INPUT SPACE PARTITIONING

Introduction

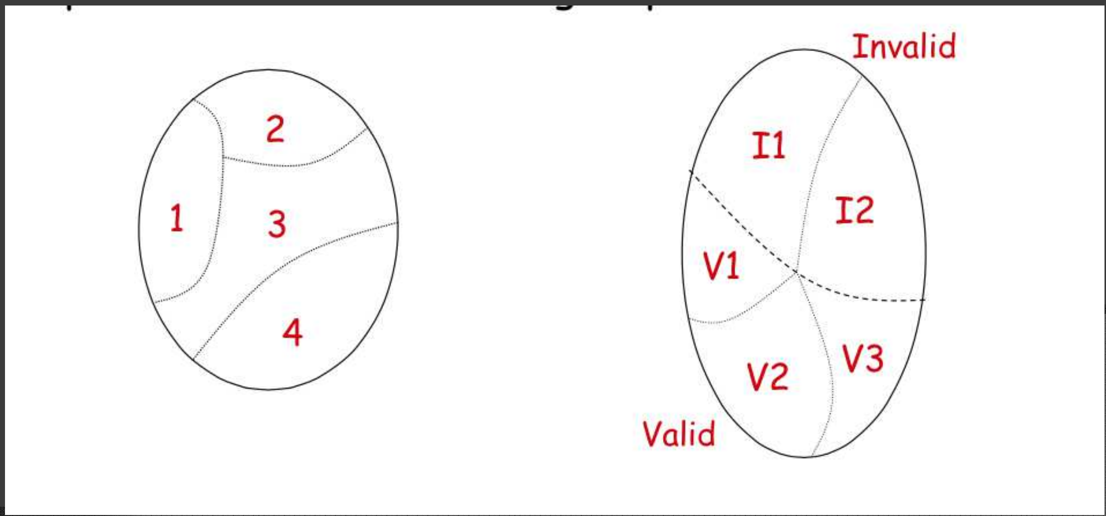
- Testing is about choosing elements from *input domain*.
- The *input domain* of a program consists of all possible inputs that could be taken by the program.
  - Easy to get started, based on description of the inputs

Test Selection Problem

- 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 of the input domain such that the execution of T will reveal as many error as possible.

Partitioning

- The input domain partitioned into region that are contained equally useful values for testing, and values are selected from each region.
  1. The partition must cover the entire domain (completeness)
  2. The blocks must not overlap (disjoint)



Input Domain Modeling (IDM)

Step 1: Identify the input domain

- Read the requirements carefully and identify all input and output variables, any conditions associated with their use.

Step 2: Identify equivalence classes

- Partition the set of values of each variable into disjoint subsets, based on the expected behavior.

Step 3: Combine equivalence classes.

- Use some well-defined strategies to avoid potential explosion

Step 4: Remove infeasible combinations of equivalence classes

Different Approaches to IDM

- Interface-Based IDM

Strength:

1. Easy to identify characteristics
2. Easy to translate abstract test cases to concrete test case

Weakness

1. IDM may be incomplete
2. Each parameter analyzed in isolation so that important sub combination may be missed

Different Approaches to IDM

- functionality-Based IDM

Strength:

1. Use semantics and domain knowledge
2. Requirements are available so test cases generation can start early

Weakness

1. Hard to identify characteristics
2. Hard to translate abstract test cases to concrete test cases

Example

public boolean findElement (List list, Element element)

//If list or element is null throw NullPointerException else returns true if element is in the list , false otherwise

List Characteristics

Interface-based	
Characteristics	**Blocks and Values**
List is null	b1 = true
	b2 = false
List is empty	b1 = true
	b2 = false

Functionality-based	
Characteristics	**Blocks and Values**
Number of occurrences of element in list	b1 = 0
	b2 = 1
	b3 = more than 1
Element occurs first in list	b1 = true
	b2 = false

Identify Characteristics

- The interface-based approach develops characteristics directly from input parameters
- The functionality-based approach develops characteristics from functional or behavioral view

Choosing Block and Values

- Valid values
- Boundaries
- Normal use
- Invalid values
- Special values
- Missing partitions
- Overlapping partitions

Functionality-Based

Geometric partitioning of TriTyp's inputs				
Partition	b1	b2	b3	b4
Geometric Classification	*Scalene*	*Isosceles*	*Equilateral*	*invalid*

Geometric partitioning of TriTyp's inputs				
Partition	b1	b2	b3	b4
Geometric Classification	*Scalene*	*Isosceles, not equilateral*	*Equilatera l*	*invalid*

Geometric partitioning of TriTyp's inputs				
Partition	b1	b2	b3	b4
triangle	*(4,5,6)*	(3,3,4)	*(3,3,3)*	*(3,4,8)*

Recommended Approach

Scalene	***Isosceles***	***Equilateral***	**Valid**
true	true	true	true
false	false	false	false

- The fact that choosing Equilateral = true also means choosing Isosceles = true is then simply a ***<u>constraint</u>***.
- This approach satisfies the disjointness and completeness properties.

Combination Strategies Criteria

- The behavior of a software application may be affected by many factors, e.g., input parameters, environment configurations, and state variables.
- Techniques like equivalence partitioning and boundary-value analysis can be used to identify the possible values of individual factors.
- It is impractical to test all possible combinations of values of all those factors. (Why?)

Combinatorial Explosion

- Assume that an application has 10 parameters, each of which can take 5 values. How many possible combinations?

Combinatorial Design

- Instead of testing all possible combinations, a subset of combinations is generated to satisfy some well-defined combination strategies.
- A key observation is that not every factor contributes to every fault, and it is often the case that a fault is caused by interactions among a few factors.
- Combinatorial design can dramatically reduce the number of combinations to be covered but remains very effective in terms of fault detection.

Fault Model

- A t-way interaction fault is a fault that is triggered by a certain combination of t input values
- A simple fault is a t-way fault where t = 1; a pairwise fault is a t-way fault where t = 2.
- In practice, a majority of software faults consist of simple and pairwise faults.

Example – Pairwise Fault

```
begin
        int x, y, z;
        input (x, y, z);
        if (x == x1 and y == y2)
                output (f(x, y, z));
        else if (x == x2 and y == y1)
                output (g(x, y));
        else
                output (f(x, y, z) + g(x, y))
End
```

Expected: x = x1 and y = y1 => f(x, y, z) – g(x, y);
x = x2, y = y2 => f(x, y, z) + g(x, y)

Example – 3-way Fault

```
// assume x, y ! {-1, 1}, and z ! {0, 1}
begin
	int x, y, z, p;
	input (x, y, z);
	p = (x + y) * z // should be p = (x – y) * z
	if (p >= 0)
		output (f(x, y, z));
	else
		output (g(x, y));
end
```

All Combinations Coverage

- Every possible combination of values of the parameters must be covered
- For example, if we have three parameters P1 = (A, B), P2 = (1, 2, 3), and P3 = (x, y), then all combinations coverage requires 12 tests: {(A, 1, x), (A, 1, y), (A, 2, x), (A, 2, y), (A, 3, x), (A, 3, y), (B, 1, x), (B, 1, y), (B, 2, x), (B, 2, y), (B, 3, x), (B, 3, y)}

Each Choice Coverage

- Each parameter value must be covered in at least one test case.
- Consider the previous example, a test set that satisfies each choice coverage is the following: {(A, 1, x), (B, 2, y), (A, 3, x)}

Pairwise Coverage

- Given any two parameters, every combination of values of these two parameters are covered in at least one test case.
- A pairwise test set of the previous example is:

	P1	P2	P3
	A	1	x
	A	2	x
	A	3	x
	A	-	y
	B	1	y
	B	2	y
	B	3	y
	B	-	x

T-Wise Coverage

- Given any t parameters, every combination of values of these t parameters must be covered in at least one test case.
- For example, a 3-wise coverage requires every triple be covered in at least one test case.
- Note that all combinations, each choice, and pairwise coverage can be considered to be a special case of t-wise coverage.

Base Choice Coverage

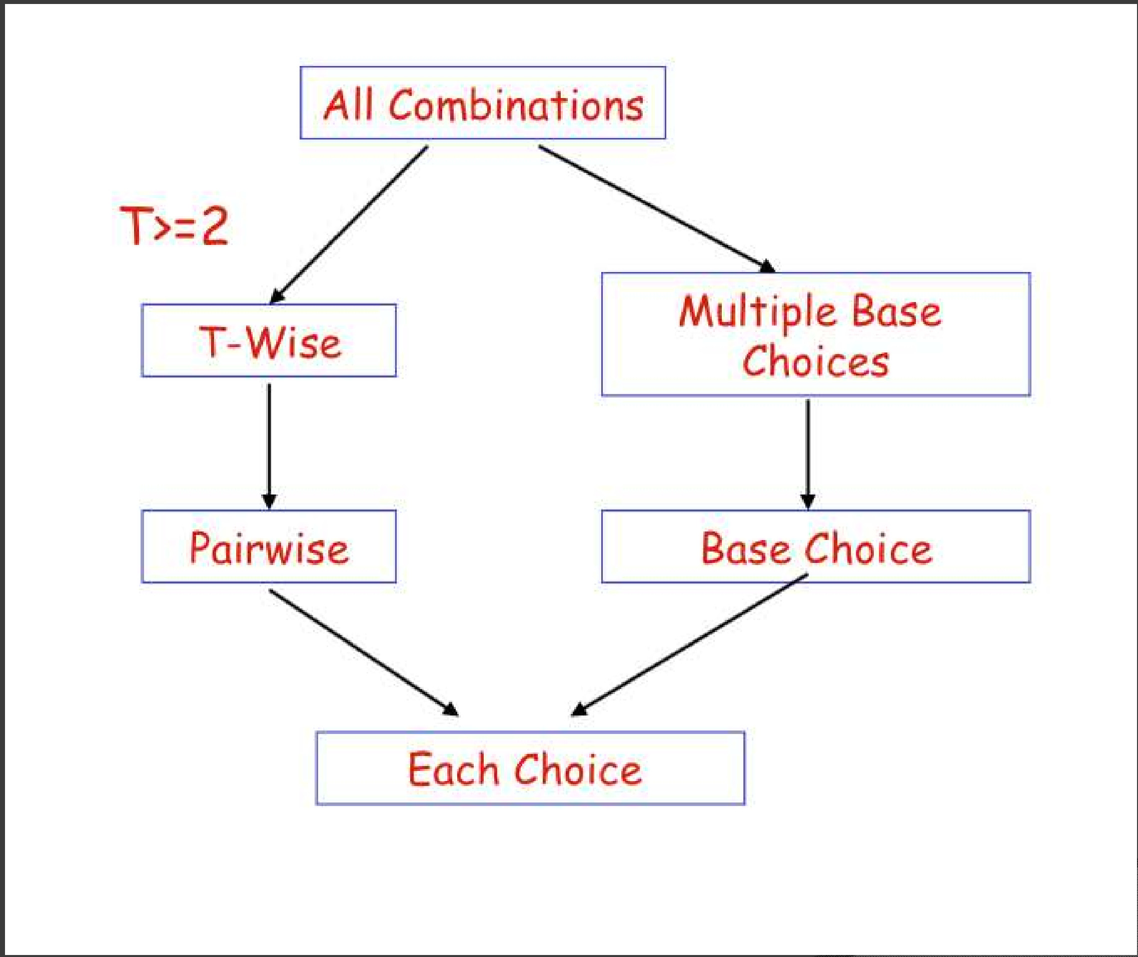
- For each parameter, one of the possible values is designated as a base choice of the parameter
- A base test is formed by using the base choice for each parameter
- Subsequent tests are chosen by holding all base choices constant, except for one, which is replaced using a non-base choice of the corresponding parameter:

P1	P2	P3
A	1	x
B	1	x
A	2	x
A	3	x
A	1	y

Multiple Base Choices Coverage

- At least one, and possibly more, base choices are designated for each parameter.
- The notions of a base test and subsequent tests are defined in the same as Base Choice.

Subsumption Relation



Pairwise Test Generation

Why Pairwise?

- Many faults are caused by the interactions between two parameters
  - 92% statement coverage, 85% branch coverage
- Not practical to cover all the parameter interactions
  - Consider a system with n parameter, each with m values. How many interactions to be covered?
- A trade-off must be made between test effort and fault detection
  - For a system with 20 parameters each with 15 values, pairwise testing only requires less than 412 tests, whereas exhaustive testing requires 1520 tests.

Example

Consider a system with the following parameters and values:

- parameter A has values A1 and A2
- parameter B has values B1 and B2, and
- parameter C has values C1, C2, and C3

Example cont.,

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C3
A2	B1	C1
A1	B2	C2
A1	B1	C3

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C2
A2	B1	C1
A1	B1	C2
A1	B1	C3
A2	B2	C3

The IPO Strategy

- First generate a pairwise test set for the first two parameters, then for the first three parameters, and so on
- A pairwise test set for the first n parameters is built by extending the test set for the first n – 1 parameters
  - Horizontal growth: Extend each existing test case by adding one value of the new parameter
  - Vertical growth: Adds new tests, if necessary

Summary

- Combinatorial testing makes an excellent trade- off between test effort and test effectiveness.
- Pairwise testing can often reduce the number of dramatically, but it can still detect faults effectively.
- The IPO strategy constructs a pairwise test set incrementally, one parameter at a time.
- In practice, some combinations may be invalid from the domain semantics, and must be excluded, e.g., by means of constraint processing.