Input Space Partition Testing

Software Testing

(3104313)

Amirkabir University of Technology
Spring 1399-1400

Input Domain

The input domain is defined in terms of the **possible values** that the input parameters can have.

- Method parameters
- Global variables
- Objects representing current state
- User-level inputs
- …

- The input domain is **partitioned** into **regions** (blocks)
- At least **one value** is selected from each region

Partitioning Domains

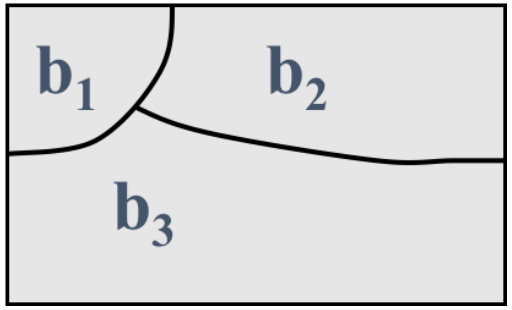
- *Domain **D***
- *Partition scheme **q** of **D***
- The partition ***q*** defines a set of blocks, $\boldsymbol{Bq = b_1, b_2, \ldots, b_Q}$
- The partition must satisfy **two** properties

1. Blocks must be **pairwise disjoint** (no overlap)

$$\mathbf{b_i \cap b_j = \Phi,\ \forall\ i \neq j,\ b_i, b_j \in B_q}$$

2. Together the blocks cover the domain D (**complete**)

$$\bigcup_{b \in Bq} \mathbf{b = D}$$



Example

Partitioning for integers

Design a partitioning for all integers

Choosing Right Partitions

Consider the "***order of elements in list F***"

Design blocks for that characteristic

- ?

Input Domain Model (IDM)

- An input domain model (IDM) represents the **input space** of the system under test in an **abstract way**.

- A test engineer describes the **structure** of the input domain in terms of **input characteristics**.

Modelling the Input Domain

- Consider Java ***Iterator*** Interface.
- How would you design an IDM for **Iterator**?

We will coma back to this example later!

Modelling the Input Domain

Step 1- Identify testable functions

Step 2- Find all inputs, parameters, & characteristics

Move from impl. level to design abstraction level

Step 3- Model the input domain

Step 4- Apply a test criterion to choose combinations of values

Entirely at the design abstraction level

Step 5- Refine combinations of blocks into test inputs

Back to the implementation abstraction level

Modelling the Input Domain

Step 1- Identify testable functions

Step 2- Find all inputs, parameters, & characteristics

Move from impl. level to design abstraction level

Step 3- Model the input domain

Step 4- Apply a test criterion to choose combinations of values

Entirely at the design abstraction level

Step 5- Refine combinations of blocks into test inputs

Back to the implementation abstraction level

Modelling the Input Domain

Step 1- Identify testable functions

Step 2- Find all inputs, parameters, & characteristics

Move from impl. level to design abstraction level

Step 3- Model the input domain

Step 4- Apply a test criterion to choose combinations of values

Entirely at the design abstraction level

Step 5- Refine combinations of blocks into test inputs

Back to the implementation abstraction level

Modelling the Input Domain

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

Suggest an IDM?

Modelling the Input Domain

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

Modelling the Input Domain

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

(Syntax)

Characteristic: *Relation of side with zero*

Blocks: *negative; positive; zero*

Behaviour

Characteristic: *Type of triangle*

Blocks*: Scalene; Isosceles; Equilateral; Invalid*

In-Class Exercise #10

- Identify testable functions, parameters, and characteristics for ***findElement ()***
  - Two approaches

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

- You have 5 minutes
- Do the exercise individually/in groups (of 3)
- Upload your answer in Moodle.

Characteristics, Blocks, & Values

- Valid, invalid, special values
- Explore boundaries
- Balance the number of blocks in the characteristics
- Missing blocks
- …

Candidates for characteristics

- Interface-based approach
- Preconditions and postconditions
- Relationships among variables
- Relationship of variables with special values (zero, null, blank, …)
- …

triang()

Characteristic: Relation of Side with Zero

Characteristic	b_1	b_2	b_3
q_1 = “Relation of Side 1 to 0”	positive	equal to 0	negative
q_2 = “Relation of Side 2 to 0”	positive	equal to 0	negative
q_3 = “Relation of Side 3 to 0”	positive	equal to 0	negative

triang()

Characteristic: Type of Triangle

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles	equilateral	invalid

triang()

Characteristic: Type of Triangle

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles	equilateral	invalid

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles, not equilateral	equilateral	invalid

triang()

Four separate characteristics!

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

- Use **constraints** to ensure that
  - *Equilateral = True* implies *Isosceles = True*
  - *Valid = False* implies *Scalene = Isosceles = Equilateral = False*

Characteristics, Blocks, & Values

- Valid, invalid, special values
- Explore boundaries
- Balance the number of blocks in the characteristics
- Missing blocks
- …

Candidates for characteristics

- Interface-based approach
- Preconditions and postconditions
- Relationships among variables
- Relationship of variables with special values (zero, null, blank, …)
- …

- M**ore characteristics** with **fewer blocks**
  - Fewer mistakes
  - Fewer tests
- Choose values **strategically**

How should we consider **multiple partitions** at the same time?

What **combination of blocks** should we choose values from?

Combination Strategies Criteria

All Combinations Criterion (ACoC)

Test with all combinations of blocks from all characteristics.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

All Combinations Criterion (ACoC)

Test with all combinations of blocks from all characteristics.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

a1 b1 c1 d1	a1 b2 c1 d1	a2 b1 c1 d1	a2 b2 c1 d1
a1 b1 c1 d2	a1 b2 c1 d2	a2 b1 c1 d2	a2 b2 c1 d2
a1 b1 c2 d1	a1 b2 c2 d1	a2 b1 c2 d1	a2 b2 c2 d1
a1 b1 c2 d2	a1 b2 c2 d2	a2 b1 c2 d2	a2 b2 c2 d2
a1 b1 c3 d1	a1 b2 c3 d1	a2 b1 c3 d1	a2 b2 c3 d1
a1 b1 c3 d2	a1 b2 c3 d2	a2 b1 c3 d2	a2 b2 c3 d2

All Combinations Criterion (ACoC)

Test with all combinations of blocks from all characteristics.

- Number of tests is the product of the number of blocks in each characteristic

$$\prod_{i=1}^{Q} (B_i)$$

Too Many Tests?!!!

Many Invalid Test?!!!

Each Choice Coverage (ECC)

Use at least one value from each block for each characteristic in at least one test case.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

Each Choice Coverage (ECC)

Use at least one value from each block for each characteristic in at least one test case.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

a1 b1 c1 d1
a2 b2 c2 d2
a1 b1 c3 d1

Each Choice Coverage (ECC)

Use at least one value from each block for each characteristic in at least one test case.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

a1 b1 c1 d1
a2 b2 c2 d2
a1 b1 c3 d1

- Number of tests is the number of blocks in the largest characteristic

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

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.

- 4 Characteristics: A, B, C, D
- Abstract blocks

  A = [a1, a2];

  B = [b1, b2];

  C = [c1, c2, c3];

  D = [d1, d2]

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.

- 4 Characteristics: A, B, C, D
- Abstract blocks

  A = [a1, a2];

  B = [b1, b2];

  C = [c1, c2, c3];

  D = [d1, d2]

(a1, b1)	(a2, b1)	(b1, c1)	(b2, c1)	(c1, d1)
(a1, b2)	(a2, b2)	(b1, c2)	(b2, c2)	(c1, d2)
(a1, c1)	(a2, c1)	(b1, c3)	(b2, c3)	(c2, d1)
(a1, c2)	(a2, c2)	(b1, d1)	(b2, d1)	(c2, d2)
(a1, c3)	(a2, c3)	(b1, d2)	(b2, d2)	(c3, d2)
(a1, d1)	(a2, d1)			
(a1, d2)	(a2, d2)			

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

$$(\text{Max}_{i=1}^{Q}(B_i)) * (\text{Max}_{j=1,\, j!=i}^{Q}(B_j))$$

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

After-Class Exercise #11

- Give a set of tests for the running example applying the pair-wise coverage criterion.

- You have ∞ minutes ☺
- Do the exercise individually/in groups (of 3)
- Upload your answer in Moodle.

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.

- 4 Characteristics: A, B, C, D
- Abstract blocks

  A = [a1, a2];

  B = [b1, b2];

  C = [c1, c2, c3];

  D = [d1, d2]

Base	**a1 b1 c1 d1**
A	**a2** b1 c1 d1
B	a1 **b2** c1 d1
C	a1 b1 **c2** d1
C	a1 b1 **c3** d1
D	a1 b1 c1 **d2**

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

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

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 for each base test and using each non-base choice in each other characteristic.

- If *M* base tests and m_i base choices for each characteristic:

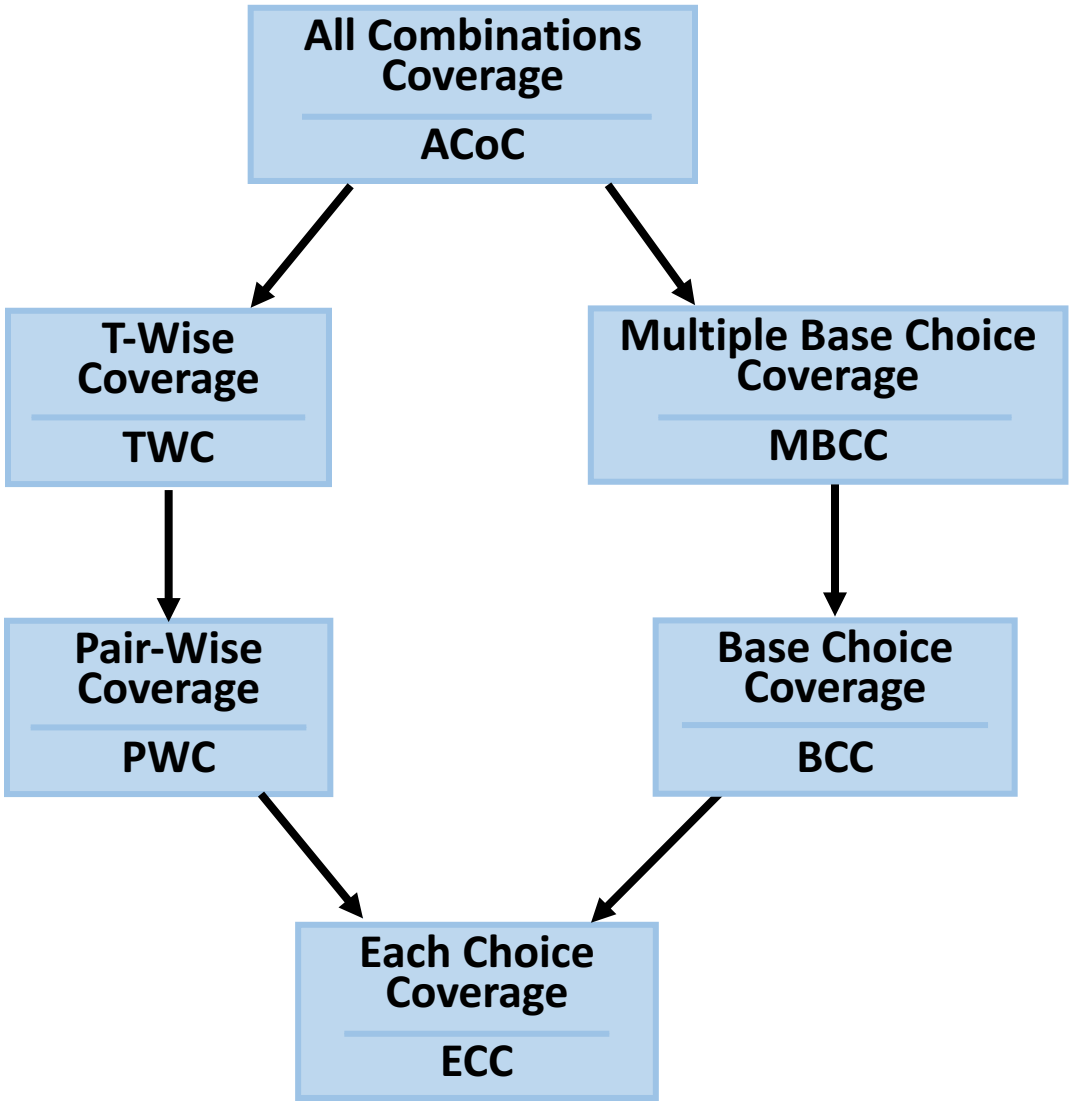
$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

In/After-Class Exercise #12

- Define a test set for the running example, that satisfies the multiple base choice coverage (MBCC) with the following base tests:
  - a1, b1, c1, d1
  - a2, b2, c2, d2

- You have ∞ minutes ☺
- Do the exercise individually/in groups (of 3)
- Upload your answer in Moodle.

Coverage Criteria Subsumption



Another Example

```
public final class Time  {
   private int hours;
   private int minutes;
  /**
   * Constructor.
   * @param h hours (from 0 to 23)
   * @param m minutes (from 0 to 59)
   * @throws IllegalArgumentException if arguments are invalid
   */
  public Time(int h, int m) throws IllegalArgumentException { ... }
  public int getHours() { ... }
  public int getMinutes() { ... }
  /**
   * Advance one minute to the current time.
   */
  public void tick() { ... }
  public boolean equals(Object o) { ... }
  public String toString() { ... }
}
```

ACoC Coverage for `Time.toString()`

Test	Combination	hours	minutes	Result
1	(am, n, n)	10	30	“10:30 AM”
2	(am, n, y)	10	0	“10:00 AM”
3	(am, y, n)	9	30	“09:30 AM”
4	(am, y, y)	9	0	“09:00 AM”
5	(pm, n, n)	22	30	“10:30 PM”
6	(pm, n, y)	22	0	“10:00 PM”
7	(pm, y, n)	21	30	“09:30 PM”
8	(pm, y, y)	21	0	“09:00 PM”

BCC Coverage for `Time.toString()`

Test	Combination	hours	minutes	Result
1	**(am, n, n)**	10	30	“10:30 AM”
2	(**pm**, n, n)	22	0	“10:30 AM”
3	(am, **y**, n)	9	30	“09:30 AM”
4	(am, n, **y**)	10	0	“10:00 AM”

An Industrial Study of Applying Input Space Partitioning to Test Financial Calculation Engines

Jeff Offutt
Software Engineering
George Mason University
Fairfax, VA, USA
offutt@gmu.edu

Chandra Alluri
Freddie Mac
McLean, VA, USA
chandra_alluri@freddiemac.com

Abstract This paper presents results from an industrial study that applied input space partitioning and semi-automated requirements modeling to large-scale industrial software, specifically financial calculation engines. Calculation engines are used in financial service applications such as banking, mortgage, insurance, and trading to compute complex, multi-conditional formulas to make high risk financial decisions. They form the heart of financial applications, and can cause severe economic harm

Offutt, J., Alluri, C. An industrial study of applying input space partitioning to test financial calculation engines . Empir Software Eng 19, 558–581 (2014).
https://doi.org/10.1007/s10664-012-9229-5