

Software Engineering Testing

Input Space Partition Testing

# Input Space Coverage

## Four Structures for Modeling Software

**Graphs**

Applied to

Source

Specs

Design

Use cases

**Logic**

Applied to

Source

FSMs

Specs

DNF

**Input Space**

**Syntax**

Applied to

Source

Models

Integ

Input

## Input Domains

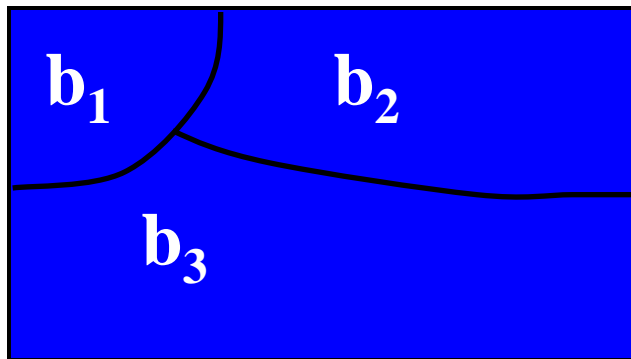
- The input domain to a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be infinite
- Testing is fundamentally about choosing finite sets of values from the input domain
- *Input parameters* define the scope of the 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  $q$  of  $D$
- The partition  $q$  defines a set of blocks,  $Bq = b_1, b_2, \dots, b_Q$
- The partition must satisfy two properties :
  1. blocks must be pairwise disjoint (no overlap)
  2. together the blocks cover the domain  $D$  (complete)



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

$$\bigcup_{b \in B_q} b = D$$

## 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 characteristics
  - 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 is easy to get wrong
- Consider the “order of file F”

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

but ... something's fishy ...

What if the file is of length 1?

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

### Solution:

Each characteristic should address just one property

File F sorted ascending

- $b_1$  = true
- $b_2$  = false

File F sorted descending

- $b_1$  = true
- $b_2$  = 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 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**
  - 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

## 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

***Input Domain Model (IDM)***

## 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 last lecture (on Logic Coverage)**

**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 :
  - Preconditions and postconditions
  - 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

## Steps 1 & 2 : Interface vs Functionality-Based

```
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
```

### Interface-Based Approach

Two parameters : list, element

#### Characteristics :

list is null (block1 = true, block2 = false)

list is empty (block1 = true, block2 = false)

### Functionality-Based Approach

Two parameters : list, element

#### Characteristics :

number of occurrences of element in list  
(0, 1, >1)

element occurs first in list  
(true, false)

element occurs last in list  
(true, 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**, had one testable function and three integer inputs

### First Characterization of TriTyp's Inputs

Characteristic	$b_1$	$b_2$	$b_3$
$q_1$ = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
$q_2$ = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
$q_3$ = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of  $3 \times 3 \times 3 = 27$  tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests ...

## Second Characterization of TriTyp's Inputs

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
$q_1$ = "Refinement of $q_1$ "	greater than 1	equal to 1	equal to 0	less than 0
$q_2$ = "Refinement of $q_2$ "	greater than 1	equal to 1	equal to 0	less than 0
$q_3$ = "Refinement of $q_3$ "	greater than 1	equal to 1	equal to 0	less than 0

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

## Possible values for partition $q_1$

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
Side1	2	1	0	-1

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 represent a triangle

### Geometric Characterization of TriTyp's Inputs

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

- 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

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

## Functionality-Based IDM – TriTyp (*cont*)

- Values for this partitioning can be chosen as

Possible values for geometric partition $q_1$				
Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

## Functionality-Based IDM – TriTyp (*cont*)

- A different approach would be to break the geometric characterization into four separate characteristics

### Four Characteristics for TriTyp

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

## Using More than One IDM

- Some programs may have dozens or even hundreds of parameters
- Create several small IDMs
  - A divide-and-conquer approach
- Different parts of the software can be tested with different amounts of rigor
  - For example, some IDMs may include a lot of invalid values
- It is okay if the different IDMs overlap
  - The same variable may appear in more than one IDM

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

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

## 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 (EC) : One value from each block for each partition must be used in at least one test case.**

- **Number of tests is the number of blocks in the largest partition**

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

**For TriTyp: 2, 2, 2**

**1, 1, 1**

**0, 0, 0**

**-1, -1, -1**



## ISP Criteria – Pair-Wise

- Each choice yields few tests – cheap but perhaps ineffective
- Another approach asks values to be combined with other values

**Pair-Wise (PW) : A value from each block for each partition must be combined with a value from every block for each other partition.**

- Number of tests is at least the product of two largest partition

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

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

## ISP Criteria –T-Wise

- A natural extension is to require combinations of  $t$  values instead of 2

**t-Wise (TW) : A value from each block for each group of  $t$  partitions must be combined.**

- **Number of tests is the number of choice tests raised to the power  $t$**

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

- **If  $t$  is the number of partitions  $Q$ , then all combinations**
- **That is ...  $Q$ -wise = AC**
- **$t$ -wise is expensive and benefits are not clear**

## ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program

**Base Choice (BC) : A base choice block is chosen for each partition, and a base test is formed by using the base choice for each partition. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other parameter.**

- Number of tests is one base test + one test for each other block

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

For TriTyp: <u>Base</u>	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

- Testers sometimes have more than one logical base choice

**Multiple Base Choice (MBC)** : One or more base choice blocks are chosen for each partition, and base tests are formed by using each base choice for each partition. 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 parameter.

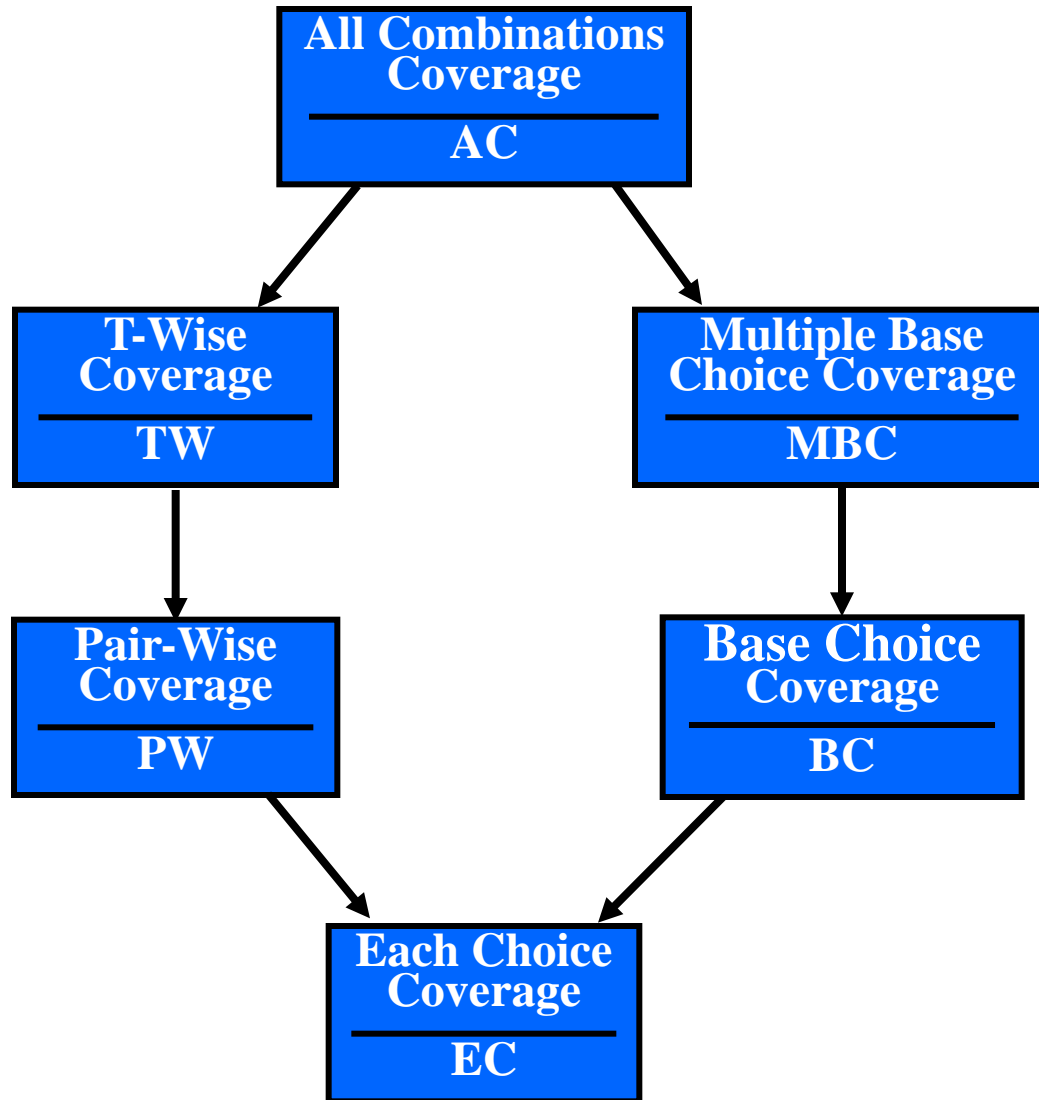
- If there are  $M$  base tests and  $m_i$  base choices for each partition:

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

**For TriTyp: Base**

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

# ISP Coverage Criteria Subsumption



## 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
  - AC, PW, TW : Drop the infeasible pairs
  - BC, MBC : 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

**Blocks from other characteristics are irrelevant**

### Characteristics   Partitions:

- |            |   |
|------------|---|
| • Length   | • Len    { 0, 1, 2..100, 101..MAXINT }          |
| • Type of  | • Type    { int, char, string, other }          |
| • Max val  | • Max    { ≤0, 1, >1, 'a', 'Z', 'b', ..., 'Y' } |
| • Min val  | • Min    { ... }                                |
| • Position | • Max Pos { 1, 2 .. Len-1, Len }                |
| • Position | • Min Pos { 1, 2 .. Len-1, Len }                |

**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



# References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008