Intro to Software Testing

chapter 6

Input Space Coverage

Dr. Brittany Johnson-Matthews
(Dr. B for short)

https://go.gmu.edu/SWE637

Adapted from slides by Jeff Offutt and Bob Kurtz

More software in the news

NASA finds 'fundamental' software problems in Boeing's Starliner spacecraft

Investigators probing the botched flight of Boeing's Starliner spacecraft in December have found widespread and "fundamental" problems with the company's software that could have led to a disastrous outcome more grievous than previously known, the agency said Friday.

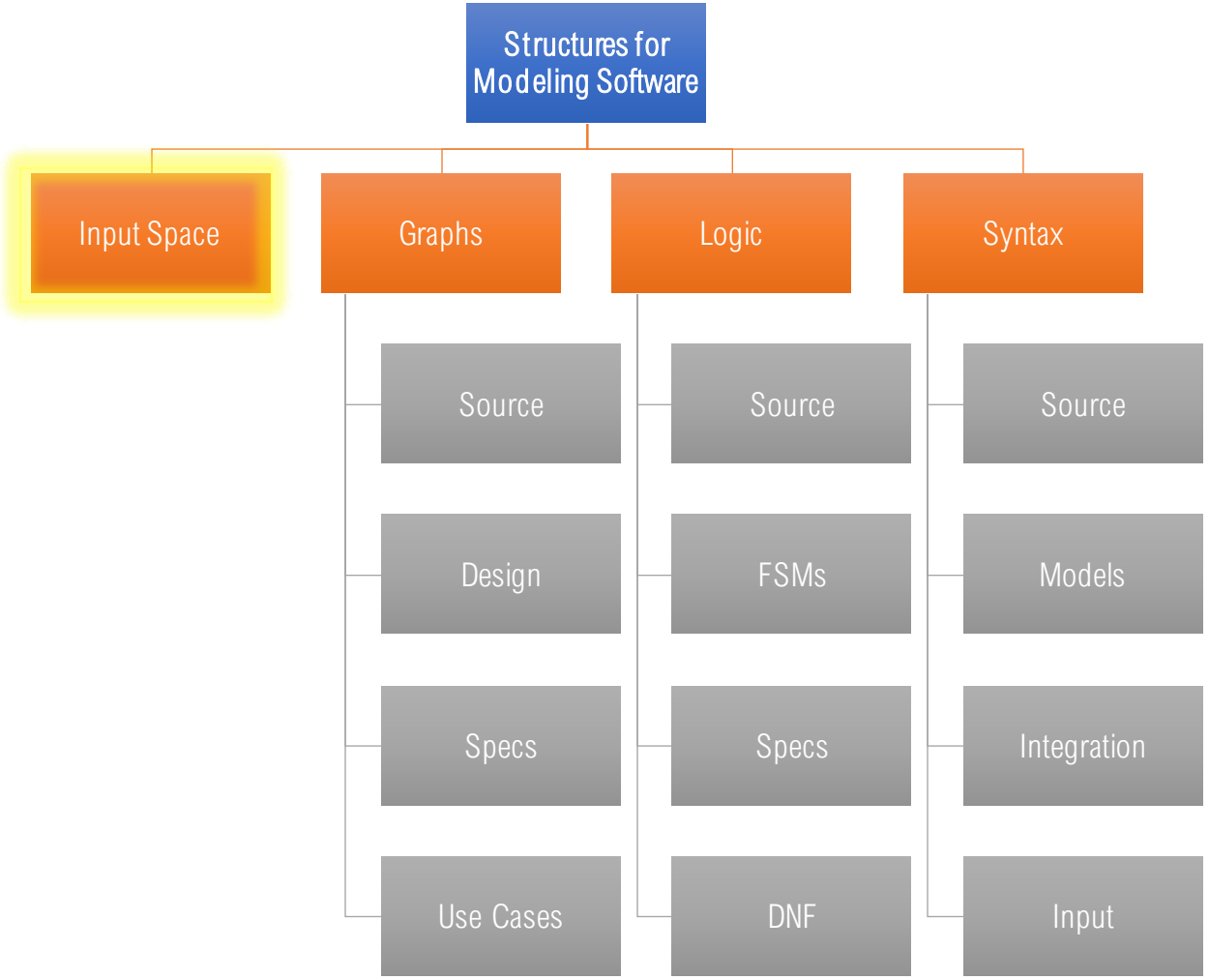
Boeing is now reviewing all 1 million lines of code in the capsule's computer systems, officials said. How long that review will take is uncertain, Boeing officials said. […]

"We don't know how many software errors we have — if we have just two or many hundreds," Loverro said. In an interview, he added that the "***bottom line is that industry is very bad at doing software***."

-Washington Post, 2/7/2020

Boeing added a second unmanned test launch to the schedule as a result of these problems. As of September 2021, that second launch has not occurred.

Input Space Coverage



Benefits of ISP

Can be *equally applied* at several levels

- Unit testing
- Integration testing
- System testing

Relatively easy to apply *without automation*

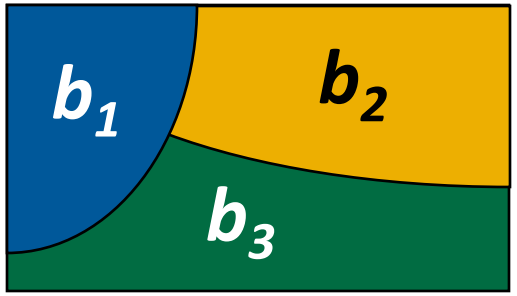
Easy to *adjust* to get more or fewer tests

No *implementation knowledge* is needed, just an understanding of the input domain (“black box”)

Partitioning Domains

Given domain *D*, there is a partition scheme *q* of *D* such that:

- Partition *q* defines a set of blocks
  Bq = $b_1, b_2, \ldots, bQ$
- The partition must satisfy two properties
  - Blocks must be ***disjoint*** (no overlaps)
  - Blocks must be ***complete*** (cover the domain D)



Partitioning Assumptions

Choose a *value* from each block

All possible values in a block are assumed to be *equally useful* for testing

That's not just a coincidence, it's why we construct the blocks the way that we do

Application to testing

Find *characteristics* in the inputs: parameters, semantic descriptions, etc.

Choose tests by combining values from characteristics

Example characteristics:

- Input X is null
- Order of the input file F (sorted, reverse sorted, random, etc.)
- Input device (DVD, CD, HDMI, etc.)

Choosing Partitions

Choosing (defining) partitions seems easy, but it's easy to get wrong

Consider the *order of file F*

- b1 = sorted in ascending order
- b2 = sorted in descending order
- b3 = random order

But what about a file of length 1?

- The file will be in all three blocks, and the **disjointness** property will not be satisfied

Properties of Partitions

If the partitions are not complete and disjoint, then they haven't been considered carefully enough

Like any design, partitioning should be reviewed carefully and different alternatives should be considered

Modeling the Input Domain

Step 1/5: Identify the *testable functions.*

- Individual *methods* have one testable function
- Methods in a *class* often have the same characteristics
- *Programs* have more complicated characteristics, modeling documents like UML can be used to design characteristics
- *Systems* of integrated hardware and software components can have many testable functions – devices, operating systems, hardware platforms, browsers, etc.

Modeling the Input Domain

Step 2/5: Find all the *parameters*

- Often straightforward or mechanical
  - Preconditions and postconditions
  - Relationships among variables
  - Special values (zero, null, etc.)
- Do not use program source code, characteristics should be based on the *input domain*
- *Methods*: parameters and state variables
- *Components*: parameters to methods and state variables
- *Systems*: all inputs, including files and databases

Modeling the Input Domain

Step 3/5: Model the *input domain*

- The domain is scoped by the *parameters*
- The structure is defined by *characteristics*
- Each characteristic is partitioned into *sets of blocks*
- Each block represents a *set of values*
- This is the most creative design step in ISP
  - Better to have more characteristics and fewer blocks; leads to fewer tests
  - Strategies include valid/invalid/special values, boundary values, “normal” values

Modeling the Input Domain

Step 4/5: Apply a *test criterion* to choose *combinations* of values

- A test input has *one value* for each parameter
- There is *one block* for each characteristic
- Choosing *all combinations* is usually infeasible
  - Coverage criteria allow subsets to be chosen

Modeling the Input Domain

Step 5/5: Refine combinations of blocks into *test inputs*

- Choose *appropriate values* for each block
- Combinatorial test optimization tools can help
  - These tools dramatically reduce the number of tests

Two Approaches to IDM

Interface-based approach

- Develop characteristics directly from *individual input parameters*
- Simplest application of IDM
- Can often be automated, at least partially

Functionality-based approach

- Develop characteristics from a *behavioral view* of the program under test
- Harder to develop and requires more design effort
- May result in better tests, or fewer tests that are just as effective

Interface-Based Approach

Mechanically consider each parameter in isolation

This is a simple technique based on syntax

Some domain and semantic information won't be used, which can lead to an incomplete IDM

Ignore relationships between parameters

Interface-Based Example

Consider the TriangleType.triang() method

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid };

// side1, side2, and side3 represent the lengths of the sides,
// returns corresponding Triangle enum value
public static Triangle triang (int side1, int side2, int side3) {
... };
```

The IDM for each parameter is identical because the types and semantics are identical

A candidate characteristic: “relation of side to zero”

Interface-Based Example

Characteristic	b_1	b_2	b_3
q_1 = "relation of side 1 to zero"	> 0	= 0	< 0
q_2 = "relation of side 2 to zero"	> 0	= 0	< 0
q_3 = "relation of side 3 to zero"	> 0	= 0	< 0

- Maximum 3*3*3 = 27 tests – (1,1,1), (1,1,0), (1,1,-1), (1,0,1), …
- Most of the resulting triangles are invalid – maybe a shortcoming
- We can refine the characterization (with special case bn=1) to get more tests with more valid triangles…

Characteristic	b_1	b_2	b_3	b_4
q_1 = "length of side 1"	> 1	= 1	= 0	< 0
q_2 = "length of side 2"	> 1	= 1	= 0	< 0
q_3 = "length of side 3"	> 1	= 1	= 0	< 0

Now we have 4*4*4 = 64 tests – (2,2,2), (2,2,1), (2,2,0), (2,2,-1), …

Functionality-Based Approach

Identify characteristics that correspond to the intended functionality

Requires more design effort from the tester

Can incorporate domain and semantic knowledge

Can use relationships between parameters

Modeling can be based on requirements, not implementation

The same parameter may appear in multiple characteristics, making it harder to translate values into test cases

Functionality-Based Example

Consider `TriangleType.triang()` again

All three parameters together represent a triangle

The IDM can combine all parameters

- A candidate characteristic: “type of triangle”

Functionality-Based Example

A semantic characterization could use the fact that the three integers represent a triangle

Characteristic	b_1	b_2	b_3	b_4
q_1 = "triangle classification"	scalene	isosceles	equilateral	invalid

But there's a problem here – equilateral triangles are also isosceles, so the blocks are not disjoint; let's reconsider

Characteristic	b_1	b_2	b_3	b_4
q_1 = "triangle classification"	scalene	isosceles but not equilatera l	equilateral	invalid

This results in only 4 tests – among the infinitely many possible values are (4,5,6), (3,3,4), (3,3,3), (3,4,99)

Functionality-Based Example

A different approach would be to break the single geometric characterization into four separate characterizations

Characteristic	b_1	b_2
q_1 = "scalene"	True	False
q_2 = "isosceles"	True	False
q_3 = "equilateral"	True	False
q_4 = "valid"	True	False

We can then apply constraints to ensure that

- Equilateral = true --> isosceles = true
- Valid = false --> scalene = isosceles = equilateral = false

For this IDM, this approach is probably not beneficial

Multiple IDMs

Some programs have tens or hundreds of parameters.

Create several small IDMs

- Different parts of the software can be tested with different amounts of rigor using different IDMs
- It's okay if the IDMs overlap; the same variable may appear in more than one IDM

ACoC Criterion for Choosing Values

We can use criteria to choose effective subsets of values

The most obvious criterion is to choose all combinations

DEFINITION

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

Number of ACoC TRs

The number of test requirements is the product of the number of blocks in each characteristic.

$\prod_{i=1}^{Q} B_i$, where:

- Q is the number of characteristics
- B_i is the number of blocks in characteristic i

For `triang()` using length of each side (>1, 1, 0, <0), this is 4*4*4 = 64... too many?

- And only 8 of the 64 are valid combinations with side >0, which hints this may not be very good partitioning

ACOC Example

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

TR = { (a1, b1, c1), (a1, b1, c2),
(a1, b2, c1), (a1, b2, c2),
(a2, b1, c1), (a2, b1, c2),
(a2, b2, c1), (a2, b2, c2),
(a3, b1, c1), (a3, b1, c2),
(a3, b2, c1), (a3, b2, c2) }

ECC Criterion for Choosing Values

We can reduce the number of combinations by taking one value from each block

DEFINITION

Each Choice Coverage (ECC) – one value from each characteristic must be used in at least one test

Number of ECC TRs

The number of test requirements is the number of blocks in the largest characteristic

For **triang()** using length of each side (>1, 1, 0, <0), this is 4… too few?

Example values:
(2,2,2), (1,1,1), (0,0,0), (-1,-1,-1)

Do these look like especially effective tests?

ECC Example

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

TR = { (a1, b1, c1),
(a2, b2, c2),
(a3, b2, c1) }

Since all values of B and C have already been used, we can randomly select values to use here.

PWC Criterion for Choosing Values

We can combine values from one block with values from other blocks

DEFINITION

Pair-Wise Coverage (PWC) – a value from each block for each characteristic must be combined with a value from each block of every other characteristic

Number of PWC TRs

The number of test requirements is at least the product of the two largest characteristics

- For **triang()** using length of each side (>1, 1, 0, <0), this is 4*4 = 16 … perhaps just right?
- Example values: (2,2,2), (2,1,1), (2,0,0), (2,-1,-1), (1,2,2), (1,1,1), (1,0,0), (1,-1,-1), …
  - At a glance, this looks like a more varied and interesting set of tests, though many will still be invalid

PWC Example

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

TR = { (a1, b1, c*), (a1, b2, c*),
(a1, b*, c1), (a1, b*, c2),
(a2, b1, c*), (a2, b2, c*),
(a2, b*, c1), (a2, b*, c2),
(a3, b1, c*), (a3, b2, c*),
(a3, b*, c1), (a3, b*, c2),
(a*, b1, c1), (a*, b1, c2),
(a*, b2, c1), (a*, b2, c2) }

We can satisfy all these TRs with optimized combinations:
TR = { (a1, b1, c1),
(a1, b2, c2),
(a2, b2, c1),
(a2, b1, c2),
(a3, b1, c2),
(a3, b2, c1) }
(other combinations are possible)

TWC Criterion for Choosing Values

An extension of PWC is to require combinations of *t* values instead of 2

DEFINITION

_t_-Wise Coverage (TWC) – a value from each block for each group of *t* characteristics must be combined

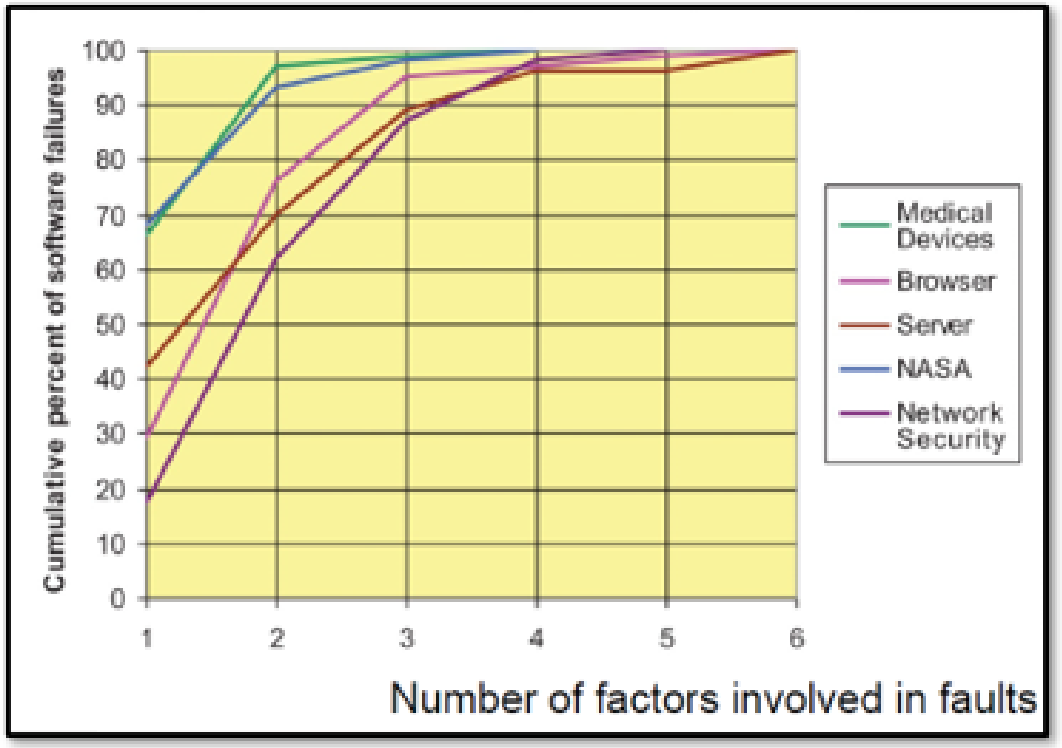
Number of TWC TRs

The number of test requirements is at least the product of the *t* largest characteristics

- If *t* is equal to the number of characteristics, as it is with the `triang()` example, then *t*-wise is equivalent to ACoC

TWC Criterion for Choosing Values

Fault detection increases as *t* increases, but with diminishing returns (while the number of tests increases dramatically)



https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=910001

BCC Criterion for Choosing Values

Use *domain knowledge* of the program to identify important values

DEFINITION

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 BCC TRs

The number of tests in one *base test* plus one test for each other block

$$1 + \sum_{i=1}^{Q} (Bi - 1)$$

where:

- Q is the number of characteristics
- B_i is the number of blocks in characteristic i

For `triang()` using length of each side (>1, 1, 0, <0), this is 10

- Examples: (2,2,2), (2,2,1), (2,2,0), (2,2,-1), (2,1,2), (2,0,2), (2,-1,2), (1,2,2), (0,2,2), (-1,2,2)

BCC Criterion for Choosing Values

The base test must be *feasible,* that is, all values in the base choice must be compatible

Base choices can be:

- The most likely or most common values
- The simplest values
- The smallest values
- The first values in some logical ordering

Happy path tests make good base choices

The base choice is a *crucial design decision*

- Test designers should document why the base choice was selected
- A poor base choice can result in many infeasible combinations

BCC Example

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

Base choices

TR = { **(a1, b1, c1)**, Base test
(a2, b1, c1),
(a3, b1, c1), } Variations on A
(a1, **b2**, c1), Variation on B
(a1, b1, **c2**) } Variation on C

MBCC Criterion for Choosing Values

There can sometimes be more than one logical base choice for each characteristic

DEFINITION

Multiple Base Choice Coverage (MBCC) – at least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

Number of MBCC TRs

The number of tests is no more than

$$M + \sum_{i=1}^{Q} (M * (Bi - m_i))$$

where:

- Q is the number of characteristics
- M is the number of base tests
- B_i is the number of blocks in characteristic i
- m_i is the number of base choices for characteristic i

Informally, the process is:

- Pick a base choice from each characteristic to create a base test
- Vary the characteristic one at a time, but skip other choices for the characteristic that are also base choices
- Repeat for next set of base choices

MBCC Example

Multiple base choices

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

TR = { **(a1, b1, c1)**, Base choice #1

(a2, b1, c1), Variations on A

~~**(a3**, b1, c1)~~,

(a1, **b2**, c1), Variation on B

~~(a1, b1, **c2**)~~, Variation on C

(a3, b1, c2), Base choice #2

~~**(a1**, b1, c2)~~, Variations on A

(**a2**, b1, c2), Variation on B

(a3, **b2**, c2), Variation on C

~~(a3, b1, **c1**)~~ }

Substituting a3 in place of a1 is not necessary because a3 is also a base choice and will show up in a later TR

Constraints Among Characteristics

Some combinations are **infeasible**

- Can't have "less than zero" and "scalene"

This is represented as **constraints**

Two general types of constraints

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

Handling constraints depends on the criterion used

- ACC, PWC, TWC – drop the infeasible pairs
- BCC, MBCC – change a value to another non-base choice to find a feasible combination

Constraints Example

```
public boolean findElement (List list, Object element) {
   // Effects: if list or element is null throw NullPointerException
   //          else element is in list return true
   //          else return false
   ...
}
```

Characteristic	b_1	b_2	b_3	b_4	b_5	b_6
A: size and contents	list=null	size=0	size=1	size>1 varied unsorted	size>1 varied sorted	size>1 all same
B: match	Element not found	Element found once	Element found more than once	--	--	--
Infeasible combinations: (Ab_1, Bb_2), (Ab_1, Bb_3), (Ab_2, Bb_2), (Ab_2, Bb_3), (Ab_3, Bb_3), (Ab_6,Bb_2)						

Element cannot be in a null list once (or more than once)

Element cannot be in a 0-element list once (or more than once)

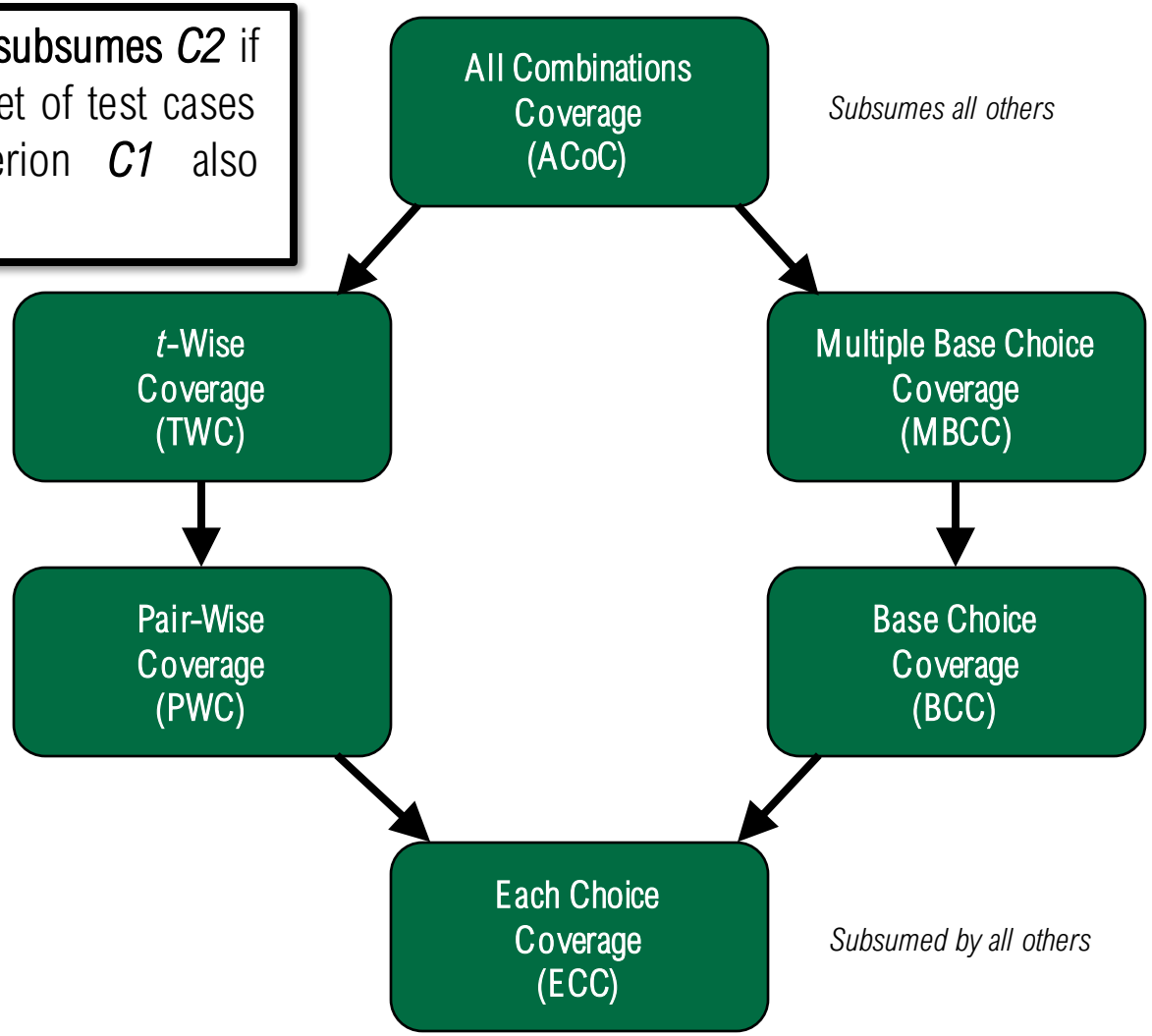
Element cannot be in a 1-element list more than once

If a list has many of the same element, we can't find it just once

ISP Criteria Subsumption

DEFINITION

A test criterion *C1* **subsumes** *C2* if and only if every set of test cases that satisfies criterion *C1* also satisfies *C2*



ISP Summary

Fairly easy to apply, even with no automation

Convenient ways to increase or decrease test cases

Applicable to all levels of testing

Based on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used!