Input Space Partitioning Testing

CS 4501 / 6501 Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 6.1]
Structures for Criteria-Based Testing

Four structures for modeling software

Input space

Applied to

Graph
- Source
- Design
- Specs
- Use cases

Logic
- Source
- Specs
- FSMs
- DNF

Syntax
- Source
- Models
- Integration
- Inputs

---R
R--R
RI-R
RIPR
Today’s Objectives

• Input domain (or input space)

• Fundamental of Input Space Partitioning (ISP)
  • Benefits of ISP
  • Partitioning input domain
  • Modeling input domain
Software Testing

- **Testing** = process of finding test input values to check against a software

Test case consists of test values and expected results

1. Testing is fundamentally about choosing finite sets of values from the **input domain** of the software being tested

2. Given the test inputs, compare the actual results with the expected results
Input Domains

- **All possible values** that the input parameters can have
- The input domain may be infinite even for a small program
- Testing is fundamentally about **choosing finite sets of values** from the input domain

- **Input parameters** can be
  - Parameters to a method (in unit testing)
  - Global variables (in unit testing)
  - Objects representing current state (in class or integration testing)
  - User level inputs (in system testing)
  - Data read from a file
# Example Input Domains

```python
def get_index_of(string, letter):
    index = -1
    for i in range(1, len(string)):
        if string[i] == letter:
            return i
    return index
```

What is the domain of `string`?
What is the domain of `letter`?
Overview: ISP

• Input space partitioning describes the input domain of the software

• Domain \((D)\) are partitioned into blocks \((b_1, b_2, \ldots, b_n)\)

• The partition (or block) must satisfy two properties
  • Blocks must not overlap (disjointness)
  • Blocks must cover the entire domain (completeness)

• At least one value is chosen from each block
  • Each value is assumed to be equally useful for testing
Benefits of ISP

• Easy to get started
  • Can be applied with no automation and very little training

• Easy to adjust to procedure to get more or fewer tests

• No implementation knowledge is needed
  • Just a description of the inputs

• Can be equally applied at several levels of testing
  • Unit (inputs from method parameters and non-local variables)
  • Integration (inputs from objects representing current state)
  • System (user-level inputs to a program)
Applying ISP

Task I: Model input domain
(choose characteristics and partition)

The most creative design step in using ISP

Task II: Choose combinations of values
(apply coverage criterion)

Identify testable functions

Identify parameters, return types, return values, exceptional behavior

Model the input domain

Input Domain Model (IDMs)

Apply a test criterion to choose combinations of blocks

Test requirements (TRs)

Derive test values

Test cases

Identify parameters, return types, return values, exceptional behavior

Model the input domain

Apply a test criterion to choose combinations of blocks

Test requirements (TRs)

Derive test values

Test cases
Modeling the Input Domain

- The domain is scoped by the **parameters**

- **Characteristics** define the structure of the input domain
  - Characteristics should be based on the input domain – not program source

- Two Approaches
  - **Interface-based** (simpler)
    - Develop characteristics from individual parameters
  - **Functionality-based** (harder)
    - Develop characteristics from a behavior view

- Design characteristics
  - Partition each characteristic into blocks
  - Identify values of each block
Design Characteristics

Interface-based

- Develop characteristics directly from parameters
  - Translate parameters to characteristics

- Consider each parameter separately

- Rely mostly on syntax

- Ignore some domain and semantic information
  - Can lead to an incomplete IDM

- Ignore relationships among parameters

Functionality-based

- Develop characteristics that correspond to the intended functionality

- Can use relationships among parameters, relationships of parameters with special values (null, blank, ...), preconditions, and postconditions

- Incorporate domain and semantic knowledge
  - May lead to better tests

- The same parameter may appear in multiple characteristics
Partition Characteristics

Strategies for both approaches

- Partition is a set of blocks, designed using knowledge of what the software is supposed to do
- Each block represents a set of values
- More blocks means more tests
- Partition must satisfy disjointness and completeness properties

- Better to have more characteristics with few blocks
  - Fewer mistakes and fewer tests

How partitions should be identified and how representative value should be selected from each block
Identify Values

Strategies for both approaches

- 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

- Each value is assumed to be equally useful for testing
# Return index of the first occurrence of a letter in string, # Otherwise, return -1

def get_index_of(string, letter):

**Task I: Model Input Domain**

1. **Identify testable functions**
   - `get_index_of()`

2. **Identify parameters, return types, return values, and exceptional behavior**
   - Parameters: `string`, `letter`
   - Return type: `int`
   - Return value: index of the first occurrence, -1 if no occurrence
   - Exceptional behavior: ??
Interface-based Example1 (cont)

3. Model the input domain
   - Develop characteristics
     - $C_1 = \text{string}$ is empty
     - $C_2 = \text{letter}$ is empty
   - Partition characteristics
     - What are other possible characteristics?

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 = \text{string}$ is empty</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>$C_2 = \text{letter}$ is empty</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

- Identify (possible) values

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 = \text{string}$ is empty</td>
<td>&quot;&quot;</td>
<td>&quot;testing&quot;</td>
</tr>
<tr>
<td>$C_2 = \text{letter}$ is empty</td>
<td>&quot;&quot;</td>
<td>&quot;t&quot;</td>
</tr>
</tbody>
</table>
Interface-based Example1 (cont)

Task II: Choose combinations of values

4. Combine partitions to define test requirements
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 2 * 2 = 4
     (True, True)  (False, True)
     (True, False) (False, False)
   - Eliminate redundant tests and infeasible tests

5. Derive test values

<table>
<thead>
<tr>
<th>Test</th>
<th>string</th>
<th>letter</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (True, True)</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>-1</td>
</tr>
<tr>
<td>T2 (True, False)</td>
<td>&quot;&quot;</td>
<td>&quot;t&quot;</td>
<td>-1</td>
</tr>
<tr>
<td>T3 (False, True)</td>
<td>&quot;testing&quot;</td>
<td>&quot;&quot;</td>
<td>-1</td>
</tr>
<tr>
<td>T4 (False, False)</td>
<td>&quot;testing&quot;</td>
<td>&quot;t&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
Functionality-based Example 1

# Return index of the first occurrence of a letter in string, 
# Otherwise, return -1

def get_index_of(string, letter):

Task I: Model Input Domain

1. Identify testable functions
   • get_index_of()

2. Identify parameters, return types, return values, and exceptional behavior
   • Parameters: string, letter
   • Return type: int
   • Return value: index of the first occurrence, -1 if no occurrence
   • Exceptional behavior: ??
Functionality-based Example 1 (cont)

3. Model the input domain

- Develop characteristics
  - $C_1 =$ number of occurrence of letter in string
  - $C_2 =$ letter occurs first in string

- Partition characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 =$ number of occurrence of letter in string</td>
<td>0</td>
<td>1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>$C_2 =$ letter occurs first in string</td>
<td>True</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

- Identify (possible) values

<table>
<thead>
<tr>
<th>C</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>&quot;software engineering&quot;, &quot;&quot;</td>
<td>&quot;software engineering&quot;, &quot;s&quot;</td>
<td>&quot;software engineering&quot;, &quot;n&quot;</td>
</tr>
<tr>
<td>$C_2$</td>
<td>&quot;software engineering&quot;, &quot;s&quot;</td>
<td>&quot;software engineering&quot;, &quot;t&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Functionality-based Example 1 (cont)

Task II: Choose combinations of values

4. Combine partitions into tests
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 3 * 2 = 6
     - T1 (0, True)
     - T2 (1, True)
     - T3 (1, False)
     - T4 (>1, True)
     - T5 (>1, False)
   - Eliminate redundant tests and infeasible tests

5. Derive test values

<table>
<thead>
<tr>
<th>Test</th>
<th>string</th>
<th>letter</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>&quot;software engineering&quot;</td>
<td>&quot;&quot;</td>
<td>-1</td>
</tr>
<tr>
<td>T2 (1, True)</td>
<td>&quot;software engineering&quot;</td>
<td>&quot;s&quot;</td>
<td>0</td>
</tr>
<tr>
<td>T3 (1, False)</td>
<td>&quot;software engineering&quot;</td>
<td>&quot;t&quot;</td>
<td>3</td>
</tr>
<tr>
<td>T4 (&gt;1, True)</td>
<td>&quot;software testing&quot;</td>
<td>&quot;s&quot;</td>
<td>0</td>
</tr>
<tr>
<td>T5 (&gt;1, False)</td>
<td>&quot;software engineering&quot;</td>
<td>&quot;n&quot;</td>
<td>10</td>
</tr>
</tbody>
</table>
Interface-based Example 2

```java
public enum Triangle {Scalene, Isosceles, Equilateral, Invalid}
public static Triangle triang (int Side1, int Side2, int Side3)
# Side1, Side2, and Side3 represent the lengths of the sides of a
# triangle.
# Return the appropriate enum value
```

**Task I: Model Input Domain**

1. **Identify testable functions**
   - `triang()`

2. **Identify parameters, return types, return values, and exceptional behavior**
   - Parameters: `Side1`, `Side2`, `Side3`
   - Return type: `enum`
   - Return value: enum describing type of a triangle
   - Exceptional behavior: ??
3. Model the input domain

• Develop characteristics
  • $C_1 =$ relation of $\text{Side}_1$ to 0
  • $C_2 =$ relation of $\text{Side}_2$ to 0
  • $C_3 =$ relation of $\text{Side}_3$ to 0

• Partition characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 =$ relation of $\text{Side}_1$ to 0</td>
<td>greater than 0</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
<tr>
<td>$C_2 =$ relation of $\text{Side}_2$ to 0</td>
<td>greater than 0</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
<tr>
<td>$C_3 =$ relation of $\text{Side}_3$ to 0</td>
<td>greater than 0</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
</tbody>
</table>

• Identify (possible) values

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 =$ relation of $\text{Side}_1$ to 0</td>
<td>7</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>$C_2 =$ relation of $\text{Side}_2$ to 0</td>
<td>3</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$C_3 =$ relation of $\text{Side}_3$ to 0</td>
<td>2</td>
<td>0</td>
<td>-2</td>
</tr>
</tbody>
</table>
### Interface-based Example2 (cont)

- **Refine characteristics** (can lead to more tests)
  - $C_1 = \text{length of Side}_1$
  - $C_2 = \text{length of Side}_2$
  - $C_3 = \text{length of Side}_3$

- **Partition characteristics**

<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>$C_1 = \text{length of Side}_1$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
<tr>
<td>$C_2 = \text{length of Side}_2$</td>
<td>greater than 0</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
<tr>
<td>$C_3 = \text{length of Side}_3$</td>
<td>greater than 0</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>less than 0</td>
</tr>
</tbody>
</table>

- **Identify (possible) values**

<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>$C_1 = \text{length of Side}_1$</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$C_2 = \text{length of Side}_2$</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$C_3 = \text{length of Side}_3$</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Complete? Disjoint?**

- **Valid triangles?**

- **Boundary tests**

  Refining characterization to get more fine-grained testing (if the budget allows)
Task II: Choose combinations of values

4. Combine partitions to define test requirements
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = \(4^3 = 64\)
     - (C1b1, C2b1, C3b1)
     - (C1b1, C2b2, C3b1)
     - (C1b1, C2b3, C3b1)
     - (C1b1, C2b4, C3b1)
     - (C1b1, C2b1, C3b2)
     - (C1b1, C2b2, C3b2)
     - (C1b1, C2b3, C3b2)
     - (C1b1, C2b4, C3b2)
     - (C1b1, C2b1, C3b3)
     - (C1b1, C2b2, C3b3)
     - (C1b1, C2b3, C3b3)
     - (C1b1, C2b4, C3b3)
     - (C1b1, C2b1, C3b4)
     - (C1b1, C2b2, C3b4)
     - (C1b1, C2b3, C3b4)
     - (C1b1, C2b4, C3b4)
     - (C1b2, C2b1, C3b4)
     - (C1b2, C2b2, C3b4)
     - (C1b2, C2b3, C3b4)
     - (C1b2, C2b4, C3b4)
     - ...
     - Do we really need these many tests?
   - Eliminate redundant tests and infeasible tests

5. Derive test values
   - (2, 2, 2)
   - (2, 1, 2)
   - (2, 0, 2)
   - (2, -1, 2)
   - (2, 2, 1)
   - (2, 1, 1)
   - (2, 0, 1)
   - (2, -1, 1)
   - ...

Do we really need these many tests?
public enum Triangle {Scalene, Isosceles, Equilateral, Invalid}
public static Triangle triang (int Side1, int Side2, int Side3)
# Side1, Side2, and Side3 represent the lengths of the sides of a
# triangle.
# Return the appropriate enum value

**Task I: Model Input Domain**

1. Identify testable functions
   - `triang()`

2. Identify parameters, return types, return values, and exceptional behavior
   - Parameters: `Side1, Side2, Side3`
   - Return type: `enum`
   - Return value: enum describing type of a triangle
   - Exceptional behavior: ??
3. Model the input domain

- Develop characteristics
  - \( C_1 \) = Geometric classification

- Partition characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 ) = Geometric classification</td>
<td>scalene</td>
<td>isosceles</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>

What are other possible characteristics?

- Refine characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 ) = Geometric classification</td>
<td>scalene</td>
<td>isosceles, not equilateral</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>

- Identify (possible) values

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 ) = Geometric classification</td>
<td>(4, 5, 6)</td>
<td>(3, 3, 4)</td>
<td>(3, 3, 3)</td>
<td>(3, 4, 8)</td>
</tr>
</tbody>
</table>
**Functionality-based Example 2 (cont)**

**Task II: Choose combinations of values**

4. Combine partitions into tests
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 4
     - (C1b1)  (C1b2)  (C1b3)  (C1b4)
   - Eliminate redundant tests and infeasible tests

5. Derive test values

<table>
<thead>
<tr>
<th>Test</th>
<th>Side1</th>
<th>Side2</th>
<th>Side3</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (scalene)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>scalene</td>
</tr>
<tr>
<td>T2 (isosceles, not equilateral)</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>isosceles</td>
</tr>
<tr>
<td>T3 (equilateral)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>equilateral</td>
</tr>
<tr>
<td>T4 (invalid)</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>invalid</td>
</tr>
</tbody>
</table>

This characteristic results in a simple set of test requirements. Is this good enough? If we define the characteristics differently? Multiple IDMs?
ISP Task I Summary

- Easy to apply, even with no automation and little training
- Easy to add more or fewer tests
- Rely on the input space, not implementation knowledge
- Applicable to all levels of testing, effective and widely used

**Interface-based approach**

**Strength**
- Easy to identify characteristics
- Easy to translate abstract tests into executable test cases

**Weakness**
- Some information will not be used – lead to incomplete IDM
- Ignore relationships among parameters

**Functionality-based approach**

**Strength**
- Incorporate semantic
- Input domain modeling and test case generation in early development phases

**Weakness**
- Difficult to design reasonable characteristics
- Hard to generate tests
What’s Next?

- How should we consider multiple partitions or IDMs at the same time?
- What combinations of blocks should we choose values from?
- How many tests should we expect?