UT ECE THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

EE360T/382V Software Testing

khurshid@ece.utexas.edu

April 1, 2020

Overview

Last class – Started Input space partitioning

Today – Complete Input space partitioning

Next class – Start Syntax-based testing

Recall: Criteria based on structures

The textbook focuses on four kinds of structures to define criteria:

- Graphs (Chapter 2)
  - E.g., control-flow graphs (CFGs)
- Logical expressions (Chapter 3)
  - E.g., if-conditions
- Input domain characterization (Chapter 4)
  - E.g., sorted array
- Syntactic structures (Chapter 5)
  - E.g., mutation

EE360T/382V Software Testing

khurshid@ece.utexas.edu

Input Space Partitioning (Chapter 4)*

*Introduction to Software Testing by Ammann and Offutt

Chapter 4: Outline

Input domain modeling

- Interface-based input domain model (IDM)
- Functionality-based IDM
- Identifying characteristics
- Choosing blocks and values

Combination strategies criteria

Constraints among partitions

Background

A key part of testing is choosing elements from the space of possible inputs

Input domain – set of possible values for the parameters

Partition of a domain D – set of *blocks* $b_1, \ldots, b_n$:

- $b_i \cap b_j = \phi$, for $i \neq j$
- $\bigcup_i b_i = D$

Coverage criteria can provide a way to partition the input space according to the test requirements

- E.g., all inputs that execute the same path

Ideally, all inputs in a block are equally "useful"

Overview

This chapter considers the input space and its partitions **explicitly** – at the black-box level for code

- Chapters 2 and 3 focused at the white-box level

Input domain modeling

A common way to apply input space partitioning:

- Consider each parameter's domain separately
  - Partition each domain's values into blocks
  - Identify a representative value for each block
- Combine the representative values from domains for different parameters
  - Combination strategies defined by several criteria

Partitions are usually based on some characteristic of the program, e.g., its inputs or its environment

Illustration

Example characteristics:

- Input *x* is *null*
- Order of text file *f* that has a list of words
- Minimum separation distance of two planes

Exercise – Consider these blocks w.r.t. “order of file”:

- b_1 = sorted in ascending order
- b_2 = sorted in descending order
- b_3 = arbitrary order

Do blocks b_1, b_2, and b_3 form a partition? If not, define new blocks that do form a partition

Input domain modeling

1. Identify functionality, e.g., method, under test
2. Identify its inputs
3. Create an input domain model (IDM)
   - Scope input domain based on inputs
   - Define structure of input domain using input characteristics
   - Create a partition w.r.t. each characteristic
4. Apply a test criterion to form block combinations
   - Take one block for each characteristic at a time
5. Reify block combinations into test inputs
   - Choose appropriate values for the blocks

Two approaches to build IDM

Interface-based IDM

- Considers each parameter in isolation at the interface level
- +: Easy to define characteristics and create tests
- -: Ignores parameter relationships

Functionality-based IDM

- Considers characteristics based on intended functionality to incorporate domain knowledge
- +: Allows utilizing requirements and specs in IDM
  - Tests can be designed early in development
- -: Identifying characteristics can be cumbersome
  - Test generation can require complex analysis

Example of each approach

Consider a triangle-classification program

```
static int Triang(int Side1, int Side2, int Side3);
// returns 1 if scalene, 2 if isosceles, 3 if
// equilateral, and 4 if not a triangle
```

Interface-based IDM – each parameter has the same type

- Can use the same characteristic for each parameter, e.g., “relation of side to 0”

Functionality-based IDM – parameters are lengths of the sides of a triangle

- IDM can combine all parameters, e.g., use characteristic “type of triangle”

How to identify characteristics?

Consider relation of parameter with special values, e.g., 0, null, or “”

Use preconditions and postconditions

Consider aliasing possibilities

- E.g., c.removeAll(d);

Utilize domain knowledge in defining characteristics

- Focus is not on structure/details of source-code (that are used in graph/logic-based criteria)

Characteristics example

```
static boolean contains(List l, Object e);
// returns true if and only if l contains element e
```

Interface-based

- l is null (2 blocks: true, false)
- e is null (2 blocks: true, false)

Functionality-based

- Number of occurrences of e in l (3 blocks: 0, 1, >1)
- e is first element in l (2 blocks: true, false)
- e == l (2 blocks: true, false)

Choosing blocks and values

More blocks (per characteristic) mean more tests

- More time to create and run tests

Fewer blocks may reduce test quality

Values in each block may be selected in various ways

- Valid, invalid, special, “normal use” values
- Domain boundaries
- Sub-partition blocks as needed
- Validate the partition (complete, disjoint blocks)

Example blocks and values: interface-based IDM

```
static int Triang(int Side1, int Side2, int Side3);
```

Charecteristic	b_1	b_2	b_3
Relation of Side 1 to 0	> 0	== 0	< 0
Relation of Side 2 to 0	> 0	== 0	< 0
Relation of Side 3 to 0	> 0	==0	< 0

Using one value from each block gives at most 3^3 tests

Some block combinations represent (in)valid triangles

Can refine this categorization for finer granularity

Example finer granularity

Alternative interface-based IDM

Characteristic	b_1	b_2	b_3	b_4
Length of Side 1	> 1	== 1	== 0	< 0
Length of Side 2	> 1	== 1	== 0	< 0
Length of Side 3	> 1	== 1	== 0	< 0

Using one value from each block gives at most 4^3 tests

This partition is valid since inputs are integers

- E.g., no side has 0 < length < 1

Example possible values

boundaries

Length of Side 1	2	1	0	-1

Example blocks and values: functionality-based IDM

A domain-specific characterization could use the fact that the program is a triangle classifier

Characteristic	b_1	b_2	b_3	b_4
Geometric classification	scalene	isosceles	equilateral	invalid

Not quite right…

Refine characterization to make it valid

Characteristic	b_1	b_2	b_3	b_4
Geometric classification	scalene	isosceles, not equilateral	equilateral	invalid

Example possible values

Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Another example IDM

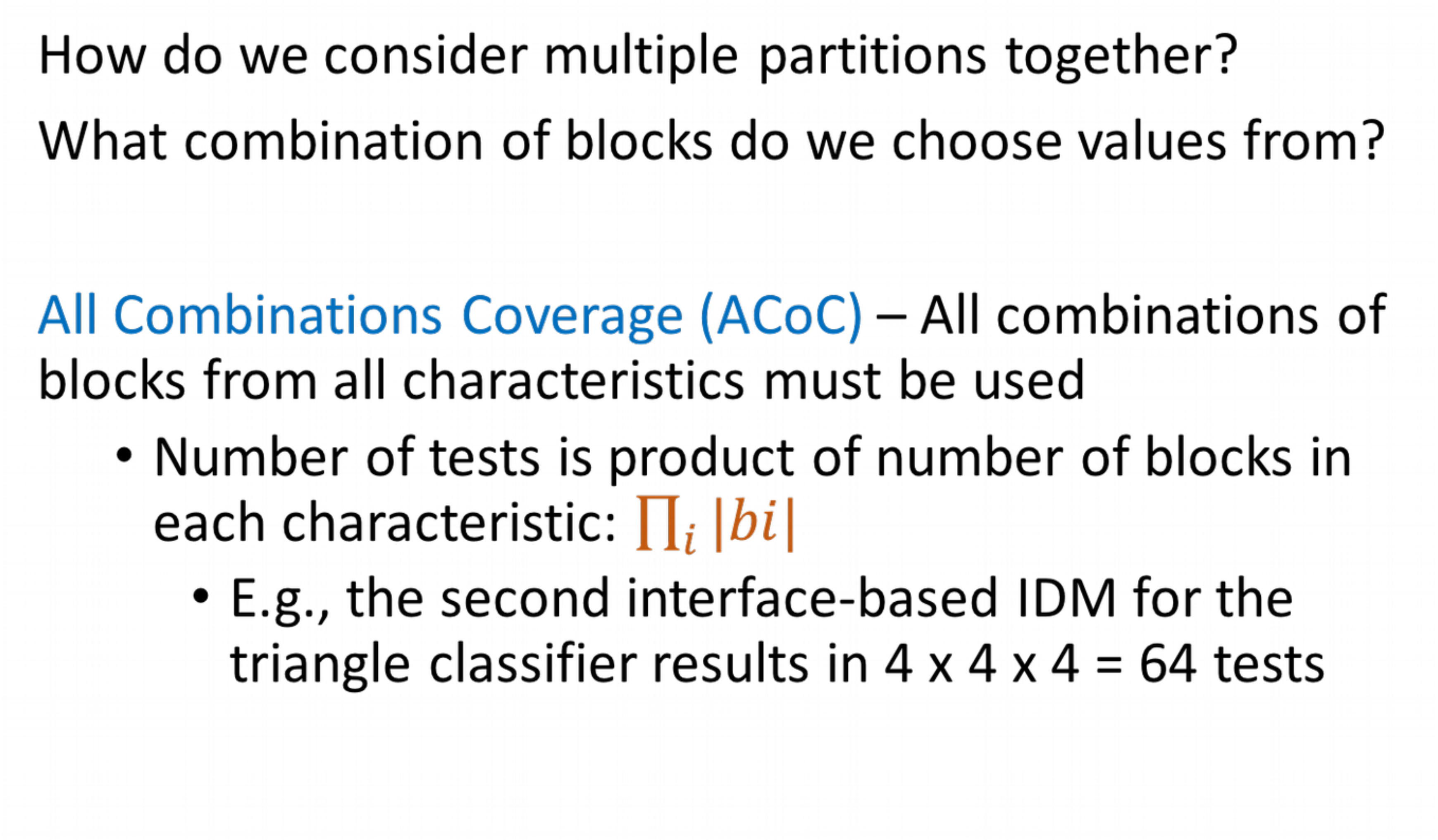
Break the geometric characterization into four separate characteristics

Characteristic	b_1	b_2
scalene	true	false
isosceles	true	false
equilateral	true	false
valid	true	false

Need *constraints* to ensure that

- equilateral = true implies isosceles = true
- valid = false implies
  scalene = isosceles = equilateral = false

Combination strategies criteria



How do we consider multiple partitions together?

What combination of blocks do we choose values from?

All Combinations Coverage (ACoC) – All combinations of blocks from all characteristics must be used

- Number of tests is product of number of blocks in each characteristic: $\prod_i |bi|$
  - E.g., the second interface-based IDM for the triangle classifier results in 4 x 4 x 4 = 64 tests

Each Choice Coverage (ECC)

One value from each block for each characteristic must be used in at least one test case

Number of tests is at least the number of blocks in the largest characteristic: $\text{Max}_i\ |b_i|$

Example tests for the triangle classifier:

(2, 2, 2)
(1, 1, 1)
(0, 0, 0)
(-1, -1, -1)

Pair-Wise Coverage (PWC)

A value from each block for each characteristic must be combined with a value from every block for each other characteristic

Number of tests is at least the product of two largest characteristics: $\text{Max}_i \, |b_i| \text{ x } \text{Max}_{j \,!= i} \, |b_j|$

Example tests for triangle classifier:

2, 2, 2	2, 1, 1	2, 0, 0	2, -1, -1
1, 2, 1	1, 1, 0	1, 0, -1	1, -1, 2
0, 2, 0	0, 1, -1	0, 0, 2	0, -1, 1
-1, 2, -1	-1, 1, 2	-1, 0, 1	-1, -1, 0

T-Wise Coverage (TWC)

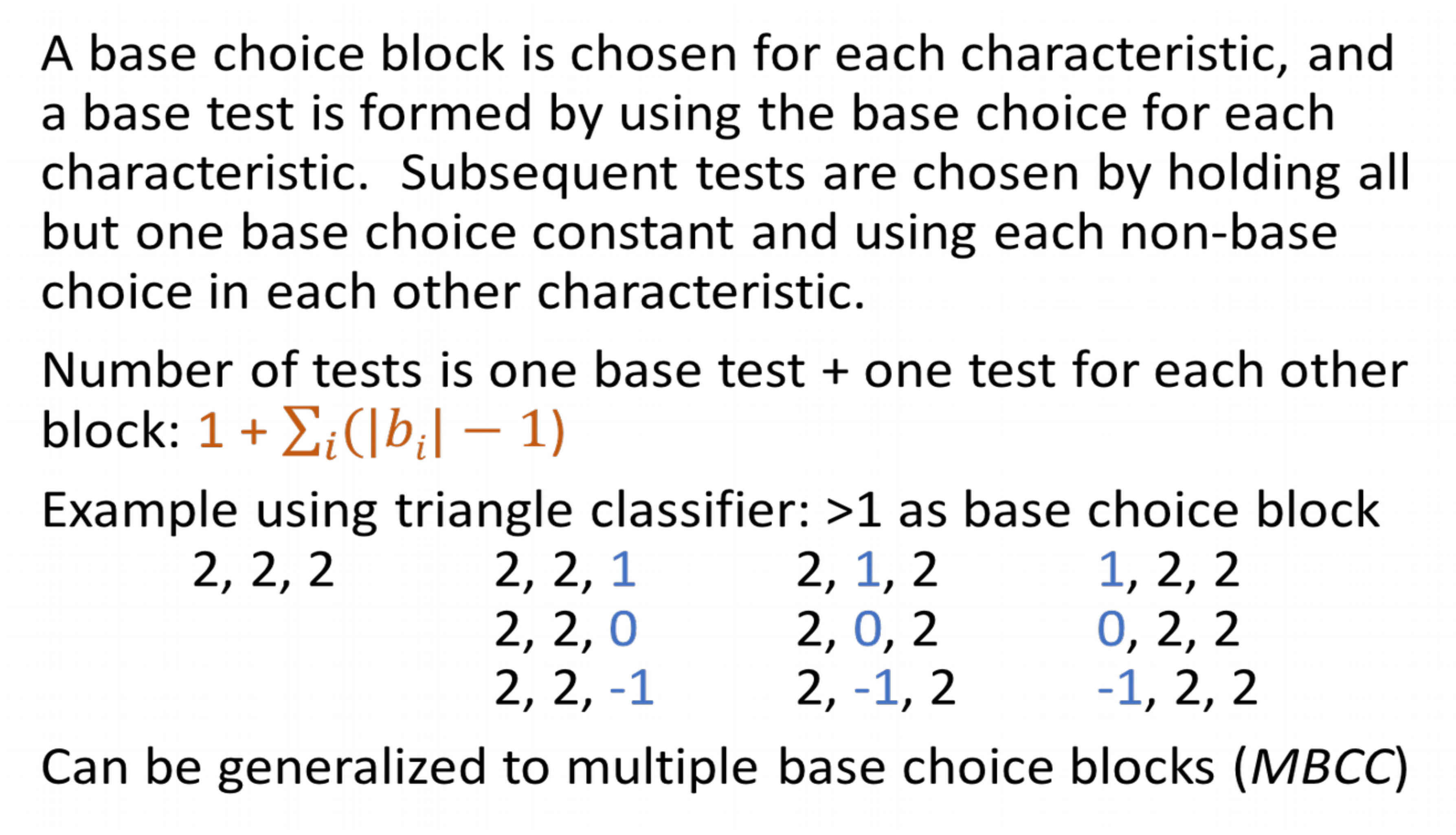
A value from each block for each group of *t* characteristics must be combined

If all characteristics are the same size, number of tests is at least $(\mathrm{Max}_i |b_i|)^t$

If *t* is the number of characteristics, TWC is equivalent to all combinations coverage (ACoC)

Benefits of *t*-wise (over pair-wise) are not clear

Base Choice Coverage (BCC)



A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

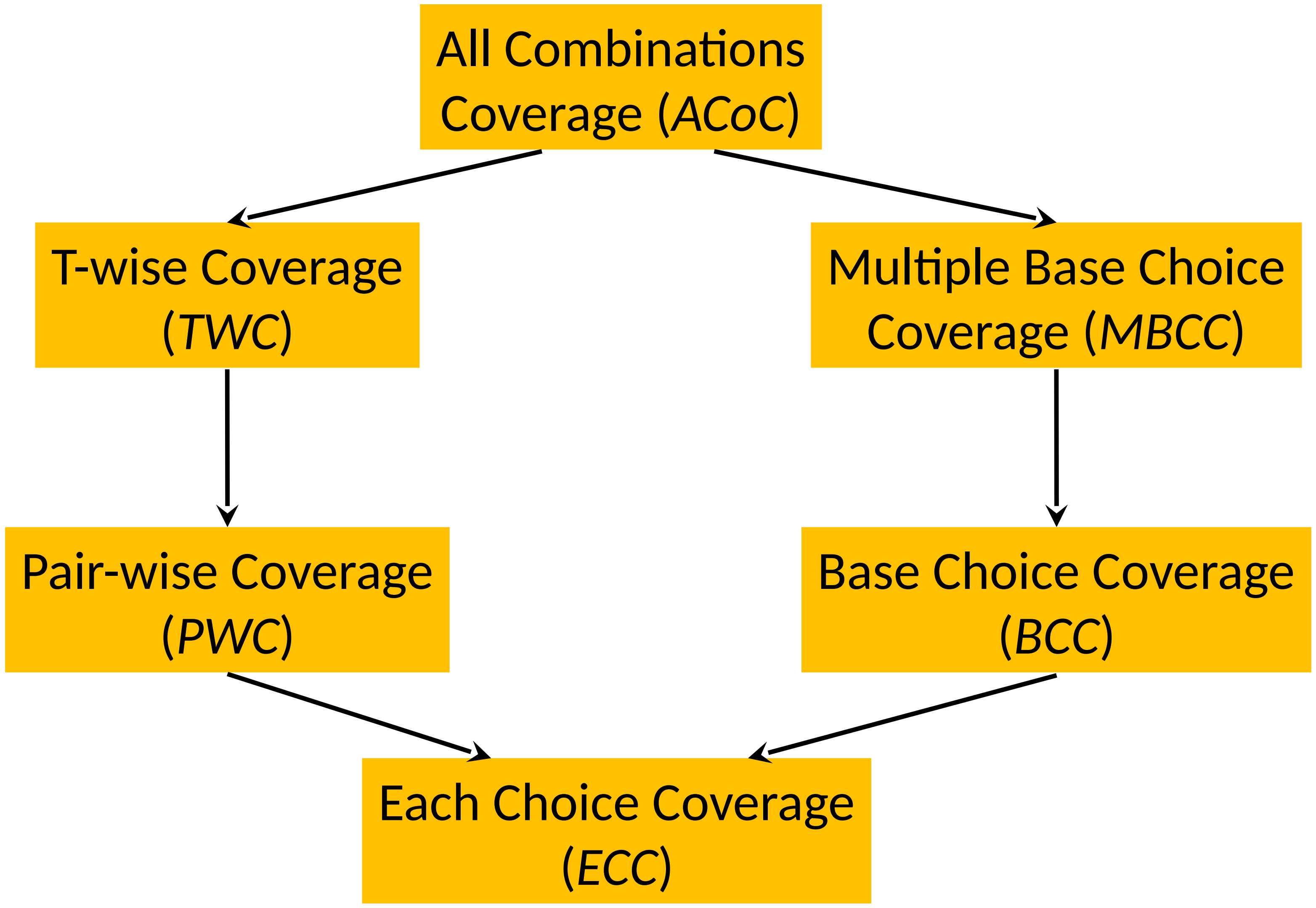
Number of tests is one base test + one test for each other block: $1 + \sum_i(|b_i| - 1)$

Example using triangle classifier: >1 as base choice block

2, 2, 2	2, 2, 1	2, 1, 2	1, 2, 2
	2, 2, 0	2, 0, 2	0, 2, 2
	2, 2, -1	2, -1, 2	-1, 2, 2

Can be generalized to multiple base choice blocks (*MBCC*)

Subsumption


All Combinations Coverage (*ACoC*)
T-wise Coverage (*TWC*)
Multiple Base Choice Coverage (*MBCC*)
Pair-wise Coverage (*PWC*)
Base Choice Coverage (*BCC*)
Each Choice Coverage (*ECC*)

Constraints among partitions

Some block combinations may be infeasible

- E.g., valid = false and scalene = true not possible

Constraints capture feasibility properties

Two basic types of constraints

- A block from one characteristic **cannot** be combined with a specific other block
- A block from one characteristic **can only** be combined with a specific other block

Handling constraints depends on specific criteria

- *ACoC*, *PWC*, *TWC*: drop infeasible combinations
- *BCC*, *MBCC*: change a value to another non-base choice to try to find a feasible combination

Example constraint handling

Sorting an array

Input: array with elements of some arbitrary type

Output: array in sorted order, largest, smallest value

Characteristics:

- Length of
- Type of el
- Max value
- Min value
- Position o
- Position o

Blocks from other characteristics are irrelevant

Blocks must be combined

Partitions:

- Len { 0, 1, 2..100, 101..MAXINT }
- Type { int, char, string, other }
- Max { ≤0, 1, >1, 'a', 'Z', 'b', …, 'Y' }
- Min { … }
- Max Pos { 1, 2 .. Len-1, Len }
- Min Pos { 1, 2 .. Len-1, Len }

Applying input domain modeling to an example recursive structure

Recall our singly linked-list example:

```
public class SLList { // invariant: acyclicity
    Node header;
    int size;

    static class Node {
        int elem;
        Node next;
    }

    int removeFirst() {
        // precondition: header != null
        // postcondition: returns the element in the first
        //   node and removes that node if header != null;
        //   else, throws NullPointerException
       ... }
}
```

One way to form characteristics and apply coverage criteria

Observe – *removeFirst*'s behavior depends on **one input**: receiver object, which is a list

- Note: empty list violates method precondition

Consider properties of lists (based on type declaration or functionality) to form characteristics

1. List has repetitions
   - 2 blocks: T; F
2. Length of list
   - 3 blocks: length == 1; length == 2; length > 2

Can apply combination strategies criteria to blocks for these two characteristics

Another way to form characteristics and apply coverage criteria

Observe – *removeFirst*'s behavior depends on **values of object fields reachable from *this***

- Each reachable object field is effectively an input

Consider properties of fields of *this* to define characteristics

1. Value of *this.header* (assume non-null)
   - 2 blocks
     - *header.next* == *header*; *header.next* != *header*
2. Value of *this.size* (assume non-zero)
   - 3 blocks: 1; 2; >2

Again, can apply combination strategies criteria

?/!