Specification-based Testing
Software Engineering
Gordon Fraser • Saarland University

Program behaviors
Specified
Implemented
Structural Testing

Program behaviors
Specified
Implemented
Functional Testing
### Program Behaviors

- Specified
- Implemented

**Structural + Functional Testing**

---

### Structural Testing

- Path coverage criteria
- Logic coverage criteria
- Dataflow coverage criteria
- Mutation testing

---

### Functional Testing

- Boundary Value Testing
- Equivalence Class Testing
- Decision Table-Based Testing
- Combinatorial Testing
- Grammar-based Testing
- Model-based Testing
**Specification-based Testing**

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

**Representative Values**

- Try to select inputs that are especially valuable
- Usually by choosing representatives of equivalence classes that are apt to fail often or not at all

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

**Boundary Value Analysis**
Boundary Value Testing

- Minimum, minimum+1, nominal, maximum-1, maximum
- Robustness testing
  Minimum-1, maximum+1
- Generalized - single fault assumption
  Boundary values for one, nominal values for others
- Worst-case testing
  All possible combinations

Single-fault assumption - therefore only one boundary value at a time

Single Fault Assumption

Failures occur rarely as the result of the simultaneous occurrence of two (or more) faults

<table>
<thead>
<tr>
<th>Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>1</td>
<td>Isosceles</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>100</td>
<td>2</td>
<td>Isosceles</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>Equilateral</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>199</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>100</td>
<td>200</td>
<td>Invalid</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>Equilateral</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>199</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>200</td>
<td>100</td>
<td>Invalid</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>Equilateral</td>
</tr>
<tr>
<td>14</td>
<td>199</td>
<td>100</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>15</td>
<td>200</td>
<td>100</td>
<td>100</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
Equivalence Partitioning

<table>
<thead>
<tr>
<th>Input condition</th>
<th>Equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>one valid, two invalid (larger and smaller)</td>
</tr>
<tr>
<td>specific value</td>
<td>one valid, two invalid (larger and smaller)</td>
</tr>
<tr>
<td>member of a set</td>
<td>one valid, one invalid</td>
</tr>
<tr>
<td>boolean</td>
<td>one valid, one invalid</td>
</tr>
</tbody>
</table>

How do we choose equivalence classes? The key is to examine input conditions from the spec. Each input condition induces an equivalence class – valid and invalid inputs.

Equivalence Partitioning

- Weak equivalence class testing
  One test per equivalence class per input

- Strong equivalence class testing
  All combinations (cartesian product of equivalence classes)

- Robustness testing
  Include invalid values

- Combination with boundary value testing
  Test at boundaries of partitions
### Decision Table Testing

Each column represents one test case

<table>
<thead>
<tr>
<th>a, b, c form a triangle</th>
<th>F</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>a = c</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>b = c</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Not a triangle</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalene</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Isosceles</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Equilateral</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impossible</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| a < b + c              | F | T | T | T | T | T | T | T | T |
| b < a + c              | F | T | T | T | T | T | T | T | T |
| c < a + b              | F | T | T | T | T | T | T | T | T |
| a = b                  | T | T | T | T | T | F | F | F | F |
| a = c                  | T | T | T | F | F | F | T | T | T |
| b = c                  | T | F | T | F | F | T | F | T | F |
| Not a triangle         | X | X | X |   |   |   |   |   |   |
| Scalene                | X |   |   |   |   |   |   |   |   |
| Isosceles              | X | X | X | X | X | X | X | X | X |
| Equilateral            | X |   |   |   |   |   |   |   |   |
| Impossible             | X | X | X |   |   |   |   |   |   |
Decision Tables

- Outcome of decisions are not necessarily binary
- Tables can become huge
- Limited entry tables with N conditions have $2^N$ rules
- Don't care entries reduce the number of explicit rules by implying the existence of non-explicitly stated rules.

Combinatorial Testing

```java
if (pressure < 10) {
    // do something
    if (volume > 300) {
        // faulty code! BOOM!
    }
    else {
        // good code, no problem
    }
}
else {
    // do something else
}
```
Interactions leading to Failure

- Medical device
- Browser
- Server
• Maximum interactions for fault triggering for studied applications was 6
  This correlates to the number of branch statements

• Reasonable evidence
  that maximum interaction strength for fault triggering is relatively small

• If all faults are triggered by the interaction of t or fewer variables
  then testing all t-way combinations can provide strong assurance

• Pairwise testing finds about 50% to 90% of flaws
How many tests?

- There are 10 effects, each can be on or off
- All combinations is \(2^{10} = 1,024\) tests
- What if our budget is too limited for these tests?
- Instead, let's look at all 3-way interactions …

\[
\binom{10}{3} = 120 \quad \text{3-way interactions}
\]

- Naively \(120 \times 2^3 = 960\) tests.
- Since we can pack 3 triples into each test, we need no more than 320 tests.
- Each test exercises many triples:

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}
\]

A Covering Array

- Each test covers 120 3-way combinations
- All 3-way combinations (960) in 13 tests
- Finding covering arrays is NP hard
Another familiar example

No silver bullet because:
- Many values per variable
- Need to abstract values
- But we can still increase information per test

A Larger Example

Suppose we have a system with on-off switches:

How do we test this?

- 34 switches = $2^{34} = 1.7 \times 10^{10}$ possible inputs = $1.7 \times 10^{10}$ tests
What if we knew no failure involves more than 3 switch settings?

- 34 switches = $2^{34} = 1.7 \times 10^{10}$ possible inputs = $1.7 \times 10^{10}$ tests
- If only 3-way interactions, need only 33 tests
- For 4-way interactions, need only 85 tests

Two ways of using combinatorial testing

Test data inputs

Test case

System under test

Configuration

Testing Configurations

- Example: app must run on any configuration of OS, browser, protocol, CPU, and DBMS
- Very effective for interoperability testing
Combinatorial testing with existent test suite

1. Use t-way coverage for system configuration values
2. Apply existing tests

<table>
<thead>
<tr>
<th>Test case</th>
<th>OS</th>
<th>CPU</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows</td>
<td>Intel</td>
<td>IPv4</td>
</tr>
<tr>
<td>2</td>
<td>Windows</td>
<td>AMD</td>
<td>IPv6</td>
</tr>
<tr>
<td>3</td>
<td>Linux</td>
<td>Intel</td>
<td>IPv6</td>
</tr>
<tr>
<td>4</td>
<td>Linux</td>
<td>AMD</td>
<td>IPv4</td>
</tr>
</tbody>
</table>

• Common practice in telecom industry

Generating Covering Arrays

• Search-based methods:
  • Mainly developed by scientists
  • Advantages: no restrictions on the input model, and very flexible, e.g., relatively easier to support parameter relations and constraints
  • Disadvantages: explicit search takes time, the resulting test sets are not optimal

• Algebraic methods:
  • Mainly developed by mathematicians
  • Advantages: very fast, and often produces optimal results
  • Disadvantages: limited applicability, difficult to support parameter relations and constraints

IPO Strategy

• Builds a t-way test set in an incremental manner
  • A t-way test set is first constructed for the first t parameters,
  • Then, the test set is extended to generate a t-way test set for the first t + 1 parameters
  • The test set is repeatedly extended for each additional parameter.

• Two steps involved in each extension for a new parameter:
  • Horizontal growth: extends each existing test by adding one value of the new parameter
  • Vertical growth: adds new tests, if necessary
Strategy In-Parameter-Order
begin
  /* for the first t parameters p1, p2, ..., pt */
  T := {(v1, v2, ..., vt) | v1, v2, ..., vt are values of p1, p2, ..., pt, respectively}
  if n = t then stop;
  /* for the remaining parameters */
  for parameter pi, i = t + 1, ..., n do
    begin
      /* horizontal growth */
      for each test (v1, v2, ..., vi-1) in T do replace it with (v1, v2, ..., vi-1, vi), where vi is a value of pi
      /* vertical growth */
      while T does not cover all the interactions between pi and each of p1, p2, ..., pi-1 do add a new test for p1, p2, ..., pi to T;
    end
  end
end

Example

• Consider a system with the following parameters and values:
  - parameter A has values A1 and A2
  - parameter B has values B1 and B2
  - parameter C has values C1, C2, C3

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B1</td>
<td>A2</td>
<td>B1</td>
<td>C3</td>
<td>A2</td>
<td>B1</td>
<td>C3</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>A2</td>
<td>B2</td>
<td>C1</td>
<td>A2</td>
<td>B2</td>
<td>C1</td>
</tr>
</tbody>
</table>

Horizontal Growth | Vertical Growth
Example

- Testing VoIP software:
  - Caller, VoIP server, client
  - CallerOS: Windows, Mac
  - ServerOS: Linux, Sun, Windows
  - CalleeOS: Windows, Mac

Example

<table>
<thead>
<tr>
<th>Caller</th>
<th>Server</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win</td>
<td>Lin</td>
<td>Win</td>
</tr>
<tr>
<td>Win</td>
<td>Sun</td>
<td>Mac</td>
</tr>
<tr>
<td>Win</td>
<td>Win</td>
<td>Win</td>
</tr>
<tr>
<td>Mac</td>
<td>Lin</td>
<td>Mac</td>
</tr>
<tr>
<td>Mac</td>
<td>Sun</td>
<td>Win</td>
</tr>
<tr>
<td>Mac</td>
<td>Win</td>
<td>Mac</td>
</tr>
</tbody>
</table>

1. Pairwise testing protects against pairwise bugs
2. while dramatically reducing the number of tests to perform compared to testing all combinations, but not necessarily compared to testing just the combinations that matter.
3. which is especially cool because pairwise bugs represent the majority of combinatoric bugs or might not, depending on the actual dependencies among variables in the product.
4. and such bugs are a lot more likely to happen than ones that only happen with more variables, or less likely to happen, because user inputs are not uniformly distributed.
5. Plus, you no longer need to create these tests by hand, except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results.