Specification-based and boundary testing

Maurício F. Aniche
M.FinavaroAniche@tudelft.nl
We need more **systematic** and **rigorous** ways to test software!
SPECIFICATION

- Requirements
- Models

STRUCTURAL

- Structure
  (e.g., source code)
SPECIFICATION

Requirements
Models

STRUCTURAL

Structure
(e.g., source code)
A package should store a total number of kilos. There are small bars (1 kilo each) and big bars (5 kilos each). We should calculate the number of small bars to use, assuming we always use big bars before small bars. Return -1 if it can't be done.

*Input*: small bars, big bars, total.

Inspired by https://codingbat.com/prob/p191363
Examples

<table>
<thead>
<tr>
<th>Small</th>
<th>Big</th>
<th>Total</th>
<th>Small bars to use (output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
Partitions based on the requirements

A package should store a **total number of kilos**. There are **small bars** (1 kilo each) and **big bars** (5 kilos each). We should calculate the number of small bars to use, assuming **we always use big bars before small bars**. Return -1 if it can't be done.

- Identify representative classes
  - Only small bars
  - Only big bars
  - Small + big bars
  - Not enough bars
  - Not from the specs: invalid number
- Choose representative values
- Exploit the knowledge to identify trouble-prone regions of the input space.
1) The total is higher than the amount of small and big bars.
Ex: small = 1, big = 1, total = 10

2) Only big bars.
Ex: small = 5, big = 3, total = 10

3) Need for big and small bars.
Ex: small = 5, big = 3, total = 17

4) Only small bars.
Ex: small = 4, big = 2, total = 3

5) Invalid input.
Ex: small = 4, big = 2, total = -1
The category-partition method
(in a nutshell)

• Identify the parameters
• The characteristics of each parameter
  • From the specs
  • Not from the specs
• Add constraints (minimize)
  • Remove invalid combinations
• Reduce number of exceptional behaviors
• Generate combinations
We offer a discount during Christmas. If it’s Christmas, we give a 15% discount in the total amount of the order. If it’s not Christmas, no discount.
We offer a discount during Christmas. If it’s Christmas, we give a 15% discount in the total amount of the order. If it’s not Christmas, no discount.
Category Partition

Two important variables:

• The current date
• The raw amount

The current date

• Christmas
• Not Christmas

The raw amount

• Positive number
• Zero
• Negative number
Constraints

Two important variables:

- The current date
- The raw amount

The current date

- Christmas
- Not Christmas

The raw amount

- Positive number
- Zero
- Negative number [exceptional]
Combinations / Tests

• Christmas
  • Positive number
  • Zero
  • Negative number
• Not Christmas
  • Positive number
  • Zero
Partitions are representative classes of our program, and they guide me throughout the testing phase.
Partitions

2) Only big bars.

Ex: small = 5, big = 3, total = 10
Ex: small = 5, big = 4, total = 15
Ex: small = 5, big = 5, total = 20
Ex: small = 5, big = 6, total = 25
...

Which one should I pick? All?
Equivalent partitions

• If the case is really representative and independent, any instance should do.
  • ISTQB definition: “A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification:”.

• We should try to reduce the human cost.
  • Having lots of (repeated) tests increase the cost.
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small <= total)
        return -1;
    return total;
}
Your tests were not enough!

If I provide

- small = 2
- big = 3
- total = 17

It returns -1,
but it should be 2!
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small <= total)
        return -1;
    return total;
}
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small <= total) {  // Highlighted line
        return -1;
    }
    return total;
}
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small < total)
        return -1;
    return total;
}
1) The total is higher than the amount of small and big bars. 
Ex: small = 1, big = 1, total = 10

2) Only big bars. 
Ex: small = 5, big = 3, total = 10

3) Need for big and small bars. 
Ex: small = 5, big = 3, total = 17

4) Only small bars.
Ex: small = 4, big = 2, total = 3

5) Invalid input. 
Ex: small = 4, big = 2, total = -1

(2,3,17) belongs to this partition!
But.. But... Does this mean that thinking about partitions is not enough? :( 
Hmm, with these inputs, **small = 2** is on the boundary of the required number of small bars!

\[
\begin{align*}
\text{small} &= 2 \\
\text{big} &= 3 \\
\text{total} &= 17
\end{align*}
\]

- Small = 1, not possible
- Small = 2, possible
- Small = 3, possible
Hmm, ok, let me think about the boundaries for each of these partitions, and do some **boundary testing**.

1) The total is higher than the amount of small and big bars.
2) Only big bars.
3) Need for big and small bars.
4) Only small bars.
The total is higher than the amount of small and big bars.

Ex: small = 1, big = 1, total = 10

small = 1, big = 1, total = 5, = 0
small = 1, big = 1, total = 6, = 1

small = 1, big = 1, total = 7, = -1
small = 1, big = 1, total = 8, = -1
Only big bars.

Ex: small = 5, big = 3, total = 10

small = 5, big = 0, total = 10, = -1
small = 5, big = 1, total = 10, = 5
small = 5, big = 2, total = 10, = 0
small = 5, big = 3, total = 10, = 0
Need for big and small bars.

Ex: small = 5, big = 3, total = 17

- small = 0, big = 3, total = 17, = -1
- small = 1, big = 3, total = 17, = -1
- small = 2, big = 3, total = 17, = 2
- small = 3, big = 3, total = 17, = 2

- small = 2, big = 3, total = 14, = -1
- small = 3, big = 3, total = 14, = -1

- small = 4, big = 3, total = 14, = 4
- small = 5, big = 3, total = 14, = 4
Only small bars.

Ex: small = 4, big = 2, total = 3

small = 4, big = 2, total = 3, = 3
small = 3, big = 2, total = 3, = 3

-------------
small = 2, big = 2, total = 3, = -1
small = 1, big = 2, total = 3, = -1
• If the score is between 100 and 200, the player gets 50 bonus points.
• If the total ordering is above $100.00, shipping costs is $5.00.
• ...

Let me test it! I do “off-by-one” mistakes all the time!
• If the score is between 100 and 200, the player gets 50 bonus points.
• If the total ordering is above $100.00, the shipping costs is $5.00.

Let me test it! I do “off-by-one” mistakes all the time!
\[ X \geq 100 \]
$X > 100$
score >= 100

- **On point:** *Exactly on boundary*
- **In point:** *Makes the condition true*
- **Out point:** *Makes the condition false*
- **Off point:**
  - *Flips the outcome for on point and*
  - *Is as close to boundary as possible*

On is 100; In is e.g. 200; Out is e.g. 50; Off is 99.
A Simplified Domain-Testing Strategy

BINGCHANG JENG
Sun Yat-Sen University
and
ELAINE J. WEYUKER
New York University

A simplified form of domain testing is proposed that is substantially cheaper than previously proposed versions, and is applicable to a much larger class of programs. In particular, the traditional constraints to programs containing only linear predicates and variables defined over continuous domains are removed. In addition, an approach to path selection is proposed to be used in conjunction with the new strategy.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging
General Terms: Terms: Reliability, Theory, Verification
Additional Key Words and Phrases: Domain testing, software testing

1. INTRODUCTION
Domain testing is a fault-based software-testing strategy proposed by White and Cohen [1978]. Testers have frequently observed that subdomain boundaries are particularly fault-prone and should therefore be carefully checked. Domain testing was proposed as a relatively sophisticated form of boundary value testing, and is applicable whenever the input domain is subdivided into subdomains by the program’s decision statements.

In this paper, a simplified version of domain testing is described which removes several limitations associated with earlier domain-testing strategies. In particular, our strategy is applicable to arbitrary types of predicates, detects both linear and nonlinear errors, for both discrete and continuous variable spaces. In addition, we will show that our new technique requires much smaller test suites than earlier versions, and will argue that its effectiveness is comparable to, and in some cases superior to, the others.
Simplified domain-testing strategy

- Handle boundaries independently
- For each boundary, *pick on and off point*
- While testing one boundary, use varying *in points* for the remaining boundaries.
- Use *domain matrix*. 
## Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"

<table>
<thead>
<tr>
<th>Variable</th>
<th>Condition</th>
<th>type</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&gt; 0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;= 10</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>&gt;= 1.0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Condition</td>
<td>type</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>x</td>
<td>&gt; 0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>&lt;= 10</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>&gt;= 1.0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Boundary conditions for "x > 0 && x <= 10 && y >= 1.0"

<table>
<thead>
<tr>
<th>Variable</th>
<th>Condition</th>
<th>type</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&gt; 0</td>
<td>on</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>&gt; 0</td>
<td>off</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>&lt;= 10</td>
<td>on</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>&lt;= 10</td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>&gt;= 1.0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>&gt;= 1.0</td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.9</td>
</tr>
<tr>
<td>y</td>
<td>typical</td>
<td>in</td>
<td>10.0</td>
<td>16.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test cases (x, y)
## Boundary conditions for "x > 0 && x <= 10 && y >= 1.0"

<table>
<thead>
<tr>
<th>Variable</th>
<th>Condition</th>
<th>type</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&gt; 0</td>
<td>on</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;= 10</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>&gt;= 1.0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>typical</td>
<td>in</td>
<td>10.0</td>
<td>16.0</td>
<td>109.3</td>
<td>2390.2</td>
<td>1.0</td>
</tr>
</tbody>
</table>
### Boundary conditions for "x > 0 && x <= 10 && y >= 1.0"

<table>
<thead>
<tr>
<th>Variable</th>
<th>Condition</th>
<th>type</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&gt; 0</td>
<td>on</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;= 10</td>
<td>on</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>&gt;= 1.0</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typical</td>
<td>in</td>
<td>10.0</td>
<td>16.0</td>
<td>109.3</td>
<td>2390.2</td>
<td>1.0</td>
<td>0.9</td>
</tr>
</tbody>
</table>
JUnit for multiple data points?

• 6 test cases, each with three values
  < x-value, y-value, outcome >

• For each test case:
  • Check that with given inputs method produces desired output.

• Hand-code in loop?

  Use JUnit 5
  @ParameterizedTest
/**
 * Verify that squares at key positions are properly set.
 * @param x Horizontal coordinate of relevant cell.
 * @param y Vertical coordinate of relevant cell.
 */

@ParameterizedTest
@CsvSource({
    "0, 0",
    "1, 2",
    "0, 1"
})

void testSquareAt(int x, int y) {
    assertEquals(board.squareAt(x, y)).isEqualTo(grid[x][y]);
}
Open & Closed Boundaries

Closed boundary
• Score $\geq 200$
  • On point =
  • Off point =

Open boundary
• Score $> 200$
  • On point =
  • Off point =
Multiple choice

Which of the following statements is true

A. An out point cannot also be an on point
B. The in point is included in the set of on points
C. If the on point is an in point, the off point is an out point.
D. An out point can never be an off point
Multiple choice

Which of the following statements is true

A. An out point cannot also be an on point
B. The in point is included in the set of on points
C. If the on point is an in point, the off point is an out point.
D. An out point can never be an off point
Chapter 7
Boundary conditions: the Correct way
Correct: Conformance

• Many data elements must conform to a specific format.
  • Example: e-mail (always name@domain).
• Test when your input is not in conformance with what is expected.
Correct: ordering

• The order of the data might influence the output.
• What happens if the list is ordered? Unordered?
Co[r]ect: range

• Inputs should usually be within a certain range.
  • Example: Age should always be greater than 0 and smaller than 120.
  • In most programming languages, basic types give you more than you need, e.g., int when you just need a number between 1-100.
Cor[r]ect: reference

• When testing a method, consider:
  • What it references outside its scope
  • What external dependencies it has
  • Whether it depends on the object being in a certain state
  • Any other conditions that must exist
Corr[e]ct: existence

- Does something really exist? What if it doesn’t?
Corre[c]t: cardinality

• Off-by-one errors
• Loops:
  • Zero
  • One
  • Many
Correc[t]: time

• Ordering in time
  • What happens if I forget to invoke a() before b()?
• Timeouts
• Date/Time operations
  • Should we use UTC? GMT?
• Concurrency
Random vs Partition testing

• Would it be better to simply test random inputs?
• Would it be more effective or less effective?
Random testing

• If generating random inputs is cheap, then even with a small budget (e.g., 1 day), we’d generate millions of tests. A human would only generate a few.
• Random testing is an ineffective way to find singularities in the large input space.

Partition testing

• Test designers usually exploit some knowledge of application semantics to choose samples that are more likely to include "special" or trouble-prone regions of the input space.
• Partition testing is more expensive than random testing.

Given a fixed budget, the optimum may not lie in only partition testing or only random testing, but in some mix that makes use of available knowledge.
Functional testing in large systems

- Functional specifications can be large and complex. Partition the specifications into features that can be tested separately.
- An ITF is a feature that can be tested independently of other functionalities of the software.
- Given an ITF, apply partition testing.
- Instantiate (concrete and executable) test cases.

Summary

• Functional (specification-based) tests
• Partition testing
• The Category-Partition method
• Equivalence class
• Boundary tests and boundary analysis
• Multiple boundary tests
• The CORRECT way
• Random testing vs Partition tests
References


