Functional testing

Software Reliability and Testing - Barbara Russo

SwSE - Software and Systems Engineering

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Functional testing (Black Box testing)

- Deriving test cases from **program specifications**
- Functional testing does not exercise testing based on design or code (white-box testing)
- Functional testing is the baseline technique for any other testing strategy
- **It is independent from any implementation (design or code)**

- Why not simply picking random input to design test cases?

Random testing

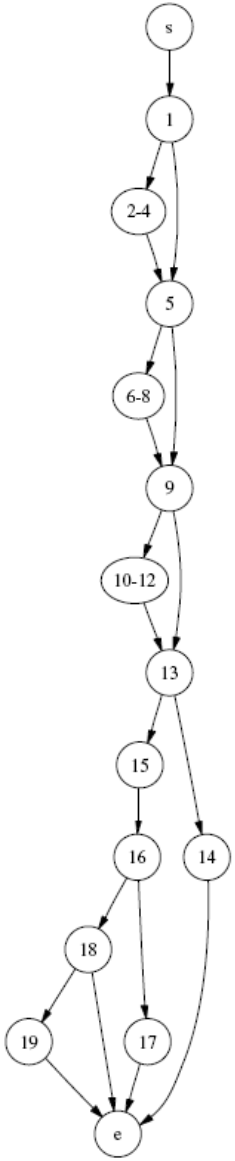
- Pick inputs uniformly
- Avoids designer bias
  - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person),
- But treat all inputs as equally valuable
- It limits costs, it does not require much knowledge of the input

Random testing

- It can be automised and produces more test cases than partition testing
- Limitation: It is not able to pick specific / critical input values as it treats all the inputs the same

- Random testing: execute the program with random inputs and observe the code coverage
- Weakness: structures having a low probability of being executed are often not covered

```
s       int tri_type(int a, int b, int c)
        {
            int type;

1           if (a > b)
2-4         {   int t = a; a = b; b = t;    }

5           if (a > c)
6-8         {   int t = a; a = c; c = t;    }

9           if (b > c)
10-12       {   int t = b; b = c; c = t;    }

13          if (a + b <= c)
            {
14              type = NOT_A_TRIANGLE;
            }
            else
            {
15              type = SCALENE;
16              if (a == b && b == c)
                {
17                  type = EQUILATERAL;
                }
18              else if (a == b || b == c)
                {
19                  type = ISOSCELES;
                }
            }

e           return type;
        }
```



Exercise

- Exercise: use any black box testing strategy to create test cases for the following method
- Pair is a class of ordered pairs of double numbers x, y

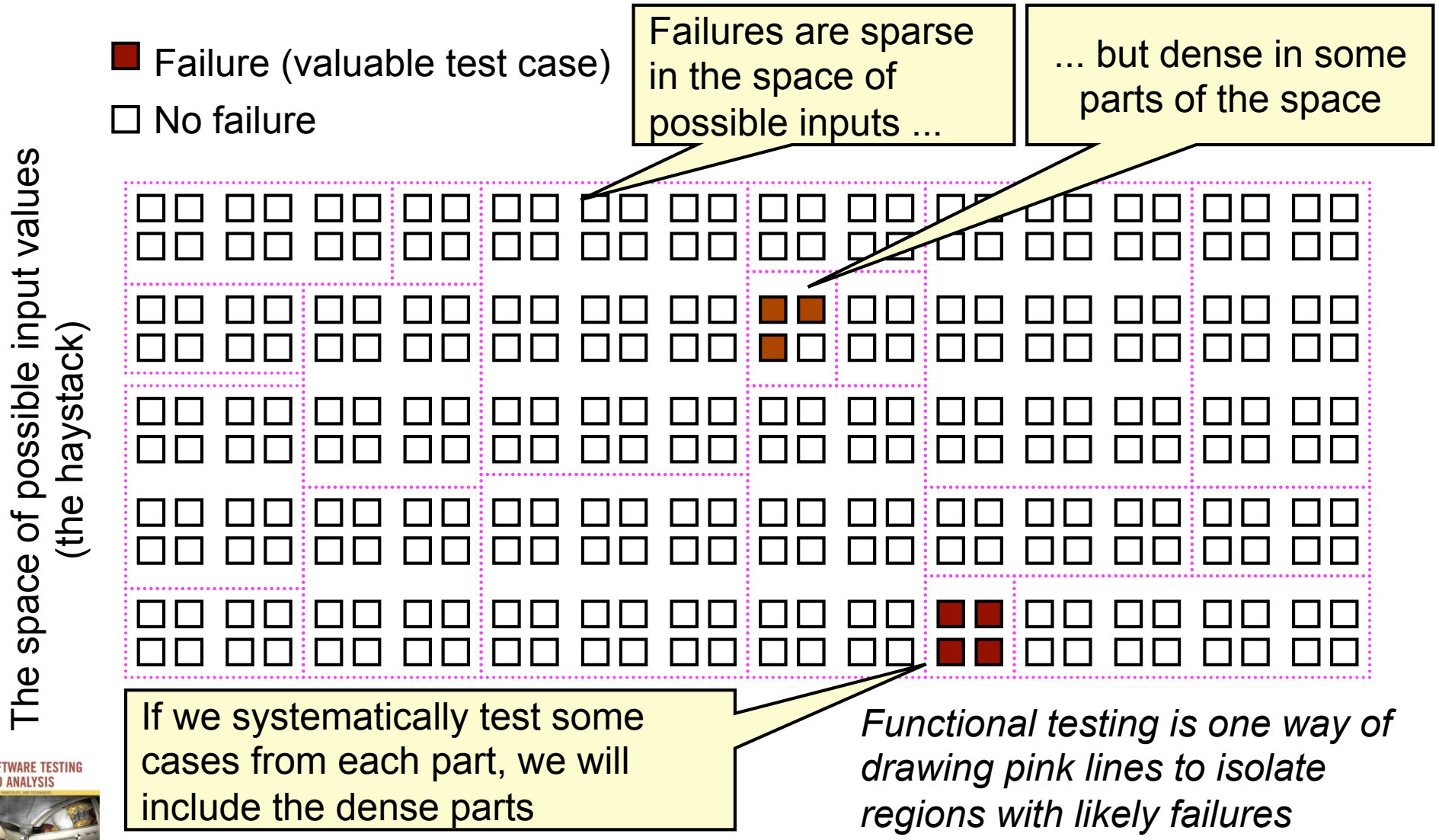
Exercise

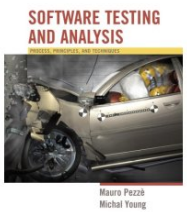
```
public static Pair solve(double a, double b, double c){
   double q= b*b-4*a*c;
   if (a!=0 && q>0){
       Pair.x =(0-b+Math.sqrt(q))/2*a
       Pair.y =(0-b-Math.sqrt(q))/2*a
   } else if (q==0){ // Bug: why?
       Pair.x =(0-b)/2*a
       Pair.y =(0-b)/2*a
  } else { Pair.x=-1; Pair.y=-1;}
}
```

Discussion

- Random test case generation is fine to test $q > 0$
- Random sampling unlikely picks a=0.0 and b=0.0 as
  - Failing values are sparse in some parts of the input space, but dense in other parts

Systematic Partition Testing





The partition principle

- In principle, it divides (infinite) input into a finite number of classes where each class can be homogeneously associated to **one output success** or **failure**

The partition principle

- In practice, **Quasi-partition testing**: it divides the infinite set of possible test inputs into a finite set of classes (**also non disjoint and with no homogeneity**) with the purpose of drawing a test case per class
- Partition divides input into a finite set of classes of program behaviour

Partition- selecting representatives

- Tests are designed on representatives (input) of classes
- Often classes and representatives are defined by using expert opinion

Partition- selecting representatives

- We do not know which testing technique would be likelier to reveal faults
- Repeating the **same test case** is **less likelier** to find a fault than exercising a **different test case**
- Running **similar test cases is less valuable** than running completely different test cases and time of testing might be limited

Partition- selecting representatives

- Example: split a buffer into lines of length 60 characters
  - **Just four test cases are available:** Buffer of length 16, 30, 40 and 100. Which test case is more valuable?
  - Note: in our selection, set of test cases skewed toward values lower than 60

Partition- selecting representatives

- To avoid this specific distribution of test cases, one can use **random generation of test cases** (with uniform distribution).
- This would be likelier to find faults in buffers with lengths greater than 60 (higher probability)

The partition principle

- Limitation: selecting representatives might be expensive
- More efficient on particular regions where fault are dense, but
  - Localising dense faulty input areas requires expert judgment or advanced techniques of **search based testing**

Boundary testing

- Boundary testing exercises values on the boundary of classes
- Boundaries are specific values
- It requires through knowledge of input, often it needs manual investigation
- Limitation: Expensive

Brute force testing

- Direct generation of test cases from specifications (brute force) might be complex and produces unacceptable results
- There is a need of a systematic general procedure

Systematic Testing

- Systematic (non-uniform):
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail often or not at all
- **Functional testing is systematic testing**

Functional testing uses partition and boundary

- Functional testing uses the specification (formal or informal) to partition the input space
  - E.g., the specification of "roots" program suggests division between cases with zero, one, and two real roots
- **Test each category, and boundaries between categories**
  - No guarantees, but experience suggests failures often lie at the boundaries

Systematic functional testing

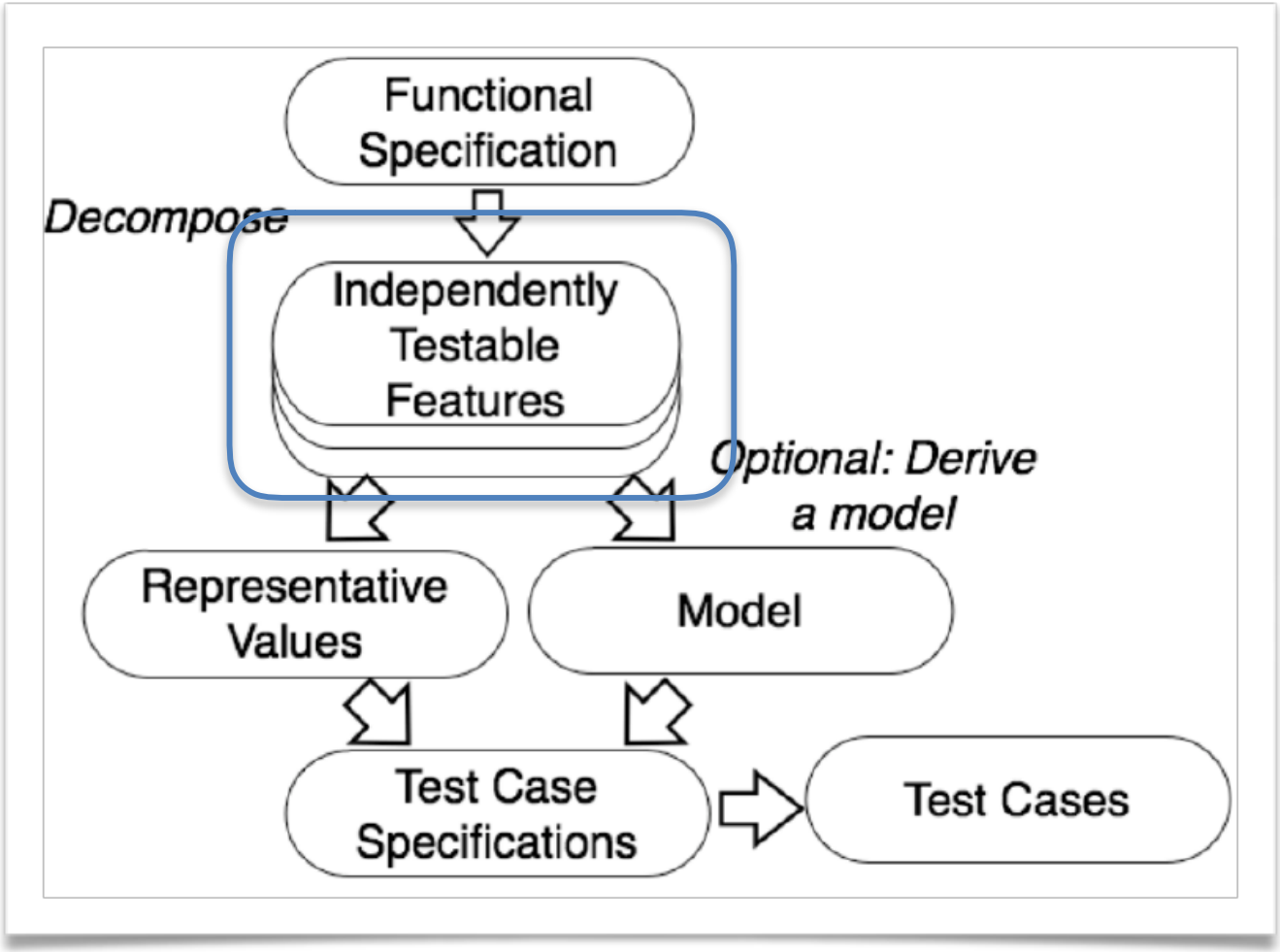
Software Reliability and Testing - Barbara Russo
SwSE - Software and Systems Engineering

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Systematic functional testing

- Systematic functional testing
  - Identify independent testable features
  - Identify variables and their characteristics
  - Identify representative values
  - Generate test case and instantiate tests
- Example - Combinatorial testing

Steps in systematic functional testing



Identify independent testable features

- Goal: partitioning specifications into features that can be tested separately
- Step 1. Identify independent testable features:
  - Parameters
  - Execution environment characteristics (like DBs that are required to execute test cases)
  - Divide features by functional use as perceived by users

Example: Diagrams of user stories in eXtreme programming

viewTransactionHistory
Acceptance Test: **ViewTransactionHistory** Priority: 2 Point: 1 Risk: 1
The user must be registered and logged in the service. The user selects an interval of time or a type of transaction to view the list of transactions. Related US: SelectTransaction

How to detect features?

- Features are identified by all the inputs that determine the **execution behaviour**
- These inputs can have different forms, they can be **explicit or implicit** inputs in the **specifications** or inputs for **some program model** (e.g., inputs that trigger the states in the finite state machine) that describes the system behaviour

How to detect features?

- Identify independent features
  - Acceptance tests in eXtreme programming identify implicit and explicit input for the user stories (not for the metaphor)
  - In eXtreme programming, the **system metaphor** identifies **implicit** form of input to augment the explicit definition in the user stories

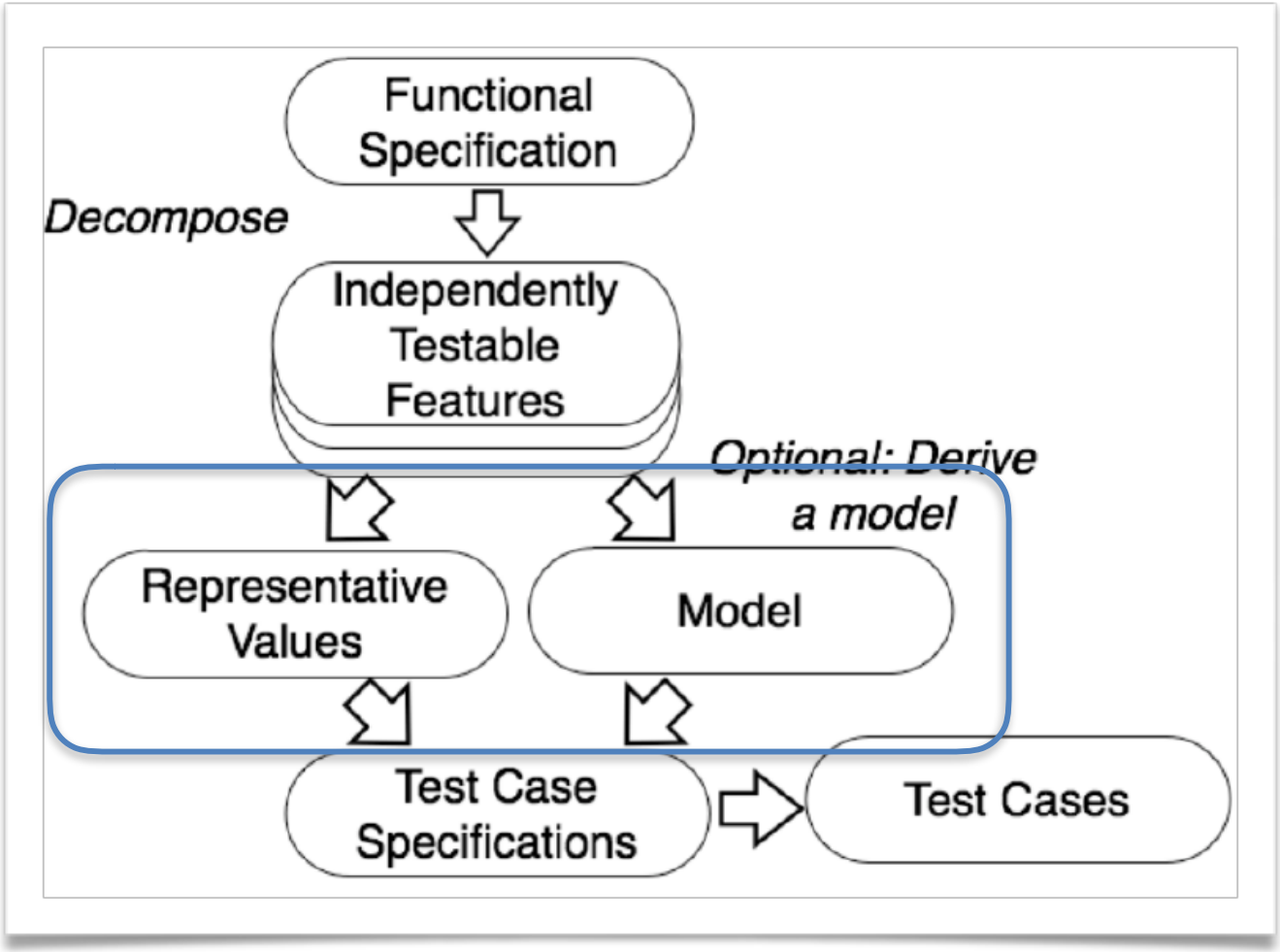
Example of features from a metaphor

- In a coffe machine scenario, the way in which ingredients are assembled and water is heated is not traceable in user stories (e.g., the grinder functionality)

Exercise

- Identify the features of a coffee machine

Steps in systematic functional testing



Select the values of input for your test cases

- Test cases are given as **combination of values** for the inputs to a unit of work
- To select the values there are two practicable ways:
  - **Representative** values of input
  - Derived from a **model**: e.g., control flow graph or finite state machine

Identify inputs and their characteristics

- Identify
  - **Implicit and explicit** parameters
  - Their elementary characteristics
  - The **environment elements and characteristics** that effect the execution of the feature in a given unit of work (like DBs that are required to execute test cases)
  - Categories of variable values defined by **system behaviour**

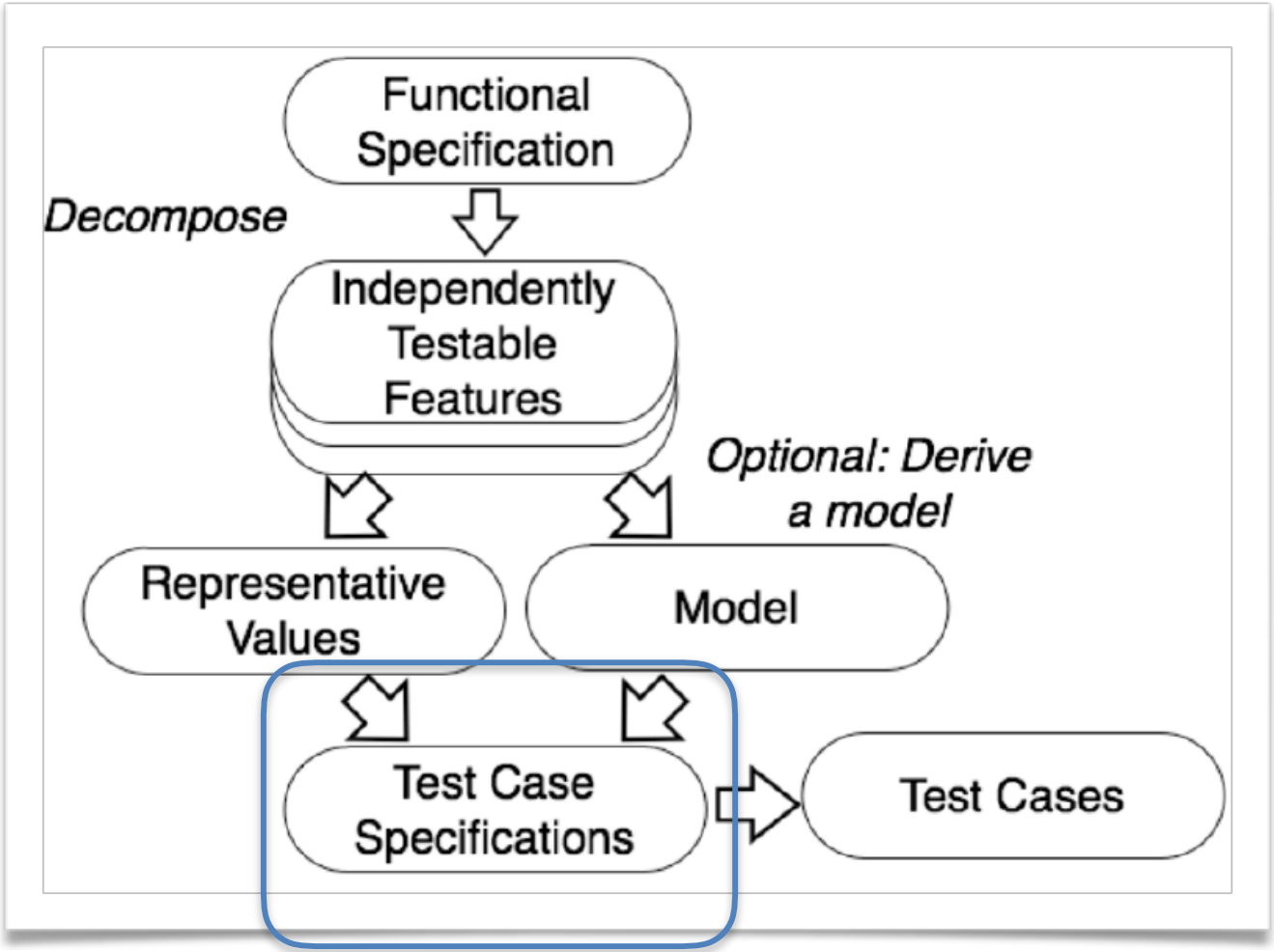
Example

- Coffe machine parameters
  - coffee (explicit): amount, temperature, poured
  - sugar (explicit): amount, type, poured
  - powder (implicit): amount
  - temperature (implicit): limit, scale
- Environmental elements
  - card: credit amount

Categories

- Coffe machine parameters
  - coffee: amount (categories: 0, positive, # over limit)
  - temperature: scale (categories: F, C, NULL)
- Environmental elements
  - card: credit amount (categories: 0, positive less than needed, positive more than needed)

Steps in systematic functional testing



Generate test case specifications

- Test Specifications are built by **combining the inputs** of the functional unit under test
- Brute force combination of inputs might be very expensive: 5 input variables with 6 values each produces 65 test cases
- In other cases, combinations are infeasible
- Reducing the inputs space is crucial to reduce the effort of test designing

Example - acceptance testing

logInTest		
Input	Description	Output (behaviors)
username= initials of name+surname password= 8 characters, alpha numeric, bounded length environmental input (DB) status= student	Preconditions: the user has accessed to the general site of the university The user introduces username and password Postcondition: the user is logged in	logged in not logged in

Example - acceptance testing

- A combination of the input values of username, password, and status defines the **test case specification**
  - For example “the user (brusso, 123456th, professor) shall not log in”.
- How many combinations of input values?
- We need to trade off between intensive testing and budget

How to reduce input combination

- Combinatorial testing, examples:
  - Pairwise combination testing
  - Category-partition testing

Pairwise combination testing

- It generates k-ways combinations of category pairs with $k < n$
- It goes blindly and does not require a specific knowledge of the domain

Category-partition testing

- Major characteristics
  - It allows test designers to add constraints and limit the number of test cases
  - Useful when we have enough knowledge of the domain and its constraints (e.g, what is valid and what is not)

Category-partition testing

- Major characteristics
  - **Flatten data structures** into **parameter characteristics** (e.g., compatibility of a selection with its slot)
  - **Filter out combinations** of values in the generation of the test case specifications:
    - First, label categories
    - Then, use labels to rule out infeasible combinations

Category-partition testing

- Labels of parameter characteristics:
  - [error],
  - [single],
  - [property: <Acronym>]

- If condition [if <Acronym>]

Category-partition testing

- The **labelling** requires expert judgment, some characteristics might be erroneous only in combination with other characteristics

Category-partition testing

- "**[error]**" : a characteristic needs to be tried in combination with non-error characteristics only once
- "**[single]**" acts as "[error]" but for any type of values (valid or not). This is not a real constraint coming form the domain, it is set by the designer to reduce the number of combinations!

Category-partition testing

- "**[property:]**" qualifies categories of values

- The **if condition** uses the properties to identify logical constraints between categories
  - These are used to rule out combinations that are not feasible

Example “Check configuration” feature

- Feature: Check the computer configuration against a reference catalogue (DB)

Example “Check configuration” feature - identify parameters

- **Variables**: Model and components
- **Model**: represents a specific product and determines a set of constraints on the available components (like screen, hard disk, processor etc)
- **Component**: a logical slot which might or might not represent a physical slot on a bus

Example “Check configuration” feature - identify parameters

- Slot compatibility with the model, slot compatibility each other
- **Components**: a collection of <slot, selection>
  - A selection is a choice of a component
- **Environmental variable**: Database of models and components that is required to execute the feature

Example “Check configuration” feature - identify variable characteristics

- Computer Model:
  - **ID key**, integer used to search and retrieve from DB, **model number, number of required slots**, and **number of optional slots**

Example “Check configuration” feature - identify variable characteristics

- Components <component, selection>
  - **number of required** / **optional slots** with non-empty selection, **compatibility** of selection in the pair with slot (e.g., 20 gigabyte of hard disk for the hard disk slot)
- External environment
  - DB: **number** of models in DB, **number** of components in DB

Example “Check configuration” feature - categories of values

- We make the example for “Components”
- We first **flatten** the data structure Components (<slot, selection>) to simple parameter characteristics:
  - Compatibility of a selection with **its slot**
  - Compatibility of a selection with **the model**
  - Matching **selection and database entry**
  - Compatibility of a selection **with another slot**

Example “Check configuration” feature - categories of values

- Then we select a category, for example “Compatibility of a selection with its slot”
  - We can represent components as an **array of compatible/non-compatible selection**. If we have n slots we have **2^n combinations** of input values

Example “Check configuration” feature - categories of values

- Best would be all the possible combinations (**2^n combinations)** of compatible and non-compatible
  - Often infeasible!
- Simpler value choices: *one compatible, one incompatible, all compatible or all incompatible, selections of slots*
- It is up to the test case designers

Parameter: Model

del number
- malformed [error]
- not in database [error]
- valid

mber of required slots for selected model (#SMRS)
- 0 [single]
- 1 [property **RSNE**] [single]
- many [property **RSNE**], [property **RSMANY**]

Number of optional slots for selected model (#SMOS)
- 0 [single]
- 1 [property **OSNE**] [single]
- many [property **OSNE**][property **OSMANY**]

Parameter: Components

Correspondence of selection with model slots
- omitted slots [error]
- extra slots [error]
- mismatched slots [error]
- complete correspondence

Number of required components with non-empty selection
- 0 [if **RSNE**] [error]
- < number of required slots [if **RSNE**] [error]
- = number of required slots [if **RSMANY**]

Number of optional components with non-empty selection
- 0
- < number of optional slots [if **OSNE**]
- = number of optional slots [if **OSMANY**]

Required component selection
- some default [single]
- all valid
- ≥ 1 incompatible with slot
- ≥ 1 incompatible with another selection
- ≥ 1 incompatible with model
- ≥ 1 not in database [error]

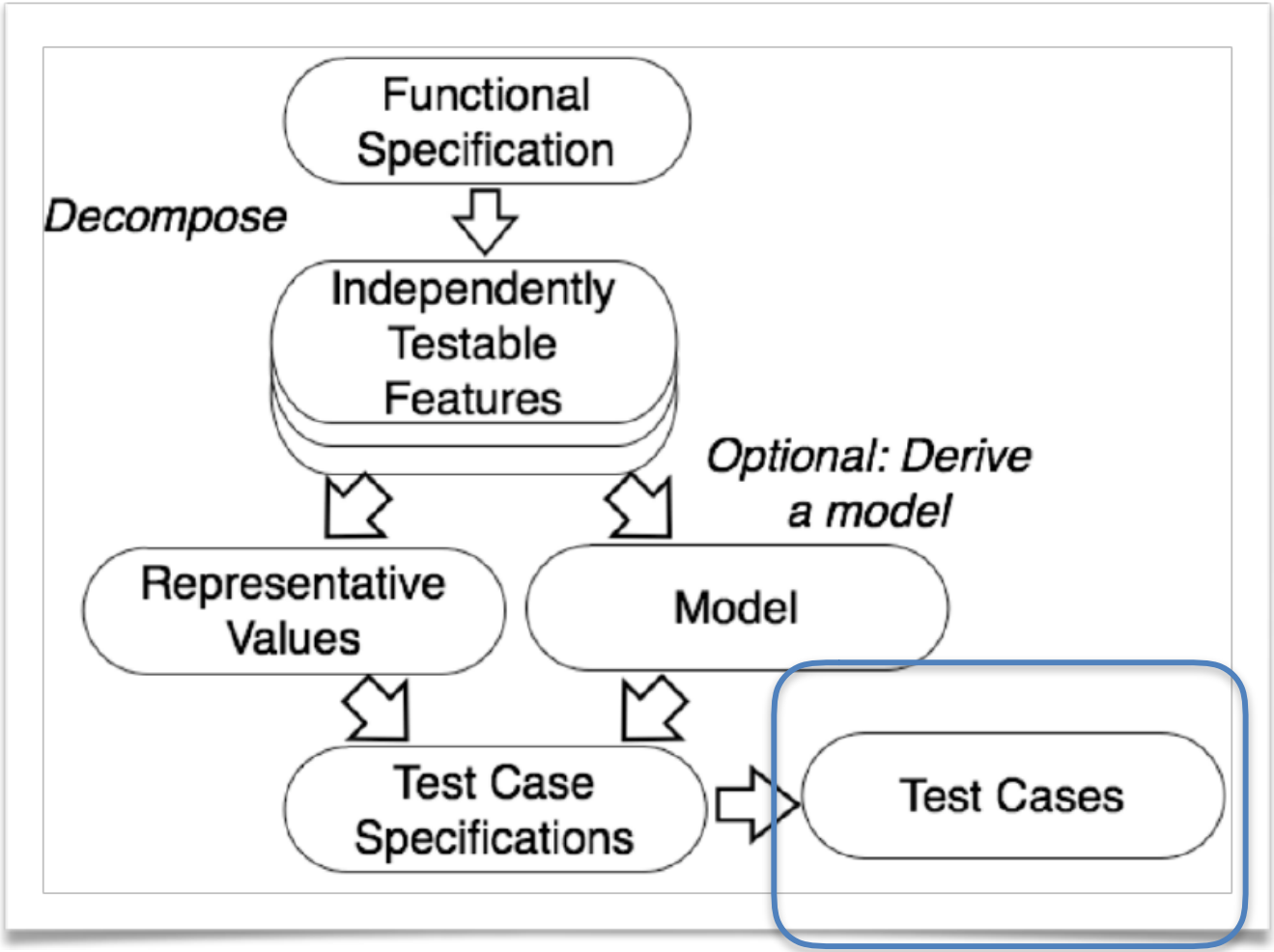
Optional component selection
- some default [single]
- all valid
- ≥ 1 incompatible with slot
- ≥ 1 incompatible with another selection
- ≥ 1 incompatible with model
- ≥ 1 not in database [error]

Environment element: Product database

Number of models in database (#DBM)
- 0 [error]
- 1 [single]
- many

Number of components in database (#DBC)
- 0 [error]
- 1 [single]
- many

Steps in systematic functional testing



Generate test case and instantiate tests

- Turning test case specs into test cases
- Implement test cases by defining the harness to execute them (e.g., FitNess)