Spec-Based Testing

Process and Input Space Design

From Attributes to Characteristics to Tests:

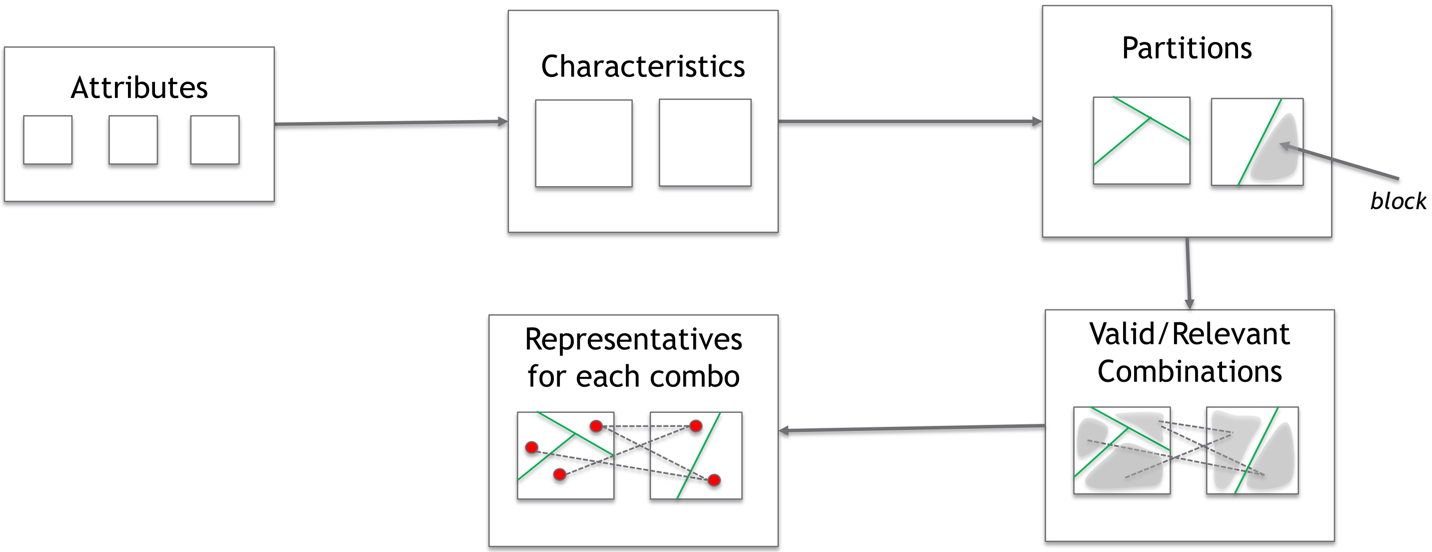
Designing Partitions and

applying Combinatorial Strategies

to create test cases

Input space of a program is often too large!

We have to sample it intelligently!



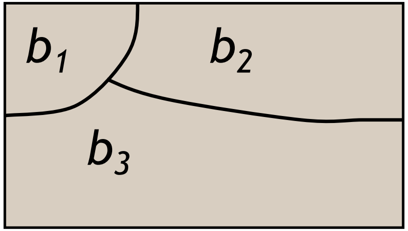
Partitions should be disjoint and complete

- *Characteristic Q*
- *Partition q* of *Q*
- The **partition** *q* defines a *set of n blocks*:

$$B_q = \{ b_1, b_2, \ldots, b_n \}$$

- *q ideally* satisfies two properties :

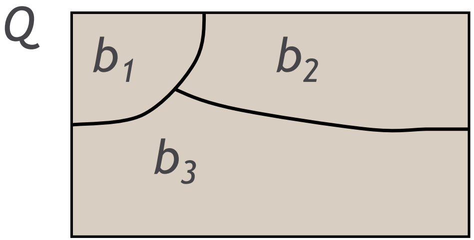
1. blocks are *disjoint* (no overlap)
2. blocks *cover* all of *Q* (complete)



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

$$\bigcup_{b \in B_n} b = Q$$

Partitions should be uniform



Hard to check or guarantee!

We want blocks to be as *uniform* as possible:

If every pair of values *v*, *w* in block *b*, *v* and *w* can be "treated as equivalents", then *b* is an equivalence class

- If value *v* (*w*) makes a test case fail, so does *w* (*v*)
- If value *v* (*w*) makes a test case pass, so does *w* (*v*)
- Test cases exercising both *v* and *w* in exactly the same combinations are redundant

Characteristics can be…

Syntactic (*mechanical*)

or

Semantic (*intelligent*)

Characteristics can be…

Syntactic

e.g., “sign of attribute *a*” -- inferred from type of an attribute and common/standard categories/states for that type; depends on a single attribute

or

Semantic

e.g., “nature of output” -- conceptual, requires domain knowledge, depends on multiple attributes

Syntactic partitions lead to straightforward test cases, but may not be very effective

In general: attributes directly give rise to syntactic characteristics; mechanistic, easy to design or automate; but less efficient & effective (more tests, not necessarily good, meaningful tests)

Example of characteristics

Syntactic: Directly inferred from attributes, their types or natural properties/states/categories and boundary values of the types

- Object X is null (true, false)
- Unsigned integer range (0, positive but not maxint, maxint)
- Order of array (sorted, inverse-sorted, arbitrary, …)
- Type of input device (DVD, CD, Stream, Computer, …)

Semantic: Indirectly inferred from attributes (possibly a property of outputs or category of execution side effect)

- Recognized triangle type (equilateral, scalene, isosceles, not-a-triangle)
- Overdraft status after transaction (true, false)
- Number of cycles in resulting graph (0, one, many)
- Separation between two aircrafts (too small, acceptable)

Choosing partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong (remember *disjoint* and *complete*)
- Consider the simple characteristic "*order of list L*"

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

What if the list is of length 1?

Choosing partitions: tricky case

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the "*order of list L*" (syntactic)

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

Need another characteristic?

What if the list is of length 1?

The list will be in all three blocks …

That is, *disjointness* is violated …

Ambiguous!

Choosing partitions: solution

- Characteristic 1: "*multiplicity of list L*" (syntactic)

$b_{1,1}$ = list is empty
$b_{1,2}$ = list has 1 element
$b_{1,3}$ = list has more than 1 element

- Characteristic 2: "*order of list L*" (syntactic)

$b_{2,1}$ = sorted in ascending order (if $b_{1,3}$)
$b_{2,2}$ = sorted in descending order (if $b_{1,3}$)
$b_{2,3}$ = arbitrary order (if $b_{1,3}$)
$b_{2,4}$ = ambiguous order (if *not* $b_{1,3}$)

Characteristic 2 is *constrained* by Characteristic 1

Sanity checking partitions is important

- If the partitions are not *complete* or *disjoint*, that means the partitions may not have not been considered carefully enough

- They should be reviewed carefully, like any *design* attempt – "*Are we missing a characteristic?*"

- Different characteristics and partition *alternatives* can be considered, and the most optimal chosen

Let's review: Test cases are generated using a defined process in systematic spec-based testing

We start with an independently testable requirement

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes and spec

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

Test Case Specs

Representatives for each block + oracles

Test Cases

Let's review: Test cases are generated using a defined process in systematic spec-based testing

Input Space Modeling

- The input space is scoped by the attributes, but...
- Its structure is defined in terms of characteristics
- Each characteristic is partitioned into sets of blocks
- *This is the most creative part*

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes and spec

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

Test Case Specs

Representatives for each block + oracles

Test Cases

Let's review: Test cases are generated using a defined process in systematic spec-based testing

Identify all attributes for requirement

- Not too difficult, but difficult to be comprehensive and get it right

1. Independently testable requirement
2. **Input space** of requirement: **attributes**
3. **Characteristics** derived from attributes and spec
4. **Partition** (set of blocks) for each characteristic
5. Valid/interesting **combinations** of blocks
6. Test Case Specs
7. **Representatives** for each block + oracles
8. Test Cases

Test cases are generated using a systematic process

Requires domain knowledge, type knowledge

Look for sources of characteristics in the spec

- Pre-conditions (properties that inputs and initial state must satisfy)
- Post-conditions (properties that outputs and end state or side effects must satisfy)
- Implicit relationships among attributes
- Relationship of attributes with special or boundary values (zero, null, blank, …) relevant to domain
- Definitions in spec

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes and spec

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

Test Case Specs

Representatives for each block + oracles

Test Cases

Test cases are generated using a systematic process

- The partitioning often flows directly from the definition of characteristics
- More blocks mean more tests
- Strategies/heuristics for creating blocks
  - Include valid, invalid, and special values
  - Explore boundaries of input domains
  - Include values that represent "normal use"
  - Check for completeness and disjointness
  - Are they likely to be uniform?

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes and spec

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

Test Case Specs

Representatives for each block + oracles

Test Cases

Test cases are generated using a systematic process

Better to have more characteristics with fewer blocks than fewer characteristics with more blocks

– *results in fewer mistakes and fewer tests*

- Independently testable requirement
- Input space of requirement: attributes
- Characteristics derived from attributes and spec
- Partition (set of blocks) for each characteristic
- Valid/interesting combinations of blocks
- Test Case Specs
- Representatives for each block + oracles
- Test Cases

Test cases are generated using a systematic process

Input Space Model done! →

- Independently testable requirement
- Input space of requirement: attributes
- Characteristics derived from attributes
- Partition (set of blocks) for each characteristic
- Valid/interesting combinations of blocks
- Test Case Specs
- Representatives for each block + oracles
- Test Cases

Test cases are generated using a systematic process

Apply a combinatorial strategy to choose combinations of blocks from the partitions

- Choosing all combinations is usually infeasible
- Which combinations to consider?
- Input space model coverage criteria allow proper/optimal combinations to be chosen
- Some combinations maybe invalid or impossible

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

Test Case Specs

Representatives for each block + oracles

Test Cases

Test cases are generated using a systematic process

Each combination is a test case spec

- Independently testable requirement
- Input space of requirement: attributes
- Characteristics derived from attributes
- Partition (set of blocks) for each characteristic
- Valid/interesting combinations of blocks
- Test Case Specs
- Representatives for each block + oracles
- Test Cases

Test cases are generated using a systematic process

Choose representatives from each block for each combination

- Representative is determined by choosing values for underlying attributes
- Vary the representatives from the same block from combination to combination
  - *remember example from mutation testing*

Add the oracle

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

Test Case Specs

Representatives for each block + oracles

Test Cases

Test cases are generated using a systematic process

- Convert to code
- Add the fixture

Get actual test cases that can be executed!

Independently testable requirement

Input space of requirement: attributes

Characteristics derived from attributes

Partition (set of blocks) for each characteristic

Valid/interesting combinations of blocks

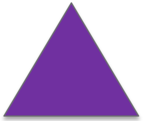
Test Case Specs

Representatives for each block

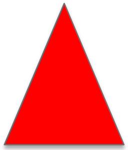
Test Cases

Example: TriTyp

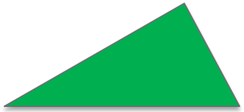
- Identify the type of an integer triangle given the length of three sides
- Single testable function
- Inputs: 3 integers representing length of the three sides
  - a, b, c
- Output: triangle type



equilateral – *all sides are equal*



isosceles – *two sides are equal*



scalene – *all sides are unequal*

TriTyp: naïve syntactic approach

3 attributes corresponding to each input parameter

First characterization of TriTyp's inputs

	Block		
Characteristic	b_1	b_2	b_3
q_1 = "Relation of *a* to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of *b* to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of *c* to 0"	greater than 0	equal to 0	less than 0

- 3*3*3 = 27 tests if one representative value is chosen from each block
- Refining the characteristic will lead to more tests …
- But how effective will this be?

Wait a second!

	Block		
Characteristic	b_1	b_2	b_3
q_1 = "Relation of *a* to 0"	2	0	-1

chosen representatives

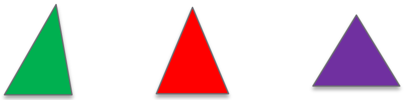
- What if a = -1 or a = 0?
- We just realized: some triangle specs could automatically be *invalid, and there isn't much of a difference between -1 and 0*
  - *Also: I know a bit of geometry, triangle validity is not just about positive sides*
  - *Also: I haven't considered the type of the triangle, the actual property that the program is is supposed to detect!*
- We hadn't thought about any of that!

TriTyp: semantic approach

A semantic-level characterization could use the fact that the three integers represent sides of a triangle, not just integers, and some triangle specs are invalid

<u>Geometric</u> characterization of TriTyp according to output properties

Characteristic	b_1	b_2	b_3	b_4
q_0 = "Geometric Classification"	scalene	isosceles	equilateral	invalid



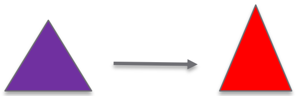
TriTyp: semantic approach (cont'd)

Geometric characterization of TriTyp

Characteristic	b_1	b_2	b_3	b_4
q_0 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

But… something's wrong … equilateral could also be isosceles!

We need to refine the example to make partition disjoint…



Correct geometric characterization of TriTyp

Characteristic	b_1	b_2	b_3	b_4
q_0 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

TriTyp: semantic approach (cont'd)

Representatives for this partition can be chosen as…

Possible attribute values for semantic partition

Characteristic	***b_1*** scalene	***b_2*** isosceles, not equilateral	***b_3*** equilateral	***b_4*** invalid
q_0 = "Geometric Classification"	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

How easy is it to choose the input values for *a*, *b*, *c* for each block?

TriTyp: we can combine syntactic and semantic characteristics to handle validity better

	Block			
Characteristic	b_1	b_2	b_3	b_4
q_0 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid
q_1 = "Positivity of a"	non-positive	positive		
q_2 = "Positivity of b"	non-positive	positive		
q_3 = "Positivity of c"	non-positive	positive		

Which combinations of blocks make sense?
How many test cases total?
How many deal with "unhappy" paths?

TriTyp: we can combine syntactic and semantic characteristics to handle validity better

	Block			
Characteristic	b_1	b_2	b_3	b_4
q_0 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid
q_1 = "Positivity of a"	non-positive	positive		
q_2 = "Positivity of b"	non-positive	positive		
q_3 = "Positivity of c"	non-positive	positive		

Which combinations of blocks make sense? b_1, b_2, b_3 only with all positive

How many test cases total? 3 + 8 = 11

How many deal with "unhappy" paths? 8
- 2^3

TriTyp: there are many ways positive-valued triangle specs can be invalid

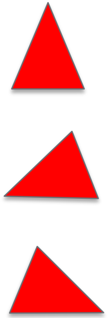
	Block			
Characteristic	***b_1***	***b_2***	***b_3***	***b_4***
q_0 = “Geometric Classification” *semantic*	scalene	isosceles, not equilateral	equilateral	invalid
q_1 = “Positivity of a”	non-positive	positive		
q_2 = “Positivity of b”	non-positive	positive		
q_3 = “Positivity of c”	non-positive	positive		
q_4 = “Validity of Positive Inputs” *semantic*	all positive, invalid: a + b <= c	all positive, invalid: b + c <= a	all positive, invalid: a + c <= b	valid *or* invalid due to negative input: “other”

How many test cases now?

- 11-1+3

TriTyp: what about different ways a triangle can be isosceles?

	Block			
Characteristic	***b_1***	***b_2***	***b_3***	***b_4***
q_0 = "Geometric Classification" *semantic*	scalene	isosceles, not equilateral	equilateral	invalid
q_1 = "Positivity of a"	non-positive	positive		
q_2 = "Positivity of b"	non-positive	positive		
q_3 = "Positivity of c"	non-positive	positive		
q_4 = "Validity of Positive Inputs" *semantic*	all positive, invalid: a + b <= c	all positive, invalid: b + c <= a	all positive, invalid: a + c <= b	valid *or* invalid due to negative input: "other"



Some programs may have dozens of attributes: this can get complicated!

Divide and conquer large requirements to avoid large input space models

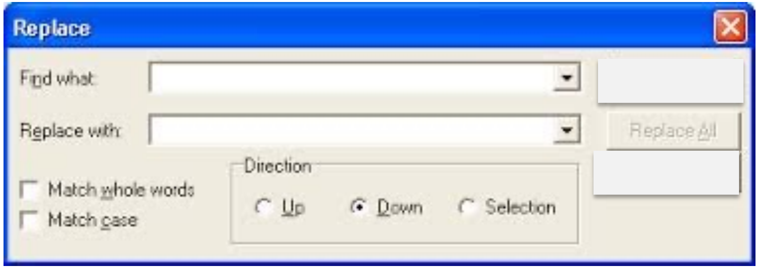
- Divide requirement into smaller requirements each scoped by a subset of attributes
- Create several small input space models
  - Input space models may overlap – difficult to avoid

Systematic spec-based testing summary

- Based only on the *input space* of the program, not the implementation
- Fairly straightforward *once input space is designed*, even with no automation
- Applicable to *all lev*els of testing
- Can be very effective *with good test design*
- Widely used in practice

Another example: Test me!

- What inputs lend well to partition testing?
- What are the characteristics?
- What are the partitions?
    - 5 Attributes
    -



Courtesy of Jeff Gennari

A final (simple) example

- Consider a program that, given a latitude and longitude, computes and returns the country in which that point is located:
  - `String locate(double longitude, double latitude);`
- What is the input domain? What are the partitions?

Think about on your own!
Post your solution to Piazza.

Courtesy of Jeff Gennari