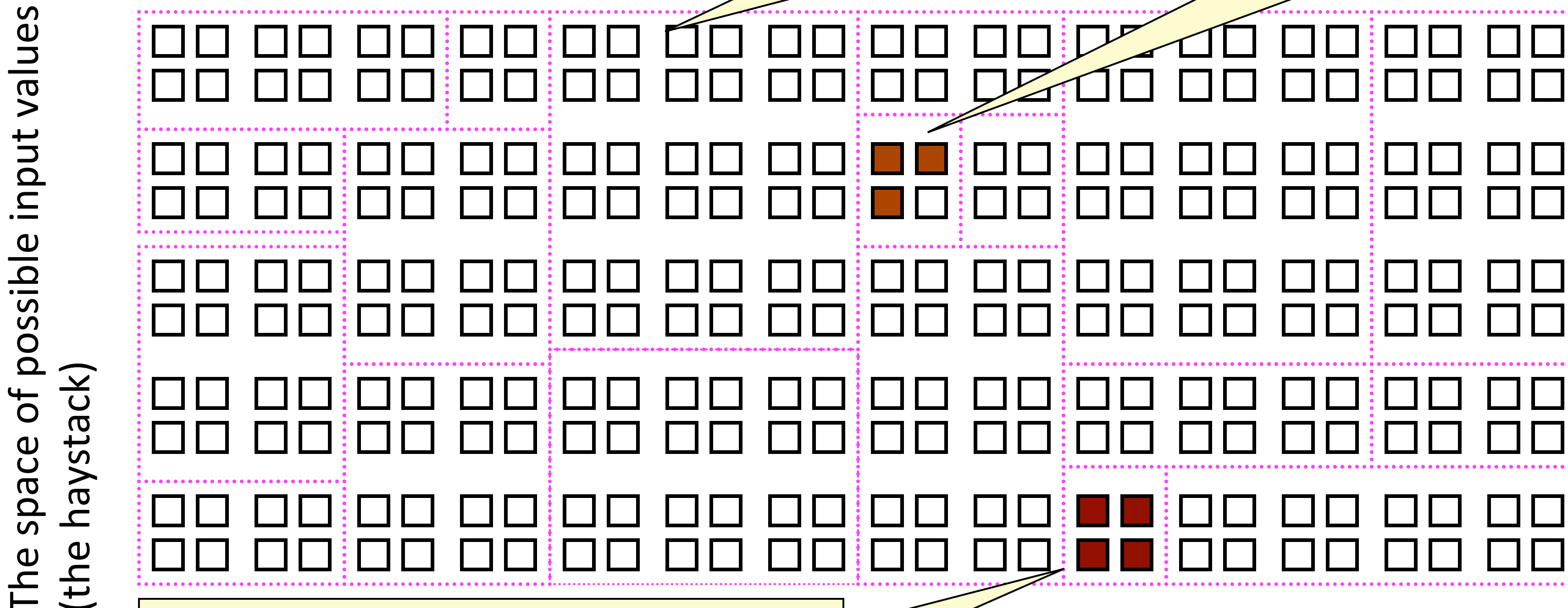
Systematic Partition Testing

Failure (valuable test case)

No failure

Failures are sparse in the space of possible inputs ...

... but dense in some parts of the space



If we systematically test some cases from each part, we will include the dense parts

Functional testing is one way of drawing pink lines to isolate regions with likely failures

Boundary Values

- X > 4   ->   X >= 4
- Y =< 2   ->   Y =< 3
- for (int l = 0; i++; i < max) { … }
- X >= 0 && X < W && Y >= 2

- ***Off by one** errors are every where!*

p.185

Boundary Values Test Strategy

- **System model**
  - Just boundary values occurring in decisions
- **Fault model**
  - Off by one
- **Test procedure**
  - Domain matrix with one on/off point per boundary
  - One to make condition true, other to make it false
  - Pick arbitrary in-points for other boundaries
- **Adequacy criterion**
  - On and off each boundary, in and out each boundary

Jeng & Weyuker.
A simplified
domain-testing strategy.
ACM TOSEM, 1994.
Also in Binder, 2000.

p.185

Boundary Value Testing Terminology

- On point: Value **on** a boundary
  - a > 3, choose the value 3 for a
- Off point: Value **not** on a boundary
  - *But as close as possible to the boundary!*
  - *a > 3, choose the values 2 or 4 for a*
- Open boundary (a > 3):
  - On point 3 -> false; Off point 4 -> true
- Closed boundary (a >= 3):
  - On point 3 -> true;   Off point 2 -> false
- In point: condition is true, neither *on* nor *off* point
  - In-point 6 -> true

Open vs Closed Boundary

- An open boundary condition is defined by a **strict inequality** operator for example x > y or z < 5.
  - The **on point** of an open boundary includes the boundary value, but makes the boundary condition **false**.
  - e.g., x > 0 is open and the on point is 0 making the condition false.
- A closed boundary condition is defined by an operation that includes **strict equality** such as a >= b.
  - An on point of a closed boundary includes the boundary value and makes the condition **true**.
  - e.g., 10.0 is the on point of (y <= 10.0). An off point of a closed boundary condition makes it false and must lie outside the domain e.g., 10.1 for (y <= 10.0)

Exercise: Domain Matrix

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
Boundary			**Test Cases**					
Variable	**Condition**	**type**	t1	t1	t3	t4	t5	t6
x	> 0	on						
		off						
	<= 10	on						
		off						
	typical	in						
y	>= 1.0	on						
		off						
	typical	in						
Expected result								

Filled With Values

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
Variable	**Condition**	**type**	t1	t1	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in					4	6
y	>= 1.0	on					1.0	
		off						0.9
	typical	in	10.0	16.0	109.3	2390.2		
Result			R	ok	ok	R	ok	R

R (reject) = false
Ok = true

One-By-One (assignment 2.5)

- The *one-by-one domain testing strategy*:
  - One off point, and one on point for each domain boundary
- Combinations of domains:
  - Essentially independent
  - Focus on one boundary, pick in points for remaining domains.
- Effective in practice
  - Only test *combinations* of boundary values if you see specific need for it