
SYSC 4101 / SYSC 5105

Input Space Partitioning—Criteria Part I

Test Criteria Based on Structure [Offutt]

- Graphs



- Logical Expressions

(not X or not Y) and A and B

- Input Domain Characterization

Describes the input domain of the software under test (method, component, system)

A: {0,1,>1}

B: {600,700,800}

C: {swe,cs,isa,ifs}

if (x>y)

z = x - y;

else

z = 2 * x

- Syntactic Structures

Basic Principles

- Choosing elements from the input space, from the specification
- The specification provides information on
 - The input parameters, but not only (see later)
 - The allowed ranges of each parameter: **Equivalence class partitioning + Boundary value analysis**
 - What are the boundaries of the range? (where faults often are)
 - What is a within-range value?
 - What is an out-of-range value? (robustness)
- Equivalence classes
 - The behavior of the software is assumed to be the same for all the values of the class.
 - Sense of completeness (each class is exercised)
 - No redundancy (one input per class)
- Account for the test engineer expertise

Basic Principles—Illustrated

- Consider a function that takes an integer as input with the following specification:

- If input is strictly negative, behavior A is triggered
- If input is in range $[0, 10]$, behavior B is triggered
- If input is in range $]10, 20]$, behavior C is triggered
- If input is strictly greater than 20, behavior D is triggered

Any value in this range triggers behavior D. It should be enough to use one (and only one) of those values

- Equivalence classes—4 inputs



- Boundary value analysis added—7 inputs (faults happen at boundaries)



- Further boundary value analysis—13 inputs (faults at/around boundaries)



Step-by-Step Procedure

1. Identify functions/functionalities to be tested
2. For each **function**, identify
 - A. The **parameters** of the function
 - These are part of the function interface/specification
 - B. The **environment variables**
 - What, in the (execution) environment of the function, could impact the function's behavior?
 - These often come from the expertise of the test engineer
3. For each parameter or environment variable, identify the **characteristics** of interest from a testing point of view
 - They are stated in plain language (**no actual value**)
 - Characteristics are “*major properties*” of the parameter (or environment variable)
 - “Major property” means that the characteristic is relevant from the point of view of the function's behavior
 - They represent **orthogonal perspectives** (\neq perspectives per category)

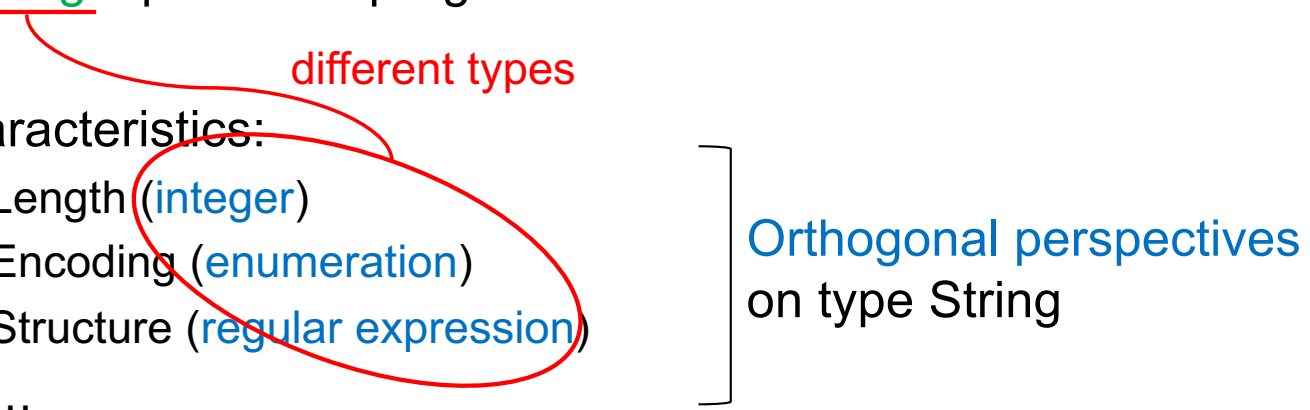
Step-by-Step Procedure (cont.)

4. For each characteristic, identify **blocks**
 - A characteristic implicitly specifies a domain of values
(not necessarily values of a parameter of the function)
 - Blocks are a partition of the characteristic's domain into sets of values
 - Blocks are often equivalence classes / boundaries
 - Advice: try to state them in plain language (instead of individual values)
5. For each block, identify representative values
 - Values are not used until the very, very end of the process (see test frames and test cases)

```
Tested function
  └─Function's parameters and environment variables
      └─Parameter/variable characteristics
          └─Characteristic's blocks
              └─Values of block
```

This is the input model, the test model.

On Characteristics ...

- Characteristics can specify various **types** of values
 - These types are **different** from input parameter or environment variable type
 - Example:
 - A string input to the program under test
 - Characteristics:
 - Length (**integer**)
 - Encoding (**enumeration**)
 - Structure (**regular expression**)
 - ...
- different types**
- Orthogonal perspectives
on type String**
- 

Template, Type-Specific Equivalence Classes (and Boundaries)

- Range
 - One class with values inside the range
 - Two classes with values outside the range
 - Possibility to add classes for the boundaries and values immediately around boundaries
- String
 - At least one class with legal strings (depending on what is considered “legal”)
 - At least one class with illegal strings
 - It is sometimes possible to identify what a boundary is (e.g., all digits)
 - Possibility to also consider the contents of a legal string and further identify classes (can different contents impact behaviour?)

Template, Type-Specific Equivalence Classes (and Boundaries)

- Array
 - One class containing all legal arrays
 - One class for the empty array
 - One class for arrays larger than expected
 - Possibility to add classes for the boundary of the array size and values immediately around such boundary
 - Possibility to add classes depending on the structural contents of the array (if specified, if leads to alternative behaviours)
- Enumeration
 - Each value is a separate class
 - May be able to group enumeration elements (depending on triggered behaviours)

Quality of the Input Model?

It is paramount to check the Input Model !

- Any missing information about how the function behaves ?
 - Can be very subjective
- Are the blocks of a characteristic disjoint ?
 - Selecting one block excludes the others
 - An input value for a parameter that fits the specification of a characteristics can only belong to one block of that characteristic
- Are the blocks of a characteristic complete ?
 - The blocks cover the entire input space (implicitly defined by the characteristic)
 - The blocks of a characteristic, together, represent all the possible values the parameter can have along the perspective (characteristics)
- Tool support for this verification (i.e., disjointness, completeness) is possible.

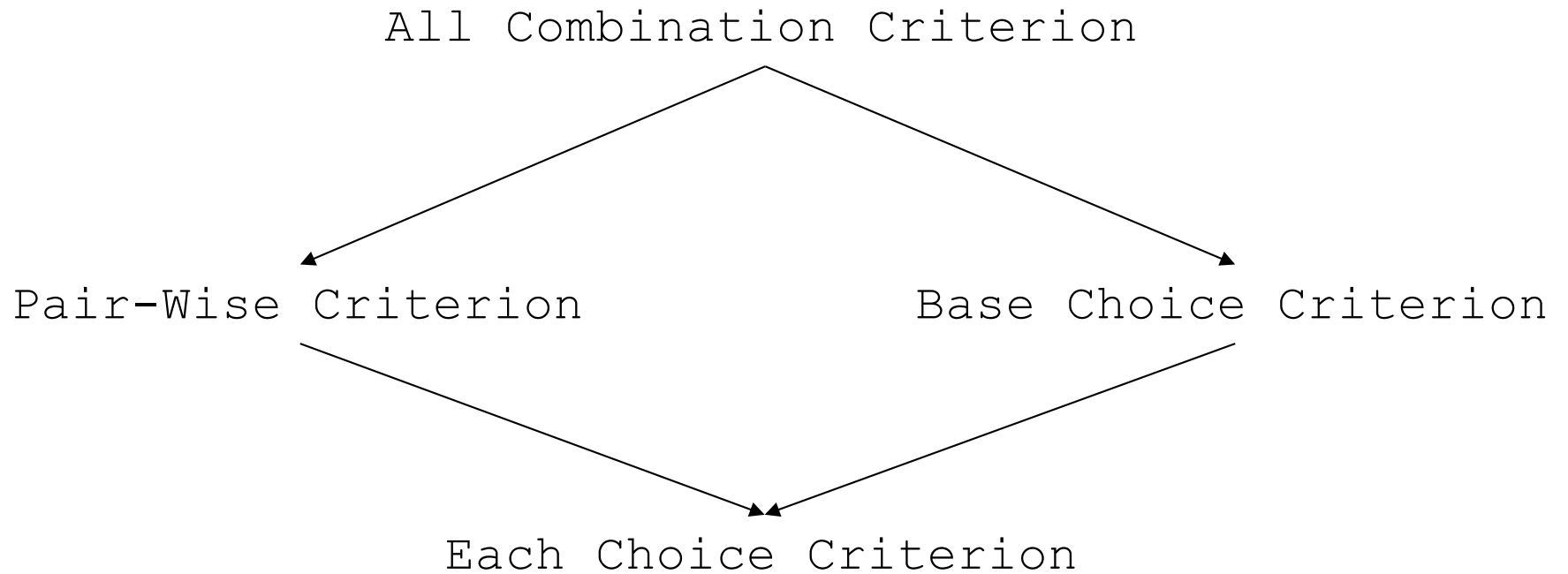
Criteria → Test Frames (combination of blocks)

- All Combinations Criterion
 - All combinations of blocks from all characteristics must be used at least once.
- Each Choice/Block Criterion
 - Each block of each characteristic must be used at least once.
- Pair-Wise Criterion
 - Each block of each characteristic must be combined with a block for each other characteristic.
(Identify pairs of blocks first, and then identify a minimum set of combinations of blocks to “exercise” the pairs.)
- Base Choice Criterion
 - A base choice block is chosen for each characteristic, and a base test frame is formed by using the base choice for each characteristic.
 - Subsequent test frames are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

Criteria → Test Frames → Test Cases

- A **test frame** is a combination of blocks from characteristics
 - Cannot combine two blocks from the same characteristic
 - Refer to conditions equivalence classes (blocks) must satisfy.
- A set of test frames **satisfy** a criterion (notion of adequacy)
- A test frame **is not** a test case
- A test frame is a **test case specification**
 - It tells what a test case should look like
 - It provides conditions that test inputs (the test case) should satisfy
- **From a test frame to a test case**
 - Need to select input values such that the conditions imposed by the test frame, i.e., its combination of blocks, are satisfied
 - Create one test case for each test frame

Criteria Comparison (subsumption)



Constraints Among Partitions

- Some combinations of blocks may be infeasible
- Constraints
 - Relations between blocks of different characteristics
 - Recall blocks for a given characteristic are disjoint (cannot be combined)
 - So we only define constraints between blocks of different characteristics
 - Can constrain different blocks of different characteristics
- *Properties, Selectors* associated with blocks
 - Specifying (with a name) a property.
 - Notation: [PropertyName]
 - Selection of (in)compatible property(ies).
 - Notation: [if PropertyName], [if propNameA and propNameB], [if not propName] ...
- Special cases: [Error], and [Single]

Constraints

- [Error]
 - It is assumed that if the parameter or environment variable has this particular value (values from the block), any call of the function using that block will result in the same error.
 - A block marked with [Error] is not combined with blocks in the other categories to create *test frames*.
 - Meaning that blocks from other categories do not matter
 - During the test, the tester can set the test's other parameters and environment conditions at will, e.g., base blocks.
- [Single]
 - This notation is intended to describe special, unusual, or redundant conditions that do not have to be combined with all possible blocks.
 - A block marked with [Single] is not combined with blocks in the other categories to create *test frames*.
 - A judgment by the tester that the marked block can be adequately tested with only one test case.

Conclusions

- Identifying parameters and environments conditions, characteristics, and blocks, heavily relies on the **experience/expertise of the tester**
- Makes testing decisions explicit (e.g., constraints), open for review
- Once the first step is completed, i.e., identifying characteristics, blocks and constraints, the technique is straightforward and can be automated
 - Using a criterion to generate test frames and then test cases (i.e., identifying test inputs)
- The technique for test case reduction (thanks to constraints) makes it useful for practical testing

Functionality-Based Input Domain Modeling vs. Category Partition

Mapping of terminology...

Between the terminology of the textbook (left) and the terminology of the original publication defining the technique.

Functionality-Based Input Domain Modeling

- Tested function
- Characteristics
- Blocks
- Values

Category Partition

- Tested function
- Parameters + Environment variables
- Categories
- Choices
- Values