Chapter 10.
Functional Testing

Learning Objectives

- Understand the rationale for systematic (non-random) selection of test cases
- Understand why functional test selection is a primary, base-line technique
- Distinguish functional testing from other systematic testing techniques

Functional Testing

- **Functional testing**
  - Deriving test cases from program specifications
  - 'Functional' refers to the source of information used in test case design, not to what is tested.

- Also known as:
  - Specification-based testing (from specifications)
  - Black-box testing (no view of source code)

- Functional specification = description of intended program behavior
  - Formal or informal

Systematic testing vs. Random testing

- **Random (uniform) testing**
  - Pick possible inputs uniformly
  - Avoids designer's bias
  - But, treats all inputs as equally valuable

- **Systematic (non-uniform) testing**
  - 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 a systematic (partition-based) testing strategy.

Why Not Random Testing?

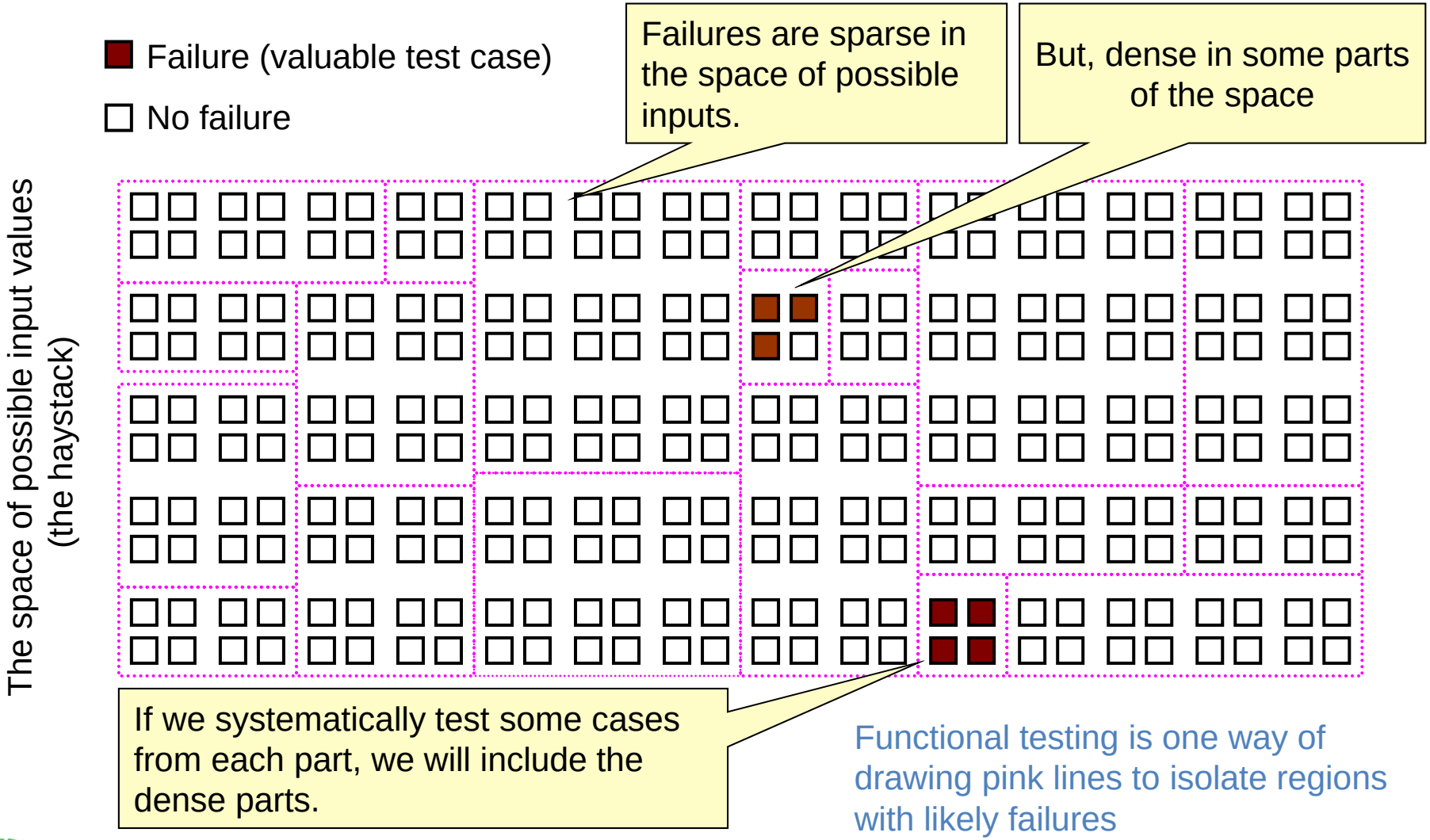
- Due to non-uniform distribution of faults
  - Example:
    - Java class "roots" applies quadratic equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
  - Supposed an incomplete implementation logic:
    - Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a=0$
  - Failing values are sparse in the input space: needles in a very big haystack
  - Random sampling is unlikely to choose a=0 and b=0.

Purpose of Testing

- Our goal is to find needles and remove them from hay.
  → Look systematically (non-uniformly) for needles !!!
  – We need to use everything we know about needles.
    - E.g. Are they heavier than hay? Do they sift to the bottom?

- To estimate the proportion of needles to hay
  → Sample randomly !!!
  – Reliability estimation requires unbiased samples for valid statistics.
  – But that's not our goal.

Systematic Partition Testing



Functional testing is one way of drawing pink lines to isolate regions with likely failures

Principles of Systematic Partitioning

- Exploit some knowledge to choose samples that are more likely to include "special" or "trouble-prone" regions of the input space
  - Failures are sparse in the whole input space.
  - But, we may find regions in which they are dense.
- (Quasi-) Partition testing: separates the input space into classes whose union is the entire space
- Desirable case: Each fault leads to failures that are dense (easy to find) in some class of inputs
  - Sampling each class in the quasi-partition selects at least one input that leads to a failure, revealing the fault.
  - Seldom guaranteed; We depend on experience-based heuristics.

A Systematic Approach: Functional Testing

- Functional testing uses the specification (formal or informal) to partition the input space.
  - E.g. 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. (as in the "roots" program)

- Functional Testing is a base-line technique for designing test cases.

Functional Testing

- The base-line technique for designing test cases
  - Timely
    - Often useful in refining specifications and assessing testability before code is written
  - Effective
    - Find some classes of fault (e.g. missing logic) that can elude other approaches
  - Widely applicable
    - To any description of program behavior serving as specification
    - At any level of granularity from module to system testing
  - Economical
    - Typically less expensive to design and execute than structural (code-based) test cases

Functional Test vs. Structural Test

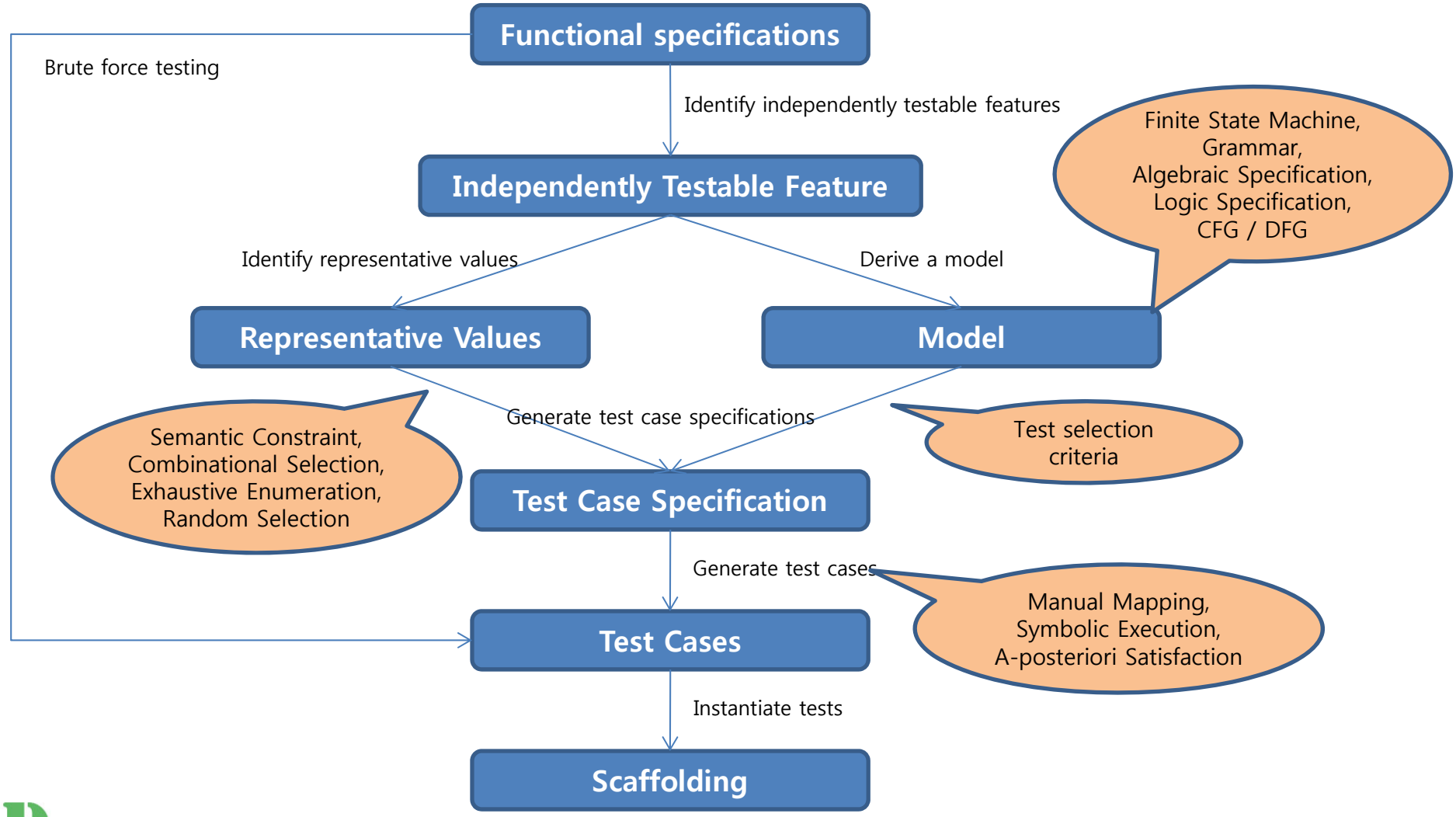
- Different testing strategies are most effective for different classes of faults.

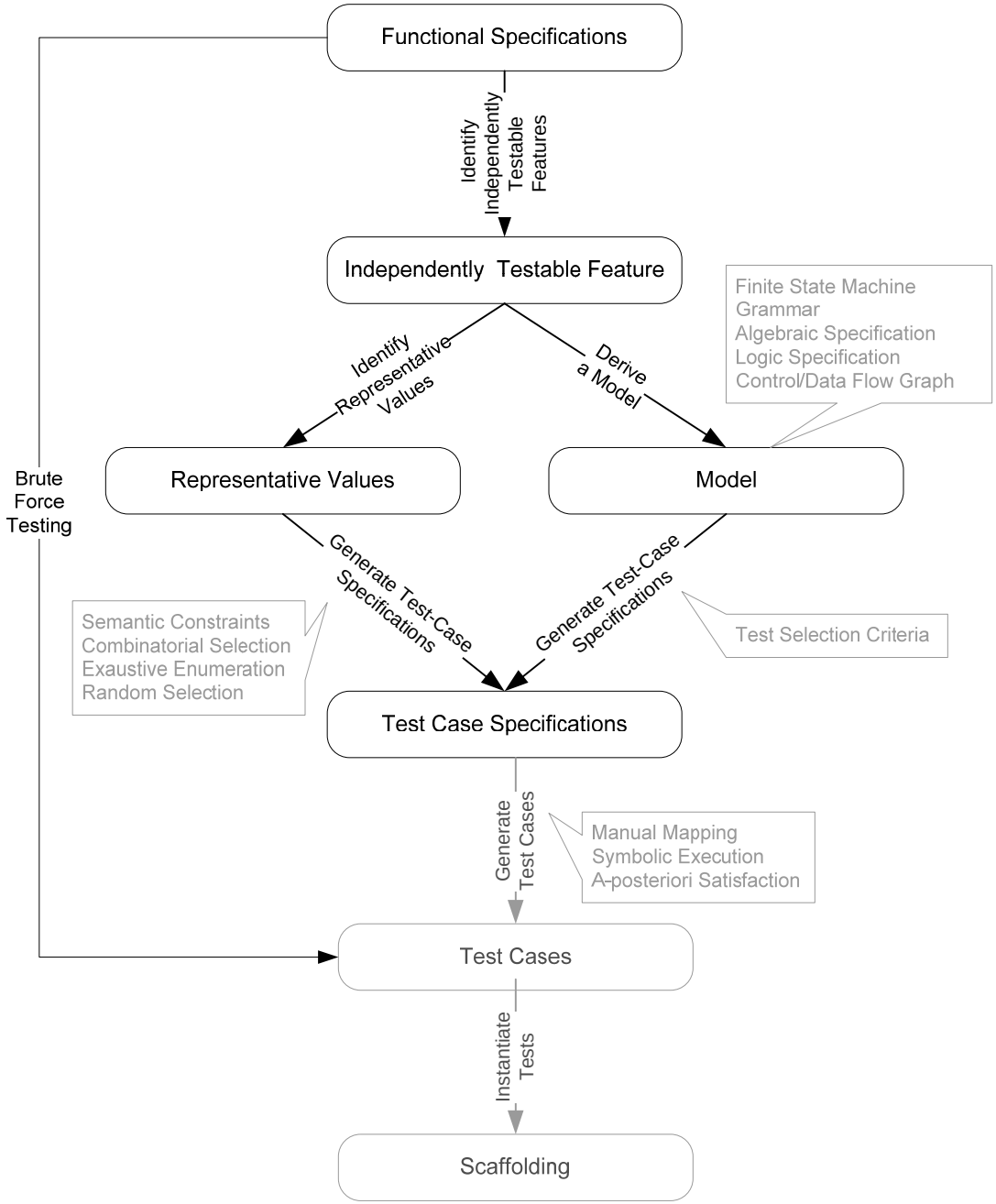
- Functional testing is best for missing logic faults.
  - A common problem: Some program logic was simply forgotten.
  - Structural (code-based) testing will never focus on code that isn't there.

- Functional test applies at all granularity levels
  - Unit (from module interface spec)
  - Integration (from API or subsystem spec)
  - System (from system requirements spec)
  - Regression (from system requirements + bug history)

- Structural test design applies to relatively small parts of a system
  - Unit and integration testing

Main Steps of Functional Program Testing




Functional Specifications
Identify Independently Testable Features
Independently Testable Feature
Identify Representative Values
Derive a Model
Finite State Machine
Grammar
Algebraic Specification
Logic Specification
Control/Data Flow Graph
Brute Force Testing
Representative Values
Model
Generate Test-Case Specifications
Generate Test-Case Specifications
Semantic Constraints
Combinatorial Selection
Exaustive Enumeration
Random Selection
Test Selection Criteria
Test Case Specifications
Generate Test Cases
Manual Mapping
Symbolic Execution
A-posteriori Satisfaction
Test Cases
Instantiate Tests
Scaffolding

From Specifications to Test Cases

1. Identify independently testable features
   - If the specification is large, break it into independently testable features.

2. Identify representative classes of values, or derive a model of behavior
   - Often simple input/output transformations don't describe a system.
   - We use models in program specification, in program design, and in test design too.

3. Generate test case specifications
   - Typically, combinations of input values or model behaviors

4. Generate test cases and instantiate tests

Summary

- Functional testing (generating test cases from specifications) is a valuable and flexible approach to software testing.
  - Applicable from very early system specifications right through module specifications

- Partition testing suggests dividing the input space into equivalent classes.
  - Systematic testing is intentionally non-uniform to address special cases, error conditions and other small places.
  - Dividing a big haystack into small and hopefully uniform piles where the needles might be concentrated

Chapter 11.
Combinatorial Testing

Learning Objectives

- Understand three key ideas in combinatorial approaches
  - Category-partition testing
  - Pairwise testing
  - Catalog-based testing

Overview

- Combinatorial testing identifies distinct attributes that can be varied.
  - In data, environment or configuration
  - Example:
    - Browser could be "IE" or "Firefox"
    - Operating system could be "Vista", "XP" or "OSX"
- Combinatorial testing systematically generates combinations to be tested.
  - Example:
    - IE on Vista, IE on XP, Firefox on Vista, Firefox on OSX, etc.
- Rationale:
  - Test cases should be varied and include possible "corner cases".

Key Ideas in Combinatorial Approaches

1. Category-partition testing
   - Separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases

2. Pairwise testing
   - Systematically test interactions among attributes of the program input space with a relatively small number of test cases

3. Catalog-based testing
   - Aggregate and synthesize the experience of test designers in a particular organization or application domain, to aid in identifying attribute values

1. Category-Partition Testing

1. Decompose the specification into independently testable features
   - for each feature, identify parameters and environment elements
   - for each parameter and environment element, identify elementary characteristics (→ categories)

2. Identify representative values
   - for each characteristic(category), identify classes of values
     - normal values
     - boundary values
     - special values
     - error values

3. Generate test case specifications

An Example: “Check Configuration”

- In the Web site of a computer manufacturer, *i.e. Dell*, ‘checking configuration’ checks the validity of a computer configuration.
  - Two parameters:
    - Model
    - Set of Components

Informal Specification of 'Model'

Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customer's needs.

Example: The required "slots" of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.

Informal Specification of 'Set of Component'

Set of Components: A set of (slot, component) pairs, corresponds to the required and optional slots of the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value empty is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

Example: The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.

Step 1: Identify Independently Testable Features and Parameter Characteristics

- Choosing categories
  - No hard-and-fast rules for choosing categories!
  - Not a trivial task

- Categories reflect test designer's judgment.
  - Which classes of values may be treated differently by an implementation.

- Choosing categories well requires experience and knowledge of the application domain and product architecture.

Identify Independently Testable Units

Model	Model number
	Number of required slots for selected model (#SMRS)
	Number of optional slots for selected model (#SMOS)

Components	Correspondence of selection with model slots
	Number of required components with selection ≠ empty
	Required component selection
	Number of optional components with selection ≠ empty
	Optional component selection

Product Database	Number of models in database (#DBM)
	Number of components in database (#DBC)

Parameters **Categories**

Step 2: Identify Representative Values

- Identify representative classes of values for each of the categories
- Representative values may be identified by applying
  - Boundary value testing
    - Select extreme values within a class
    - Select values outside but as close as possible to the class
    - Select interior (non-extreme) values of the class
  - Erroneous condition testing
    - Select values outside the normal domain of the program

Representative Values: Model

- Model number
  - Malformed
  - Not in database
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0
  - 1
  - Many
- Number of optional slots for selected model (#SMOS)
  - 0
  - 1
  - Many

Representative Values: Components

- Correspondence of selection with model slots
  - Omitted slots
  - Extra slots
  - Mismatched slots
  - Complete correspondence
- Number of required components with non empty selection
  - 0
  - < number required slots
  - = number required slots
- Required component selection
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database

Representative Values: Components

- Number of optional components with non empty selection
  - 0
  - < #SMOS
  - = #SMOS
- Optional component selection
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database

Representative Values: Product Database

- Number of models in database (#DBM)
  - 0
  - 1
  - Many
- Number of components in database (#DBC)
  - 0
  - 1
  - Many
- Note 0 and 1 are unusual (special) values.
  - They might cause unanticipated behavior alone or in combination with particular values of other parameters.

Step 3: Generate Test Case Specifications

- A combination of values for each category corresponds to a test case specification.
  - In the example, we have 314,928 test cases.
  - Most of which are impossible.
  - Example: zero slots and at least one incompatible slot

- Need to introduce constraints in order to
  - Rule out impossible combinations, and
  - Reduce the size of the test suite, if too large

  - Example:
    - Error constraints
    - Property constraints
    - Single constraints

Error Constraints

- [error] indicates a value class that corresponds to an erroneous values.
  - Need to be tried only once
- Error value class
  - No need to test all possible combinations of errors, and one test is enough.

Model number
- Malformed [error]
- Not in database [error]
- Valid

Correspondence of selection with model slots
- Omitted slots [error]
- Extra slots [error]
- Mismatched slots [error]
- Complete correspondence

Number of required comp. with non empty selection
- 0 [error]
- < number of required slots [error]

Required comp. selection
- ≥ 1 not in database [error]

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

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

Error constraints reduce test suite from 314,928 to 2,711 test cases

Property Constraints

- Constraint [property] [if-property] rule out invalid combinations of values.
  - [property] groups values of a single parameter to identify subsets of values with common properties.
  - [if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category.

Property Constraints

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] [property RSMANY]

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

Number of optional comp. with non empty selection

< number required slots	[if OSNE]
= number required slots	[if OSMANY]

from 2,711 to 908 test cases

Single Constraints

- [single] indicates a value class that test designers choose to test only once to reduce the number of test cases.

- Example
  - Value some default for required component selection and optional component selection may be tested only once despite not being an erroneous condition.

- Note
  - Single and error have the same effect but differ in rationale.
  - Keeping them distinct is important for documentation and regression testing.

Single Constraints

Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[single] [property OSNE]

Required component selection

Some default	[single]

Optional component selection

Some default	[single]

Number of models in database (#DBM)

1	[single]

Number of components in database (#DBC)

1	[single]

from 908 to 69 test cases

Check Configuration – Summary of Categories

Parameter Model

- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number 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]

Environment Product data base

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

Parameter Component

- Correspondence of selection with model slots
  - Omitted slots [error]
  - Extra slots [error]
  - Mismatched slots [error]
  - Complete correspondence
- # of required components (selection ≠ empty)
  - 0 [if RSNE] [error]
  - < number required slots [if RSNE] [error]
  - = number required slots [if RSMANY]
- Required component selection
  - Some defaults [single]
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database [error]
- # of optional components (selection ≠ empty)
  - 0
  - < #SMOS [if OSNE]
  - = #SMOS [if OSMANY]
- Optional component selection
  - Some defaults [single]
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database [error]

Category-Partitioning Testing, in Summary

- Category partition testing gives us systematic approaches to
  - Identify characteristics and values (the creative step)
  - Generate combinations (the mechanical step)

- But, test suite size grows very rapidly with number of categories.
- Pairwise (and n-way) combinatorial testing is a non-exhaustive approach.
  - Combine values systematically but not exhaustively

2. Pairwise Combination Testing

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases.
  - Without many constraints, the number of combinations may be unmanageable.

- Pairwise combination
  - Instead of exhaustive combinations
  - Generate combinations that efficiently cover all pairs (triples,…) of classes
  - Rationale:
    - Most failures are triggered by single values or combinations of a few values.
    - Covering pairs (triples,…) reduces the number of test cases, but reveals most faults.

An Example: Display Control

- No constraints reduce the total number of combinations 432 (3x4x3x4x3) test cases, if we consider all combinations.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise Combination: 17 Test Cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Adding Constraints

- Simple constraints
  - Example: "Color monochrome not compatible with screen laptop and full size" can be handled by considering the case in separate tables.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	
limited-bandwidth	Spanish	Document-loaded	16-bit	
	Portuguese		True-color	

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal		
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise Combination Testing, in Summary

- Category-partition approach gives us
  - Separation between (manual) identification of parameter characteristics and values, and (automatic) generation of test cases that combine them
  - Constraints to reduce the number of combinations
- Pairwise (or n-way) testing gives us
  - Much smaller test suites, even without constraints
  - But, we can still use constraints.
- We still need help to make the manual step more systematic.

3. Catalog-based Testing

- Deriving value classes requires human judgment.
- Therefore, gathering experience in a systematic collection can
  - Speed up the test design process
  - Routinize many decisions, better focusing human effort
  - Accelerate training and reduce human error

- Catalogs capture the experience of test designers by listing important cases for each possible type of variable.
  - Example: If the computation uses an integer variable, a catalog might indicate the following relevant cases
    - The element immediately preceding the lower bound
    - The lower bound of the interval
    - A non-boundary element within the interval
    - The upper bound of the interval
    - The element immediately following the upper bound

Catalog-based Testing Process

1. Identify elementary items of the specification
   - Pre-conditions
   - Post-conditions
   - Definitions
   - Variables
   - Operations

2. Derive a first set of test case specifications from pre-conditions, post-conditions and definitions

3. Complete the set of test case specifications using test catalogs

An Example: ‘cgi_decode’

- An informal specification of ‘cgi_decode’

Function cgi_decode translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) of most web servers.

CGI translates spaces to +, and translates most other non-alphanumeric characters to hexadecimal escape sequences.

cgi_decode maps + to spaces, %xy (where x and y are hexadecimal digits) to the corresponding ASCII character, and other alphanumeric characters to themselves.

‘cgi_digicode’ Input/Output

- **[INPUT]: encoded** A string of characters (the input CGI sequence) containing below and terminated by a null character
  - alphanumeric characters
  - the character +
  - the substring “%xy” , where x and y are hexadecimal digits

- **[OUTPUT]: decoded** A string of characters (the plain ASCII characters corresponding to the input CGI sequence)
  - alphanumeric characters copied into output (in corresponding positions)
  - blank for each ‘+’ character in the input
  - single ASCII character with value xy for each substring “%xy”

- **[OUTPUT]: return value** cgi_decode returns
  - 0 for success
  - 1 if the input is malformed

‘cgi_digicode’ Definitions

- Pre-conditions: Conditions on inputs that must be true before the execution
  - Validated preconditions: checked by the system
  - Assumed preconditions: assumed by the system
- Post-conditions: Results of the execution
- Variables: Elements used for the computation
- Operations: Main operations on variables and inputs
- Definitions: Abbreviations

Step 1: Identify Elementary Items of the Specification

VAR 1 encoded: a string of ASCII characters
VAR 2 decoded: a string of ASCII characters
VAR 3 return value: a boolean

DEF 1 hexadecimal characters, in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']
DEF 2 sequences %xy, where x and y are hexadecimal characters
DEF 3 CGI items as alphanumeric character, or '+', or CGI hexadecimal

OP 1 Scan the input string encoded

PRE 1 (Assumed) input string encoded null-terminated string of chars
PRE 2 (Validated) input string encoded sequence of CGI items

POST 1 if encoded contains alphanumeric characters, they are copied to the output string
POST 2 if encoded contains characters +, they are replaced in the output string by ASCII SPACE characters
POST 3 if encoded contains CGI hexadecimals, they are replaced by the corresponding ASCII characters
POST 4 if encoded is processed correctly, it returns 0
POST 5 if encoded contains a wrong CGI hexadecimal (a substring xy, where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1
POST 6 if encoded contains any illegal character, it returns 1VAR 1 encoded: a string of ASCII characters

Step 2: Derive an Initial Set of Test Case Specifications

- Validated preconditions:
  - Simple precondition (expression without operators)
    - 2 classes of inputs:
      - inputs that satisfy the precondition
      - inputs that do not satisfy the precondition
  - Compound precondition (with AND or OR):
    - apply modified condition/decision (MC/DC) criterion

- Assumed precondition:
  - apply MC/DC only to "OR preconditions"

- Postconditions and Definitions:
  - if given as conditional expressions, consider conditions as if they were validated preconditions

Test Cases from PRE

PRE 2 (Validated): the input string encoded is a sequence of CGI items

- TC-PRE2-1: encoded is a sequence of CGI items
- TC-PRE2-2: encoded is not a sequence of CGI items

POST 1: if encoded contains alphanumeric characters, they are copied in the output string in the corresponding position

- TC-POST1-1: encoded contains alphanumeric characters
- TC-POST1-2: encoded does not contain alphanumeric characters

POST 2: if encoded contains characters +, they are replaced in the output string by ASCII SPACE characters

- TC-POST2-1: encoded contains character +
- TC-POST2-2: encoded does not contain character +

Test Cases from POST

POST 3: if encoded contains CGI hexadecimals, they are replaced by the corresponding ASCII characters

- TC-POST3-1 Encoded: contains CGI hexadecimals
- TC-POST3-2 Encoded: does not contain a CGI hexadecimal

POST 4: if encoded is processed correctly, it returns 0

POST 5: if encoded contains a wrong CGI hexadecimal (a substring xy, where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1

- TC-POST5-1 Encoded: contains erroneous CGI hexadecimals

POST 6 if encoded contains any illegal character, it returns 1

- TC-POST6-1 Encoded: contains illegal characters

Step 3: Complete the Test Case Specification using Catalog

- Scan the catalog sequentially
  - For each element of the catalog,
    - Scan the specifications and apply the catalog entry
- Delete redundant test cases

- Catalog
  - List of kinds of elements that can occur in a specification
  - Each catalog entry is associated with a list of generic test case specifications.

- Example: Catalog entry Boolean
  - Two test case specifications: true, false
  - Label in/out indicate if applicable only to input, output, both

A Simple Test Catalog

- Boolean
  - True [in/out]
  - False [in/out]
- Enumeration
  - Each enumerated value [in/out]
  - Some value outside the enumerated set [in]
- Range L ... U
  - L-1 [in]
  - L [in/out]
  - A value between L and U [in/out]
  - U [in/out]
  - U+1 in]
- Numeric Constant C
  - C [in/out]
  - C –1 [in]
  - C+1 [in]
  - Any other constant compatible with C [in]
- Non-Numeric Constant C
  - C [in/out]
  - Any other constant compatible with C [in]
  - Some other compatible value [in]
- Sequence
  - Empty [in/out]
  - A single element [in/out]
  - More than one element [in/out]
  - Maximum length (if bounded) or very long [in/out]
  - Longer than maximum length (if bounded) [in]
  - Incorrectly terminated [in]
- Scan with action on elements P
  - P occurs at beginning of sequence [in]
  - P occurs in interior of sequence [in]
  - P occurs at end of sequence [in]
  - PP occurs contiguously [in]
  - P does not occur in sequence [in]
  - pP where p is a proper prefix of P [in]
  - Proper prefix p occurs at end of sequence [in]

Catalog Entry: Boolean

- Boolean
  - True [in/out]
  - False [in/out]

- Application to return value generates 2 test cases already covered by TC-PRE2-1 and TC-PRE2-2.

Catalog Entry: Enumeration

- Enumeration
  - Each enumerated value [in/out]
  - Some value outside the enumerated set [in]

- Applications to CGI item (DEF 3)
  - included in TC-POST1-1, TC-POST1-2, TC-POST2-1, TC-POST2-2, TC-POST3-1, TC-POST3-2

- Applications to improper CGI hexadecimals
  - New test case specifications
    - TC-POST5-2 encoded terminated with "%x", where x is a hexadecimal digit
    - TC-POST5-3 encoded contains "%ky", where k is not a hexadecimal digit and y is a hexadecimal digit
    - TC-POST5-4 encoded contains "%xk", where x is a hexadecimal digit and k is not
  - Old test case specifications can be eliminated if they are less specific than the newly generated cases.
    - TC-POST3-1 encoded contains CGI hexadecimals
    - TC-POST5-1 encoded contains erroneous CGI hexadecimals

Catalog Entries: the Others

We can apply in the same ways.

- range
- numeric constant
- non-numeric constant
- sequence
- scan

Summary of Generated Test Cases

TC-POST2-1: *encoded* contains *+*
TC-POST2-2: *encoded* does not contain *+*
TC-POST3-2: *encoded* does not contain a CGI-hexadecimal
TC-POST5-2: *encoded* terminated with *%x*
TC-VAR1-1: *encoded* is the empty sequence
TC-VAR1-2: *encoded* a sequence containing a single character
TC-VAR1-3: *encoded* is a very long sequence
TC-DEF2-1: *encoded* contains *%/y*
TC-DEF2-2: *encoded* contains *%0y*
TC-DEF2-3: *encoded* contains '*%xy*' (x in [1..8])
TC-DEF2-4: *encoded* contains '*%9y*'
TC-DEF2-5: *encoded* contains '*%:y*'
TC-DEF2-6: *encoded* contains '*%@y*'
TC-DEF2-7: *encoded* contains '*%Ay*'
TC-DEF2-8: *encoded* contains *'%xy'* (x in [B..E])
TC-DEF2-9: *encoded* contains '*%Fy*'
TC-DEF2-10: *encoded* contains '*%Gy*'
TC-DEF2-11: *encoded* contains *%`y'*
TC-DEF2-12: *encoded* contains *%ay*
TC-DEF2-13: *encoded* contains *%xy* (*x* in [*b..e*])
TC-DEF2-14: *encoded* contains *%fy'*
TC-DEF2-15: *encoded* contains *%gy*
TC-DEF2-16: *encoded* contains *%x/*
TC-DEF2-17: *encoded* contains *%x0*
TC-DEF2-18: *encoded* contains *%xy* (*y* in [*1..8*])
TC-DEF2-19: *encoded* contains *%x9*
TC-DEF2-20: *encoded* contains *%x:*
TC-DEF2-21: *encoded* contains *%x@*
TC-DEF2-22: *encoded* contains *%xA*
TC-DEF2-23: *encoded* contains *%xy*(*y* in [*B..E*])
TC-DEF2-24: *encoded* contains *%xF*
TC-DEF2-25: *encoded* contains *%xG*
TC-DEF2-26: *encoded* contains *%x`*
TC-DEF2-27: *encoded* contains *%xa*
TC-DEF2-28: *encoded* contains *%xy* (*y* in [*b..e*])
TC-DEF2-29: *encoded* contains *%xf*
TC-DEF2-30: *encoded* contains *%xg*
TC-DEF2-31: *encoded* terminates with *%*
TC-DEF2-32: *encoded* contains *%xyz*
TC-DEF3-1: *encoded* contains */*
TC-DEF3-2: *encoded* contains *0*
TC-DEF3-3: *encoded* contains *c* in [*1..8*]
TC-DEF3-4: *encoded* contains *9*
TC-DEF3-5: *encoded* contains *:*
TC-DEF3-6: *encoded* contains *@*
TC-DEF3-7: *encoded* contains *A*
TC-DEF3-8: *encoded* contains *c* in*[B..Y]*
TC-DEF3-9: *encoded* contains *Z*
TC-DEF3-10: *encoded* contains *[*
TC-DEF3-11: *encoded* contains `
TC-DEF3-12: *encoded* contains *a*
TC-DEF3-13: *encoded* contains *c* in [*b..y*]
TC-DEF3-14: *encoded* contains *z*
TC-DEF3-15: *encoded* contains {
TC-OP1-1: *encoded* starts with an alphanumeric character
TC-OP1-2: *encoded* starts with *+*
TC-OP1-3: *encoded* starts with *%xy*
TC-OP1-4: *encoded* terminates with an alphanumeric character
TC-OP1-5: *encoded* terminates with *+*
TC-OP1-6: *encoded* terminated with *%xy*
TC-OP1-7: *encoded* contains two consecutive alphanumeric characters
TC-OP1-8: *encoded* contains *++*
TC-OP1-9: *encoded* contains *%xy%zw*
TC-OP1-10: *encoded* contains *%x%yz*

What Have We Got from Three Methods?

- From category partition testing:
  - Division into a (manual) step of identifying categories and values, with constraints, and an (automated) step of generating combinations
- From catalog-based testing:
  - Improving the manual step by recording and using standard patterns for identifying significant values
- From pairwise testing:
  - Systematic generation of smaller test suites
- Three ideas can be combined.

Summary

- Requirements specifications typically begin in the form of natural language statements.
  - But, flexibility and expressiveness of natural language is an obstacle to automatic analysis.
- Combinatorial approaches to functional testing consist of
  - A manual step of structuring specifications into set of properties
  - An automatic(-able) step of producing combinations of choices
- Brute force synthesis of test cases is tedious and error prone.
  - Combinatorial approaches decompose brute force work into steps to attack the problem incrementally by separating analysis and synthesis activities that can be quantified and monitored, and partially supported by tools.