MTAT.03.159: Software Testing

Lecture 02:
Black-Box Testing
(Textbook Ch. 4)

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2017
Lecture >> Lab 2

- Chapter 4: Test case design
  - Black-box testing techniques (→ Lab 2 – Part 1)
Exam Dates

Exam 1: Tue 09-May, 15:15-17:15, room 405 – max. 46 students
Exam 2: Fri 12-May, 14:15-16:15, room 404 – max. 35 students
Exam 3: Mon 15-May, 15:15-17:15, room 405 – max 46 students

You must have at least 20 marks from the homework assignments (labs 1-6) to qualify for the exam.

You must receive at least 10 marks in the exam to not fail the course.

In total, you need at least 50 marks to not fail the course.
Structure of Lecture 2

• Black-Box vs. White-Box Testing
• Black-Box Testing Methods
• Lab 2 – Part 1
Testing is difficult

Assume a ’magic’ Function $M$

$M(x, y) \rightarrow z$

with $x, y$: int (32 bit)

Exhaustive testing:

$2^{32} \times 2^{32}$

$= 2^{64} \sim 1.8 \times 10^{19}$ test cases (input data + expected output)
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z \]

with \( x, y: \text{int (32 bit)} \)

Possible approaches:
- Execute each statement
- Read (review) code

```
... if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            z = x / (y - 100);
        }
    }
}
...```
Testing is difficult

Assume a 'magic' Function $M$

$M(x, y) \rightarrow z$

with $x, y$: int (32 bit)

1st if = true: $x \leq 100$

2nd if = true: $y \leq 100$

3rd if = true: $x + y = 200$

$M(100, 100) \rightarrow \text{crash}$

**White Box**

```java
...
if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            z = x / (y - 100);
        }
    }
}
...
```
Why Test Case Design Techniques?

Exhaustive testing (use of all possible inputs and conditions) is impractical; Therefore:

• Must use a subset of all possible test cases

• Must have high probability of detecting faults

Need strategies & techniques that help us selecting ‘good’ test cases

• Side-effect: makes different testers equally good in triggering failures

Effective testing – detect more faults

• Focus attention on specific types of faults

• Know you’re testing the right thing

Efficient testing – detect faults with less effort

• Avoid duplication

• Systematic techniques are measurable and repeatable
Basic Testing Strategies

<table>
<thead>
<tr>
<th>Test Strategy</th>
<th>Tester’s View</th>
<th>Knowledge Sources</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black box</td>
<td>Inputs</td>
<td>Requirements document, Specifications, Domain knowledge, Defect analysis, data</td>
<td>Equivalence class partitioning, Boundary value analysis, State transition testing, Cause and effect graphing, Error guessing</td>
</tr>
<tr>
<td>White box</td>
<td>Outputs</td>
<td>High-level design, Detailed design, Control flow graphs, Cyclomatic complexity</td>
<td>Statement testing, Branch testing, Path testing, Data flow testing, Mutation testing, Loop testing</td>
</tr>
</tbody>
</table>
Black-Box vs. White-Box

External/user view:
- Check conformance with specification -> function coverage
- Abstraction from details:
  - Source code not needed
- Scales up:
  - Different techniques at different levels of granularity

Internal/developer view:
- Allows tester to be confident about code coverage
- Based on control and data flow:
  - Easier debugging
- Does not scale up:
  - Most useful at unit & integration testing levels, as well as regression testing

USE BOTH!
Black-Box vs. White-Box

Gray-Box Testing

External/user view:
- Check conformance with specification -> function coverage
- Abstraction from details: Source code not needed
- Scales up: Different techniques at different levels of granularity

Internal/developer view:
- Allows tester to be confident about code coverage
- Based on control or data flow: Easier debugging
- Does not scale up: Most useful at unit & integration testing levels, as well as regression testing

Gray-Box Testing combines black-box and white-box testing; typically, the focus is on input/output testing (black-box view) which is informed by structural information of the code (white-box view).

Example: The tester knows that certain constraints on the input are checked by the unit under test.

Application, e.g., in regression testing: apply (or update) black-box test cases only where code has been changed;

Different techniques at different levels of granularity

Most useful at unit & integration testing levels, as well as regression testing

USE BOTH!
Black-Box vs. White-Box

**Specification-based Testing:** Test against specification

**System**

**Specification**

**Implementation**

**Goal of BBT:** Tries to check whether specified functionality is available and working correctly

**Unexpected functionality:** Cannot be (directly) revealed by black-box techniques
Black-Box vs. White-Box

System

Specification

Implementation

Structural Testing:
Test against implementation

Goal of WBT: Tries to check, whether the Implementation is working correctly (there is no dead code, it’s maintainable, etc.); useful for debugging;

Missing functionality:
Cannot be (directly) revealed by white-box techniques
Black-Box vs. White-Box

Specification-based Testing:
Test against specification

System

Implementation

Specification

Missing functionality:
Cannot be (directly) revealed
by white-box techniques

Unexpected functionality:
Cannot be (directly) revealed
by black-box techniques

Structural Testing:
Test against implementation
Structure of Lecture 2

• Black-Box vs. White-Box Testing
• Black-Box Testing Methods
• Lab 2 – Part 1
Black-Box Testing Techniques

• Equivalence class partitioning (ECP)
• Boundary value analysis (BVA)
• Cause and effect graphing
• Combinatorial testing
• State transition testing (State-based testing)
• Error guessing (Exploratory testing)
• A/B testing (UX)
Equivalence Class Partitioning (ECP)

• Split input/output space into classes which the software handles equivalently

• Select test cases to cover each class
public static boolean adultFunction(int age) {
    boolean adult;
    if (age >= 18)
        adult = true;
    else
        adult = false;
    return adult;
}

What are the ECs?

green = valid
white = invalid
ECP Guidelines [Myers 79/04]

If input condition:

- is a range, e.g., \( x = [0, 9] \) or
- is an ordered list of values, e.g., \( \text{owner} = <1, 2, 3, 4> \)
  \( \rightarrow \) one in-range/list and two out-of-range/list classes are defined
- is a set, e.g., vehicle is in \{car, motorcycle, truck\} or
- is a “must be” condition (boolean)
  \( \rightarrow \) one in-set and one out-of-set class are defined
- is anything else (e.g., invalid)
  \( \rightarrow \) partition further

Have enough test cases to cover all valid input classes
Have one test case for each invalid input class

Union set of all ECs should cover complete input/out space.
ECs must not overlap.
Example – Insurance System

Specification Statement:

• System shall reject over-age insurance applicants

Specification Item:

• Reject male insurance applicants, if over the age of 80 years on day of application

• Reject female insurance applicants, if over the age of 85 years on day of application
Example (cont.)

Input: Gender & Age | Output: accept/reject

UI – Case A

Age: in [18, 80] ? O
     in (80, 85] ? O
     in (85, 99] ? O
     other ? O

Gender: male O
        female O

Enter

Result: <text>
Message: <text>

Result text in {<empty>, accept, reject}
Message text in {<empty>, missing input}

UI – Case B

Please enter gender (m, f):
<Message text>
Please enter age (integer>0):
<Message text>
Result: <text>

Message text in {
    <empty>,
    invalid input –
    retry or quit with Ctrl^D}
Result text in {
    accept,
    reject}
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

C1: ???

Age:
- in [18, 80] ? O
- in (80, 85] ? O
- in (85, 99] ? O
- other ? O

Gender:
- male O
- female O

Enter

Result: <text>

Message: <text>

Result text in \{<empty>, accept, reject\}
Message text in \{<empty>, missing input\}
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

C1: InputAge: [18, 80]
C2: InputAge: (80, 85]
C3: InputAge: (85, 99]
C4: InputAge: other
C5: InputAge: <empty>
C6: InputGender: Male
C7: InputGender: Female
C8: InputGender: <empty>
C9: OutputResult: <empty>
C10: OutputResult: ‘accept’
C11: OutputResult: ‘reject’
C12: OutputMsg: <empty>
C13: OutputMsg: ’missing input’

What do you say about: C1, C3, and C4?

Result text in {<empty>, accept, reject}
Message text in {<empty>, missing input}
Example – UI Case A

Input: Gender & Age | Output: accept/reject

### Classes

| C1 | InputAge: [18, 80] |
| C2 | InputAge: (80, 85] |
| C3 | InputAge: (85, 99] |
| C4 | InputAge: other |
| C5 | InputAge: <empty> |
| C6 | InputGender: Male |
| C7 | InputGender: Female |
| C8 | InputGender: <empty> |
| C9 | OutputResult: <empty> |
| C10 | OutputResult: ‘accept’ |
| C11 | OutputResult: ‘reject’ |
| C12 | OutputMsg: <empty> |
| C13 | OutputMsg: ‘missing input’ |

### Test Cases

Data: age, gender, result, message

| TC1 | <empty>, <empty>, <empty>, ‘missing input’ |
| TC2 | 56, male, ’accept’, <empty> |
| TC3 | 83, male, ’reject’, <empty> |
| TC4 | 88, female, ’reject’, <empty> |
| TC5 | other, female, ’reject’, <empty> |

minimal, TCs cover all classes
Example – UI Case A
Input: Gender & Age | Output: accept/reject

**Classes**

C1: InputAge: [18, 80]
C2: InputAge: (80, 85]
C3: InputAge: (85, 99]
C4: InputAge: other
C5: InputAge: <empty>
C6: InputGender: Male
C7: InputGender: Female
C8: InputGender: <empty>
C9: OutputResult: <empty>
C10: OutputResult: ‘accept’
C11: OutputResult: ‘reject’
C12: OutputMsg: <empty>
C13: OutputMsg: ’missing input’

**Test Cases**

TC1: <empty>, male, <empty>, ’missing input’
TC2: other, <empty>, <empty>, ’missing input’
TC3: 56, male, ’accept’, <empty>
TC4: 83, male, ’reject’, <empty>
TC5: 88, female, ’reject’, <empty>
TC6: other, female, ’reject’, <empty>

If we consider ’missing input’ to be an error message caused by invalid input (<empty>), then it’s good practice to check for the effect of each invalid input class independently.
Example – UI Case A
Input: Gender & Age | Output: accept/reject

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>C1: [18, 80]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C2: (80, 85)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3: (85, 99)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4: other</td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td>Gender</td>
<td>C6: Male</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td>C8: &lt;empty&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TC</th>
<th></th>
<th></th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>56</td>
<td>83</td>
<td>88</td>
<td>oth F</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>em</td>
<td>em</td>
<td>M</td>
<td>M</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C5</td>
<td>C8</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>C6</td>
<td>C6</td>
<td>C7</td>
<td>C7</td>
</tr>
</tbody>
</table>

UNIVERSITY of TARTU
INSTITUTE of COMPUTER SCIENCE
Example – UI Case A
Input: Gender & Age | Output: accept/reject

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>C1: [18, 80]</td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C2: (80, 85]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3: (85, 99]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4: other</td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td>C6: Male</td>
<td>C8: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TC</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>em</td>
<td>oth</td>
<td>56</td>
<td>M</td>
<td>83</td>
<td>M</td>
</tr>
<tr>
<td>Gend.</td>
<td>M</td>
<td>em</td>
<td>88</td>
<td>F</td>
<td>oth</td>
<td>F</td>
</tr>
<tr>
<td>Tests</td>
<td>C5</td>
<td>C8</td>
<td>C1</td>
<td>C6</td>
<td>C2</td>
<td>C6</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>C7</td>
<td>C3</td>
<td>C7</td>
<td>C4</td>
<td>C7</td>
</tr>
</tbody>
</table>

What is missing?
Example – UI Case A

Input: Gender & Age | Output: accept/reject

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>C1: [18, 80]</td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C2: (80, 85]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3: (85, 99]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4: other</td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td>C6: Male</td>
<td>C8: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TC</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age Gend.</td>
<td>em M</td>
<td>oth M</td>
<td>56 M</td>
<td>83 M</td>
<td>88 F</td>
<td>oth F</td>
</tr>
<tr>
<td>Tests</td>
<td>C5</td>
<td>C8</td>
<td>C1 C6</td>
<td>C2 C6</td>
<td>C3 C7</td>
<td>C4 C7</td>
</tr>
</tbody>
</table>

Must also check coverage of output ECs!
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

C1: InputAge: [18, 80]
C2: InputAge: (80, 85]
C3: InputAge: (85, 99]
C4: InputAge: other
C5: InputAge: <empty>
C6: InputGender: Male
C7: InputGender: Female
C8: InputGender: <empty>
C9: OutputResult: <empty>
C10: OutputResult: ’accept’
C11: OutputResult: ’reject’
C12: OutputMsg: <empty>
C13: OutputMsg: ’missing input’

Test Cases

TC1: <empty>, male, <empty>, ’missing input’
TC2: other, <empty>, <empty>, ’missing input’
TC3: 56, male, ’accept’, <empty>
TC4: 83, male, ’reject’, <empty>
TC4*: 83, female, ’accept’, <empty>
TC5: 88, female, ’reject’, <empty>
TC6: other, female, ’reject’, <empty>

Now, we have covered all cause-effect relationships -> see Cause-Effect Graphing
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...’
C13: OutputResult: accept
C14: OutputResult: reject
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input …’
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

TC1: Ctrl^D
TC2: not(m, f), ’invalid’, Ctrl^D
TC3: m, <empty>, Ctrl^D
TC4: m, <empty>, other, ’invalid’, Ctrl^D
TC5: m, <empty>, [18, 80], <empty>, accept
TC6: m, <empty>, (80, 85], <empty>, reject
TC7: f, <empty>, <18, <empty>, reject
TC8: f, <empty>, >85, <empty>, reject
...

MTAT.03.159 / Lecture 02 / © Dietmar Pfahl 2017
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...’
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

TC1: Ctrl^D
TC2: g, ’invalid’, Ctrl^D
TC3: m, <empty>, Ctrl^D
TC4: m, <empty>, 3.5, ’invalid’, Ctrl^D
TC5: m, <empty>, 56, <empty>, accept
TC6: m, <empty>, 83, <empty>, reject
TC7: f, <empty>, 5, <empty>, reject
TC8: f, <empty>, 103, <empty>, reject
...
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...’
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

Every path from the root to a leaf is (at least) one test case
Boundary Value Analysis

• Adds to the equivalence partitioning method
• Select test cases to represent each side of the class boundaries
Boundary Value Analysis Guidelines

- Range $a..b \Rightarrow a, b, \text{just above } a, \text{just below } b$
- List of values: max, min, just below min, just above max
- Boundaries of externally visible data structures shall be checked (e.g. ordered sets, arrays)
- Output bounds should be checked
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

TC1: Ctrl^D
TC2: g, ’invalid’, Ctrl^D
TC3: m, <empty>, Ctrl^D
TC4: m, <empty>, 3.5, ’invalid’, Ctrl^D
TC5: m, <empty>, 56, <empty>, accept
TC5L: m, <empty>, 18, <empty>, accept
TC5U: m, <empty>, 80, <empty>, accept
TC6: m, <empty>, 83, <empty>, reject
TC7: f, <empty>, 5, <empty>, reject
TC8: f, <empty>, 103, <empty>, reject
...
Combinatorial Designs

- ECP and BVA define test cases per class
- Combinations of equivalence classes need to be handled
- Combinatorial explosion needs to be handled
Cause-Effect Graphing

'3' occurs if both/one of '1' and '2' are present

'2' occurs if '1' occurs

'2' occurs if '1' does not occur
One advantage of this method is that development of the rules and the graph from the specification allows for a thorough inspection (logically, semantically) of the specification.
Cause-Effect Graphing

1. The tester must decompose the specification into lower level units

2. For each specification unit, the tester needs to identify causes and effects.
   - A cause is a distinct input condition (or equivalence class of input conditions)
   - An effect is an output condition (or equivalence class of output conditions) or a system transformation
   - A table of causes and effects helps the tester to record the necessary details.
   - Then relationships between the causes and effects should be determined, ideally in the form of a set of rules (business rules).

3. From the set of rules, the boolean cause-effect graph is constructed, which in turn is converted into a decision table.

4. Test cases are then generated from the decision table.
Decision Table Format

<table>
<thead>
<tr>
<th>Case</th>
<th>Female</th>
<th>Male</th>
<th>&lt;= 80</th>
<th>&gt;80 &amp; &lt;= 85</th>
<th>&gt; 85</th>
<th>Accept</th>
<th>Reject</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TC2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TC3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TC4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A test-input would then be generated from this table, E.g.:
- TC1: male, 83 → reject
- TC2: male, 75 → accept (could also be female)
- TC3: female, 88 → reject (could also be male)
- TC4: female, 83 → accept
Combinatorial Testing – Example 0

2 input variables

- Age: 4 radio buttons + empty = 5 values
- Gender: 2 radio buttons + empty = 3 values

Minimum coverage of all ECs:

- 6 test cases

All input combinations?

Result text in {<empty>, accept, reject}
Message text in {<empty>, missing input}
Combinatorial Testing – Example 0

2 input variables
Age: 4 radio buttons + empty = 5 values
Gender: 2 radio buttons + empty = 3 values

Minimum coverage of all ECs:
6 test cases

All input combinations?
5 x 3 = 15 test cases
How Do We Test This?

34 switches (boolean) => $2^{34} = 1.7 \times 10^{10}$ possible inputs => $1.7 \times 10^{10}$ tests
How Do We Test This?

34 switches = $2^{34}$
= $1.7 \times 10^{10}$ possible inputs
= $1.7 \times 10^{10}$ tests
(34-way interaction)

1-way interactions:
= each switch at least once ‘on’ and once ‘off’
= 2 tests: 0000000000000000000000000000000000
1111111111111111111111111111111111

3-way interactions: need only 33 tests
4-way interactions: need only 85 tests
How does it work?

2 switches = 2 boolean variables (‘0’ or ‘1’):

all combinations:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2 x 2 = 4 tests

3 switches:

all combinations:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2³ = 8 tests
How does it work?

2 switches = 2 boolean variables (‘0’ or ‘1’):
all combinations:
\[
\begin{array}{cc}
X & Y \\
0 & 0 \\
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array}
\]
2 x 2 = 4 tests

3 switches:
all combinations:
\[
\begin{array}{ccc}
X & Y & Z \\
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]
2^3 = 8 tests

3 switches:
all 2-way interactions:
\[C(3, 2) \times 2^2 = 3 \times 4 = 12\]
\[
\begin{array}{ccc}
X & Y & Z \\
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]
4 tests
10 Booleans: 2-Way Interactions?

0 = effect off
1 = effect on

7 (6?) tests for all 2-way interactions

\[2^{10} = 1,024\] tests for all 10-way interactions
10 Booleans: 3-Way Interactions?

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

0 = effect off
1 = effect on

13 tests for all 3-way interactions

$2^{10} = 1,024$ tests for all 10-way interactions
Combinatorial Testing

What is it?

• Methods for systematically testing t-way interaction effects of input (or configuration parameter) values.

Why do it?

• The interaction of specific combinations of input values may trigger failures that won’t be triggered if testing input values (or configurations) only in isolation.
Combinatorial Testing – Example 1

Many variables
Many values per variable
Need to abstract values (equivalence classes, boundary values)

Plan: flt, flt+hotel, flt+hotel+car
From: CONUS, HI, AK, Europe, Asia ...
To: CONUS, HI, AK, Europe, Asia ...
Compare: yes, no
Date-type: exact, 1to3, flex
Depart: today, tomorrow, 1yr, Sun, Mon ...
Return: today, tomorrow, 1yr, Sun, Mon ...
Adults: 1, 2, 3, 4, 5, 6
Minors: 0, 1, 2, 3, 4, 5
Seniors: 0, 1, 2, 3, 4, 5
Two Scopes of Combinatorial Testing

Test Configurations

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

Pizza Ordering
System under test (SUT)
Combinatorial Testing – Example 2

Platform configuration parameters:

- OS: Windows XP, Apple OS X, Red Hat Linux
- Browser: Internet Explorer, Firefox
- Protocol: IPv4, IPv6
- CPU: Intel, AMD
- DBMS: MySQL, Sybase, Oracle

Total number of combinations: $3 \times 2 \times 2 \times 2 \times 3 = 72$

Do we need 72 test cases?
Pair-Wise Testing (2-Way Interaction)

# of pairs:

$$\binom{5}{2} = \frac{5!}{2!(5-2)!} = 10$$

# of 2-way interactions

N=57:

$$40 = \binom{5}{2}2^2 \leq N \leq \binom{5}{2}3^2 = 90$$

<table>
<thead>
<tr>
<th>Pair</th>
<th>Number of Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:b</td>
<td>3 x 2 = 6</td>
</tr>
<tr>
<td>a:c</td>
<td>3 x 2 = 6</td>
</tr>
<tr>
<td>a:d</td>
<td>3 x 2 = 6</td>
</tr>
<tr>
<td>a:e</td>
<td>3 x 3 = 9</td>
</tr>
<tr>
<td>b:c</td>
<td>2 x 2 = 4</td>
</tr>
<tr>
<td>b:d</td>
<td>2 x 2 = 4</td>
</tr>
<tr>
<td>b:e</td>
<td>2 x 3 = 6</td>
</tr>
<tr>
<td>c:d</td>
<td>2 x 2 = 4</td>
</tr>
<tr>
<td>c:e</td>
<td>2 x 3 = 6</td>
</tr>
<tr>
<td>d:e</td>
<td>2 x 3 = 6</td>
</tr>
</tbody>
</table>

------

57

What is the lower bound for the number of test cases?
Pair-Wise Testing (2-Way Interaction)

Number of pairs:
\[
\binom{5}{2} = \frac{5!}{2!(5-2)!} = 10
\]

Number of 2-way interactions
\[
N = \frac{2^5}{2} - \frac{2^2}{2} = 90
\]

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
Pair-Wise Testing (2-Way Interaction)

Only 9 tests needed, if we want to test all interactions of one parameter with one other parameter (pair-wise interaction)

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>

Pair 1:  
3 x 2 = 6 Combinations
## Pair-Wise Testing (2-Way Interaction)

Only 9 tests needed, if we want to test all interactions of one parameter with one other parameter (pair-wise interaction).

### Test 2:
2 x 2 = 4 Combinations

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
Pair-Wise Testing (2-Way Interaction)

Only 9 tests needed, if we want to test all interactions of one parameter with one other parameter (pair-wise interaction).

Pair 3:
3 x 3 = 9 Combinations

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
Is Testing 2-Way Interactions Enough?

Analyses of failure-triggering conditions showed this:

- Medical device (dark blue)
- NASA distrib. DB (light blue)
- Browser (green)
- Web server (magenta)
- Network security (orange)
- TCAS* module (purple)

* Traffic Collision Avoiding System

Several studies have shown that Pair-wise Testing triggers between 50% and 90% of all failures.

More strengths than 6-way interaction has hardly ever shown to find more defects.
ACTS Tool (NIST & UT Arlington)
ACTS – Example 1

3 variables: VAR1, VAR2, VAR3
2 values per variable: 0, 1
4 TCs
ACTS – Example 2

5 variables
2 or 3 values per variable

⇒ 9 TCs
ACTS – Example 2

<table>
<thead>
<tr>
<th>VAR1</th>
<th>VAR2</th>
<th>VAR3</th>
<th>VAR4</th>
<th>VAR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

5 variables
3 values per variable

→
15 TCs
Combinatorial Testing Links

http://cse.unl.edu/~citportal/

http://csrc.nist.gov/groups/SNS/acts/index.html
How to (Automatically) Generate Test Oracles?

Creating test (input) data is the (relatively) easy part!

How do we check that the code worked correctly on the test input?

- Using existing test sets (with known oracles) – easy if test sets exist
- Crash testing code to ensure it does not crash for randomly generated test input (‘fuzz testing’ ) – easy but of limited value
- Embedded assertions – incorporate assertions in code to check critical states at different points in the code – will they cover all possible incorrect states?
- Model-checking using a mathematical model of the system under test and a model checker to generate expected results for each input – expensive but tractable
Combinatorial Testing – Summary

Combinatorial testing makes sense where

More than ~8 variables and less than 300-400 Logical or numeric interaction of variables

New algorithms make large-scale combinatorial testing possible

Tool-support exists (prototypes)

Model-checking facilitates automatic test oracle generation
State-Transition Testing

Create a set of test cases that triggers each state-transition at least once
State Table

<table>
<thead>
<tr>
<th>Input (Event)</th>
<th>State</th>
<th>Wait for Card (S1)</th>
<th>Wait for PIN (S2)</th>
<th>Next (S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card inserted</td>
<td>Ask for PIN -&gt; S2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Invalid PIN</td>
<td>-</td>
<td>Beep -&gt; S2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Valid PIN</td>
<td>-</td>
<td>Ask amount -&gt; S3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cancel</td>
<td>-</td>
<td>Return card -&gt; S1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### State Table

<table>
<thead>
<tr>
<th>Input (Event)</th>
<th>State</th>
<th>Wait for Card (S1)</th>
<th>Wait for PIN (S2)</th>
<th>Next (S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card inserted</td>
<td>Ask for PIN -&gt; S2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Invalid PIN</td>
<td>-</td>
<td>Beep -&gt; S2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Valid PIN</td>
<td>-</td>
<td>Ask amount -&gt; S3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td>-</td>
<td>Return card -&gt; S1</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Test (=sequence of test cases):
S1 -> 'Card inserted' / 'Ask for PIN' -> S2 -> 'Cancel' / 'Return card' -> S1
State-Transition Testing: Example 2

Extract of a Specification Doc:

Parameters

- PORT_A: calling phone
- PORT_B: called phone

PORT_A identifies the connection from which a call is to be set up. The actual state of the call setup is globally available. Depending on this a new state arises after the evaluation of the transferred event. The delivered state is DISCONNECTED, if the call setup was terminated, it is DIALING, if the call setup is in progress but not completed yet. It is CONNECTED, if the call setup was successfully completed. In this case PORT_B delivers the connection of the selected subscriber, otherwise the data content of PORT_B is undefined. A call setup requires the sequence UNHOOK (DIGIT_N)* and the digit sequence must represent a valid number. HANG UP always leads to the complete termination of the call. If TIMEOUT occurs, HANG UP brings the software back into the initial state (DISCONNECTED)
State-Transition Testing: Example 2

State Chart
State-Transition Testing: Example 2

The minimal test strategy is to cover each state at least once.

A better solution is to cover each transition at least once, which leads, e.g., to the following tests:

- DISCONNECTED, unhook -> DIALING, hang up -> DISCONNECTED
- DISCONNECTED, unhook -> DIALING, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED
- DISCONNECTED, unhook -> DIALING, Digit 0..9 -> DIALING, Digit 0..9 -> DIALING, dialed number valid -> CONNECTED, hang up -> DISCONNECTED
- DISCONNECTED, unhook -> DIALING, Digit 0..9 -> DIALING, Digit 0..9 -> DIALING, dialed number invalid -> INVALID NUMBER, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED

Furthermore, it is useful to test all events, if transitions can be initiated by more than one event. The result is a hierarchy of test techniques:
all states ≤ all transitions ≤ all events

Important: Do not forget to test the failure treatment!
State-Transition Testing: Example 2

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>DISCONNECTED</th>
<th>DIALING</th>
<th>CONNECTED</th>
<th>INVALID NUMBER</th>
<th>TIMEDOUT OCCURRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>unhook</td>
<td>DISCONNECTED</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td></td>
</tr>
<tr>
<td>hang up</td>
<td>FAILURE</td>
<td>DISCONNECTED</td>
<td>FAILURE</td>
<td>DISCONNECTED</td>
<td>DISCONNECTED</td>
<td>DISCONNECTED</td>
</tr>
<tr>
<td>digit_0</td>
<td>DISCONNECTED</td>
<td>DIALING</td>
<td>CONNECTED</td>
<td>INVALID NUMBER</td>
<td>TIMEDOUT OCCURRED</td>
<td></td>
</tr>
<tr>
<td>digit_9</td>
<td>DISCONNECTED</td>
<td>DIALING</td>
<td>CONNECTED</td>
<td>INVALID NUMBER</td>
<td>TIMEDOUT OCCURRED</td>
<td></td>
</tr>
<tr>
<td>timeout</td>
<td>FAILURE</td>
<td>TIMEDOUT OCCURRED</td>
<td>FAILURE</td>
<td>TIMEDOUT OCCURRED</td>
<td>TIMEDOUT OCCURRED</td>
<td></td>
</tr>
<tr>
<td>dialed number valid</td>
<td>FAILURE</td>
<td>CONNECTED</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td></td>
</tr>
<tr>
<td>dialed number invalid</td>
<td>FAILURE</td>
<td>INVALID NUMBER</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td></td>
</tr>
</tbody>
</table>
State-Transition Testing: Example 2

State Chart
(with failure)
Error Guessing

- Exploratory testing (?), happy testing, ...
- Always worth including
- Can trigger failures that systematic techniques miss
- Consider
  - "What is the craziest thing we can do?"
  - Intuition / Experience / Brain storming
  - Past failures / Lists in literature
- Tools
Exploratory Testing

- Inventors:
  - Cem Kaner, James Bach (1990s)
- Definition:
  - “Exploratory testing is simultaneous learning, test design, and test execution.”
- Elements / Variants
  - Charter: defines mission (and sometimes tactics to use)
    - Example: “Check UI against Windows interface standards”
  - Session-based test management:
    - Defects + Notes + Interviews of the testers
A/B Testing

Two GUI Versions A & B
A/B Testing (cont’d)

Visual Website Optimizer divides traffic between the two versions

Tool support

Two GUI Versions A & B

Randomly selected
A/B Testing (cont’d)

Tools:
https://blog.crazyegg.com/2014/06/25/best-testing-software/

More Sales? 

Two GUI Versions A & B
A/B Testing (cont’d)

What to vary ...

• Call-To-Actions – Placement, wording, size
• Copywriting – Value propositions, product descriptions
• Forms – Their length, field types, text on the forms.
• Layout – Homepage, content pages, landing pages
• Product pricing – Try testing for revenue by testing your prices
• Images – Their placement, content and size
• Amount of content on the page (short vs. long)

Link:
http://conversionxl.com/how-to-build-a-strong-ab-testing-plan-that-gets-results/
Structure of Lecture 2

• Black-Box vs. White-Box Testing
• Black-Box Testing Methods
• Lab 2
Lab 2: Part 1 – Black-Box Testing

Lab 2 (week 27: Feb 28 - Mar 01) – Black-Box Testing (5%)

Submission Deadlines:
- Tuesday Labs: Monday, 06 Mar, 23:59
- Wednesday Labs: Tuesday, 07 Mar, 23:59

• Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 2: Part 1 – Black-Box Testing (cont’d)

Test Cases:
Input & Exp. Output
1 3 4 → triangle type x
2 5 5 → triangle type y
0 1 2 → no triangle
...

Strategies:
- Equivalence Class Partitioning
- Boundary Value Analysis

Test Report:
TC1 → pass
TC2 → pass
TC3 → pass
TC4 → fail → defect
...

Program
Recommended Textbook Exercises

Chapter 4
1, 2, 3, 4, 5
8, 11, 12

Practical Software Testing
Next Week

• Lab 1:
  – Submit assignment (either Monday or Tuesday by 23:59)
• Lecture 3:
  – White-Box Testing Techniques

• In addition to do:
  – Read textbook chapters 4 (available via OIS)