Introduction to Software Testing

Chapter 4
Input Space Partition Testing

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/
Ch. 4 : Input Space Coverage

Four Structures for Modeling Software

Graphs
Logic
Input Space
Syntax

Applied to

Source
Specs
DNF
FSMs

Source
Specs
Design
Use cases

Source
Models
Integ
Input
Input Domains

• The input domain to a program contains all the possible inputs to that program
• For even small programs, the input domain is so large that it might as well be infinite
• Testing is fundamentally about choosing finite sets of values from the input domain
• Input parameters define the scope of the input domain
  – Parameters to a method
  – Data read from a file
  – Global variables
  – User level inputs
• Domain for each input parameter is partitioned into regions
• At least one value is chosen from each region
Benefits of ISP

• Can be **equally applied** at several levels of testing
  – Unit
  – Integration
  – System
• Relatively easy to apply with **no automation**
• Easy to **adjust** the procedure to get more or fewer tests
• **No implementation knowledge** is needed
  – just the input space
Partitioning Domains

- Domain $D$
- Partition scheme (*characteristic*) $q$ of $D$
- The partition $q$ defines a set of blocks, $Bq = b_1, b_2, \ldots, b_Q$
- The partition must satisfy two properties:
  1. blocks must be *pairwise disjoint* (no overlap)
  2. together the blocks *cover* the domain $D$ (complete)

\[ b_i \cap b_j = \emptyset, \forall i \neq j, b_i, b_j \in B_q \]

\[ \bigcup b = D \]

\[ b \in B_q \]
Using Partitions – Assumptions

• Choose a **value** from each partition
• Each value is assumed to be **equally useful** for testing
• Application to testing
  – Find **characteristics** in the inputs: parameters, semantic descriptions, ...
  – **Partition** each characteristic
  – **Choose tests** by combining values from characteristics

• Example **Characteristics**
  – Input X is null
  – Order of the input file F (*sorted, inverse sorted, arbitrary, ...*)
  – Min separation of two aircraft
  – Input device (DVD, CD, VCR, computer, ...)
Choosing Partitions

- Choosing (or defining) partitions seems easy, **but** it is easy to get wrong.
- Consider the "order of file F".

The file will be in all three blocks, thus **disjointness is not satisfied**.

Solution:

Each characteristic should address just one property.

1. File F sorted ascending: \( b_1 = \text{true} \), \( b_2 = \text{false} \)
2. File F sorted descending: \( b_1 = \text{true} \), \( b_2 = \text{false} \)
Properties of Partitions

• If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough

• They should be reviewed carefully, like any design attempt

• Different alternatives should be considered

• We model the input domain in five steps ...
Modeling the Input Domain

- **Step 1: Identify testable functions**
  - Individual *methods* have one testable function
  - In a *class*, each method often has the same characteristics
  - *Programs* have more complicated characteristics—modeling documents such as UML *use cases* can be used to design characteristics
  - *Systems* of integrated hardware and software components can use *devices*, *operating systems*, *hardware platforms*, *browsers*, etc.

- **Step 2: Find all the parameters**
  - Often fairly *straightforward*, even mechanical
  - Important to be *complete*
  - *Methods*: Parameters and state (non-local) variables used
  - *Components*: Parameters to methods and state variables
  - *System*: All inputs, including files and databases
Modeling the Input Domain (cont.)

- **Step 3** : Model the input domain (IDM)
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in applying ISP

- **Step 4** : Apply a test criterion to choose combinations of values
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen

- **Step 5** : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block
Two Approaches to Input Domain Modeling (IDM)

1. **Interface-based** approach
   - Develops characteristics directly from individual input parameters
   - **Simplest** application
   - Can be **partially automated** in some situations

2. **Functionality-based** approach
   - Develops characteristics from a **behavioral view** of the program under test
   - **Harder** to develop—requires more design effort
   - May result in **better tests**, or fewer tests that are as effective
1. Interface-Based Approach

• Mechanically consider each parameter in isolation
• This is an easy modeling technique and relies mostly on syntax
• Some domain and semantic information won’t be used
  – Could lead to an incomplete IDM
• Ignores relationships among parameters

Consider TriTyp from Chapter 3
Three int parameters
IDM for each parameter is identical
Reasonable characteristic: Relation of side with zero
2. Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it’s harder to translate values to test cases

Consider TriTyp again

- The three parameters represent a triangle
- IDM can combine all parameters
- Reasonable characteristic: Type of triangle
**Steps 1 & 2 – Identifying Functionalities, Parameters and Characteristics**

- A creative engineering step
- More characteristics means more tests
- Interface-based: Translate parameters to characteristics
- Candidates for characteristics:
  - Pre- and Post-conditions
  - Relationships among variables
  - Relationship of variables with special values (zero, null, blank, ...)
- Should not use program source
  - Characteristics should be based on the input domain
  - Program source should be used with graph or logic criteria
- Better to have more characteristics with few blocks
  - Fewer mistakes and fewer tests
### Interface-Based Approach

Two parameters: `list`, `element`

Characteristics:
1. `list` is null (b1 = true, b2 = false)
2. `list` is empty (b1 = true, b2 = false)

### Functionality-Based Approach

Two parameters: `list`, `element`

Characteristics:
1. Number of occurrences of `element` in list
   - (b1 = 0, b2 = 1, b3 = >1)
2. `element` occurs first in list
   - (b1 = true, b2 = false)
3. `element` occurs last in list
   - (b1 = true, b2 = false)
Step 3: Modeling the Input Domain

- Partitioning characteristics into blocks and values is a very creative engineering step.
- More blocks means more tests.
- The partitioning often flows directly from the definition of characteristics and both steps are sometimes done together.
  - Should evaluate them separately – sometimes fewer characteristics can be used with more blocks and vice versa.
- Strategies for identifying values:
  - Include valid, invalid, and special values.
  - Sub-partition some blocks.
  - Explore boundaries of domains.
  - Include values that represent “normal use”.
  - Try to balance the number of blocks in each characteristic.
  - Check for completeness and disjointness.
Interface-Based IDM – TriTyp

- **TriTyp**, from Chapter 3, had one testable function and three integer inputs

<table>
<thead>
<tr>
<th>First Characterization of TriTyp’s Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristic</td>
</tr>
<tr>
<td>q₁ = “Relation of Side 1 to 0”</td>
</tr>
<tr>
<td>q₂ = “Relation of Side 2 to 0”</td>
</tr>
<tr>
<td>q₃ = “Relation of Side 3 to 0”</td>
</tr>
</tbody>
</table>

- A maximum of 3*3*3 = 27 possible tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests ...
Interface-Based IDM – TriTyp (cont.)

Second Characterization of TriTyp’s Inputs

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 = \text{“Refinement of } q_1 \text{”} )</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
<tr>
<td>( q_2 = \text{“Refinement of } q_2 \text{”} )</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
<tr>
<td>( q_3 = \text{“Refinement of } q_3 \text{”} )</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
</tbody>
</table>

- A maximum of \( 4 \times 4 \times 4 = 64 \) possible tests
- This is only complete because the inputs are integers (0 . . 1)

Possible values for partition \( q_1 \)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side 1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Test boundary conditions
Functionality-Based IDM – TriTyp

- First two characterizations are based on syntax: parameters and their type
- A semantic level characterization could use the fact that the three integers (parameters) represent a triangle

### Geometric Characterization of TriTyp’s Inputs

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b₁</th>
<th>b₂</th>
<th>b₃</th>
<th>b₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 =$ “Geometric Classification”</td>
<td>scalene</td>
<td>isosceles</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>

- **Oops … something’s fishy …** equilateral is also isosceles!
- We need to refine the example to make characteristics valid

### Correct Geometric Characterization of TriTyp’s Inputs

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b₁</th>
<th>b₂</th>
<th>b₃</th>
<th>b₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 =$ “Geometric Classification”</td>
<td>scalene</td>
<td>isosceles, not equilaterial</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>
### Possible values for geometric partition $q_1$

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td>(4, 5, 6)</td>
<td>(3, 3, 4)</td>
<td>(3, 3, 3)</td>
<td>(3, 4, 8)</td>
</tr>
</tbody>
</table>

- **Values** for this partitioning can be chosen as
A different approach would be to break the geometric characterization into four separate characteristics.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 = \text{&quot;Scalene&quot;}$</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>$q_2 = \text{&quot;Isosceles&quot;}$</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>$q_3 = \text{&quot;Equilateral&quot;}$</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>$q_4 = \text{&quot;Valid&quot;}$</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

Use constraints to ensure that:
- Equilateral = True implies Isosceles = True
- Valid = False implies Scalene = Isosceles = Equilateral = False
Using More than One IDM

• Some programs may have dozens or even hundreds of parameters

• Create several small IDM
  – A divide-and-conquer approach

• Different parts of the software can be tested with different amounts of rigor (consistency, accuracy, care, etc.)
  – For example, some IDM may include a lot of invalid values

• It is okay if the different IDM overlap
  – The same variable may appear in more than one IDM
The Test Selection Problem

• The input domain of a program consists of all possible inputs that could be taken by the program.
• Ideally, the test selection problem is to select a subset $T$ of the input domain such that the execution of $T$ will reveal all errors.
• In practice, the test selection problem is to select a subset of $T$ within budget such that it reveals as many errors as possible.
Step 4 – Choosing Combinations of Values

- Once characteristics and partitions are defined, the next step is to choose test values.
- We use criteria – to choose effective subsets.
- The most obvious criterion is to choose all combinations.

**All Combinations Coverage (ACoC)**: All combinations of blocks from all characteristics must be used.

- Number of tests is the product of the number of blocks in each characteristic: \( \prod_{i=1}^{Q} (B_i) \)
- The second characterization of TriTyp results in 4*4*4 = 64 tests – too many?
**Example**

ACoC for the **second** categorization of TriTyp’s inputs in Table 4.2.

<table>
<thead>
<tr>
<th></th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$&gt; 1$</td>
<td>1</td>
<td>0</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$&gt; 1$</td>
<td>1</td>
<td>0</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$&gt; 1$</td>
<td>1</td>
<td>0</td>
<td>$&lt; 0$</td>
</tr>
</tbody>
</table>

(2, 2, 2), (2, 1, 2), (2, 0, 2), (2,−1, 2), (2, 2, 1), (2, 1, 1), (2, 0, 1), (2,−1, 1), (2, 2, 0), (2, 1, 0), (2, 0, 0), (2,−1, 0), (2, 2,−1), (2, 1,−1), (2, 0,−1), (2,−1,−1), (1, 2, 2), (1, 1, 2), (1, 0, 2), (1,−1, 2), (1, 2, 1), (1, 1, 1), (1, 0, 1), (1,−1, 1), (1, 2, 0), (1, 1, 0), (1, 0, 0), (1,−1, 0), (1, 2,−1), (1, 1,−1), (1, 0,−1), (1,−1,−1), (0, 2, 2), (0, 1, 2), (0, 0, 2), (0,−1, 2), (0, 2, 1), (0, 1, 1), (0, 0, 1), (0,−1, 1), (0, 2, 0), (0, 1, 0), (0, 0, 0), (0,−1, 0), (0, 2,−1), (0, 1,−1), (0, 0,−1), (0,−1,−1), (−1, 2, 2), (−1, 1, 2), (−1, 0, 2), (−1,−1, 2), (−1, 2, 1), (−1, 1, 1), (−1, 0, 1), (−1,−1, 1), (−1, 2, 0), (−1, 1, 0), (−1, 0, 0), (−1,−1, 0), (−1, 2,−1), (−1, 1,−1), (−1, 0,−1), (−1,−1,−1)

**# of Test is 64**
ISP Criteria – Each Choice

- 64 tests for TriTyp is almost certainly way too many.
- One criterion comes from the idea that we should try at least one value from each block.

**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 \( \text{Max}^Q_{i=1}(B_i) \)

<table>
<thead>
<tr>
<th></th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
</tbody>
</table>

For TriTyp: 2, 2, 2

- 1, 1, 1
- 0, 0, 0
- -1, -1, -1
ISP Criteria – Pair-Wise

- Each Choice (ECC) yields few tests – cheap but perhaps ineffective
- Another approach asks values to be combined with other values

**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

\[
(\text{Max}_{i=1} Q (B_i)) \times (\text{Max}_{j=1, j \neq i} Q (B_j))
\]

<table>
<thead>
<tr>
<th></th>
<th>(b_1)</th>
<th>(b_2)</th>
<th>(b_3)</th>
<th>(b_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1)</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>(q_2)</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>(q_3)</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
</tbody>
</table>

**For TriTyp:**

- \(2, 2, 2\)
- \(2, 1, 1\)
- \(2, 0, 0\)
- \(2, -1, -1\)
- \(1, 2, 1\)
- \(1, 1, 0\)
- \(1, 0, -1\)
- \(1, -1, 2\)
- \(0, 2, 0\)
- \(0, 1, -1\)
- \(0, 0, 2\)
- \(0, -1, 1\)
- \(-1, 2, -1\)
- \(-1, 1, 2\)
- \(-1, 0, 1\)
- \(-1, -1, 0\)
### Example

- Assume that we have three partitions, each is divided into blocks:

<table>
<thead>
<tr>
<th>( b_1 )</th>
<th>( b_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_1 )</td>
<td>A</td>
</tr>
<tr>
<td>( B_2 )</td>
<td>1</td>
</tr>
<tr>
<td>( B_3 )</td>
<td>x</td>
</tr>
</tbody>
</table>

#### ACoC

(A, 1, x) (B, 1, x)
(A, 1, y) (B, 1, y)
(A, 2, x) (B, 2, x)
(A, 2, y) (B, 2, y)
(A, 3, x) (B, 3, x)
(A, 3, y) (B, 3, y)

- # of Tests is
  \[ 2 \times 3 \times 2 = 12 \]

#### ECC

- ECC can be satisfied in many ways. *i.e.*
  (A, 1, x)
  (B, 2, y)
  (A, 3, x)

- # of Tests is
  \[ 3 \]

#### PWC

- All Pairs
  (A, 1) (B, 1) (1, x)
  (A, 2) (B, 2) (1, y)
  (A, 3) (B, 3) (2, x)
  (A, x) (B, x) (2, y)
  (A, y) (B, y) (3, x)
  (A, y) (B, y) (3, y)

- # of all pairs
  \[ (2 \times 3) + (2 \times 2) + (3 \times 2) = 16 \]

- # of Tests is
  \[ 12 \]

**But:**

PWC allows the same test case to cover more than one unique pair of values

(A, 1, x) (B, 1, y)
(A, 2, x) (B, 2, y)
(A, 3, x) (B, 3, y)
(A, _, y) (B, _, x)
ISP Criteria – T-Wise

• A natural extension is to require combinations of $T$ values instead of 2

**T-Wise Coverage (TWC):** A value from each block for each group of $T$ characteristics must be combined.

• Number of tests is at least the product of $T$ largest characteristics
• If all characteristics are the same size, the formula is

$$ (\text{Max}_{i=1}^{Q}(B_i))^t $$

• If $T$ is the number of characteristics $Q$, then all combinations
• That is … $Q$-wise = ACoC
• $T$-Wise is expensive and benefits are not clear
ISP Criteria – Base Choice

• Both PWC and TWC combine values “blindly” without regard for which values are being combined.

• Testers sometimes recognize that certain values are more important than others.

• This important value is called **Base Choice**
  – It can be obtained by asking what is the most “important” block for each partition.

• Simply, **Base Choice** strengthens **ECC** by utilizing information about the domain (domain knowledge).

• The **Base Choice** can be
  – the *simplest*,
  – the *smallest*,
  – the *first in some ordering*,
  – or the *most likely* from an end-user point of view, ... etc.
ISP Criteria – Base Choice /2

**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

\[ 1 + \sum_{i=1}^{Q} (B_i - 1) \]

<table>
<thead>
<tr>
<th>( q_1 )</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
<td>&lt; 0</td>
</tr>
</tbody>
</table>

**For TriTyp: Base is 2, 2, 2**

(>1 is our BC)

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

Our Base Choice Block
ISP Criteria – Multiple Base Choice

• Testers sometimes have **more than one** logical base choice

**Multiple Base Choice Coverage (MBCC)**: One or more base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choices in each other characteristic.

• If there are $M$ base tests and $m_i$ Base Choices for each characteristic:

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

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$q_3$</td>
<td>&gt; 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**For TriTyp: Base**
(two base choices: $>1$ and $1$)

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

Introduction to Software Testing (Ch 4)
ISP Coverage Criteria Subsumption

- All Combinations Coverage (ACoC)
- T-Wise Coverage (TWC)
- Pair-Wise Coverage (PWC)
- Each Choice Coverage (ECC)
- Multiple Base Choice Coverage (MBCC)
- Base Choice Coverage (BCC)
Constraints Among Characteristics

• Some combinations of blocks are infeasible
  – “less than zero” and “scalene” ... not possible at the same time
• These are represented as constraints among blocks
• Two general types of constraints
  – A block from one characteristic cannot be combined with a specific block from another
  – A block from one characteristic can ONLY BE combined with a specific block from another characteristic
• Handling constraints depends on the criterion used
  – ACoC, PWC, TWC : Drop the infeasible pairs
  – BCC, MBCC : Change a value to another non-base choice to find a feasible combination
### Example Handling Constraints

- **Sorting an array**
  - **Input**: variable length array of arbitrary type
  - **Outputs**: sorted array, largest value, smallest value

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Partitions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of array</td>
<td>Len: { 0, 1, 2..100, 101..MAXINT }</td>
</tr>
<tr>
<td>Type of elements</td>
<td>Type: { int, char, string, other }</td>
</tr>
<tr>
<td>Max value</td>
<td>Max: { ≤0, 1, &gt;1, ‘a’, ‘Z’, ‘b’, …, ‘Y’ }</td>
</tr>
<tr>
<td>Min value</td>
<td>Min: { … }</td>
</tr>
<tr>
<td>Position of max value</td>
<td>Max Pos: { 1, 2 .. Len-1, Len }</td>
</tr>
<tr>
<td>Position of min value</td>
<td>Min Pos: { 1, 2 .. Len-1, Len }</td>
</tr>
</tbody>
</table>

Blocks from other characteristics are irrelevant

Blocks must be combined

Blocks must be combined
Input Space Partitioning Summary

• Fairly easy to apply, even with no automation

• Convenient ways to add more or less testing

• Applicable to all levels of testing – unit, class, integration, system, etc.

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

Simple, straightforward, effective, and widely used in practice