Introduction to Software Testing
Chapter 6
Input Space Partition Testing

Paul Ammann & Jeff Offutt

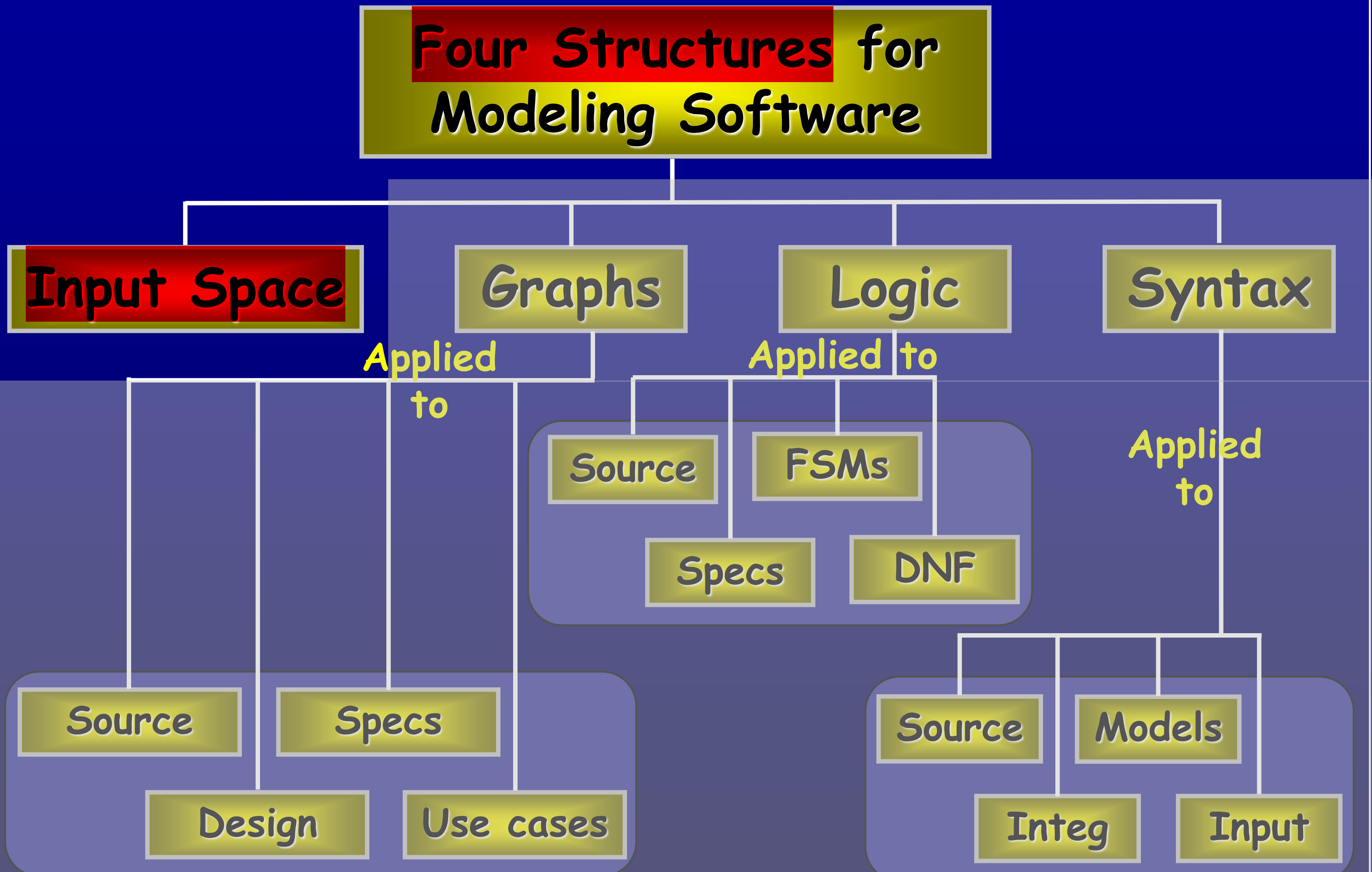
http://www.cs.gmu.edu/~offutt/softwaretest/

Engineers take ideas invented by quick thinkers

and

build products for slow thinkers.

Ch. 6 : Input Space Coverage



Input Space Partitioning

- Takes the view that we can directly divide the input space according to logical partitioning of the inputs.
- Is independent of the RIPR model–we only use the input space of the software under test.

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

Input Domains

- The input domain for 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
- Input domains are partitioned into regions (blocks)
- At least one value is chosen from each block

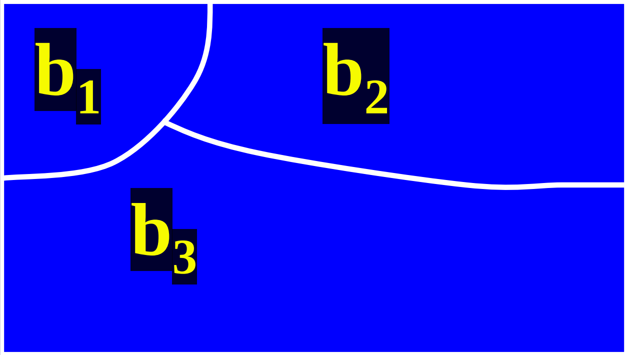
Partitioning Domains

- *Domain D*
- *Partition scheme 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)

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

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

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



Using Partitions – Assumptions

- Choose a value from each block
- 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 list F (sorted, inverse sorted, arbitrary, …)
  - Input device (DVD, CD, VCR, computer, …)

Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the characteristic "*order of elements in list F*"

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

Design blocks for that characteristic

but … something's fishy …

What if the list is of length 1?

Can you find the problem?

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

Solution:

Each characteristic should address just one property

Can you think of a solution?

C1: List F sorted ascending
- c1.b1 = true
- c1.b2 = false

C2: List F sorted descending
- c2.b1 = true
- c2.b2 = false

C1: List F sorted ascending
b1 = sorted in ascending order
b2 = sorted in descending order
b3 = arbitrary order
The list will be in all three blocks
That is, disjointness is not satisfied

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
- Different alternatives should be considered
- We model the input domain in five steps …
  - Steps 1 and 2 move us from the implementation abstraction level to the design abstraction level (from chapter 2)
  - Steps 3 & 4 are entirely at the design abstraction level
  - Step 5 brings us back down to the implementation abstraction level

Modeling the Input Domain

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

Each usecase is associated with a specific intended functionality of the system, so it is very likely that the usecase designers have useful characteristics in mind that are relevant to developing test cases.

- S
  a
  - 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

Step 2 : Find all the parameters that can affect the behavior of a given testable function.
Often fairly straightforward, even mechanical

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 using 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
- Easy to identify characteristics.
- Some domain and semantic information won't be used
  - Could lead to an incomplete IDM
  - Not all the information available to the test engineer will be reflected in the interface domain model.
- Ignores relationships among parameters
  - Important sub-combinations may be missed.

1. Interface-Based Example

- Consider method *triang()* from class *TriangleType* on the book website :
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java

```
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
```

The 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
  - Can start early in development.

2. Functionality-Based Approach(Cnt'd)

- May be yields better test cases than the interface-based approach because the input domain models include more semantic information.
- Transferring more semantic information from the specification to the IDM makes it more likely to generate expected results for the test cases.
- The same parameter may appear in multiple characteristics, or characteristics do not map to single parameters of the software interface
  - so it's harder to translate values to test cases

2. Functionality-Based Example

- Again, consider method *triang()* from class *TriangleType* :

The three parameters represent a *triangle*

The 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

The tester should apply input space partitioning by using *domain knowledge* about the problem, not the implementation.

However, in practice, the code may be all that is available.

Overall, the more semantic information the test engineer can incorporate into characteristics, the better the resulting test set is likely to be.

Steps 1 & 2—Interface & 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
- Partitioning often flows directly from the definition of characteristics and both steps are 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 –*triang()*

- *triang*() has 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*3*3 = 27 tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests …

Interface-Based IDM—*triang*()

Second Characterization of triang()'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*4*4 = 64 tests
- 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—*triang*()

- 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

Equilateral is also isosceles !
We need to refine the example to make characteristics valid

Geometric Characterization of *triang*()'s Inputs

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

دارای اضلاع نامساوی

متساوی الساقین

متساوی الاضلاع

- Equilateral is also is[osceles]
- We need to refine t[he example to make chara]cteristics valid

What's wrong with this partitioning?

Correct Geometric Characterization of *triang*()'s Inputs

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

Functionality-Based IDM—*triang*()

- 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—*triang*()

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

Four Characteristics for *triang*()

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