Coverage-Based Test Design

CS 3250
Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 5]
Today’s Objectives

• What is criteria-based test design?
• Why are test criteria used?
• Who will benefit from using test criteria? How?
• When are test criteria used?
• How are test criteria used?

• What are existing criteria? How are criteria categorized?
• Which criterion should be used? When? Why? How?

Later
All Possible Inputs?

• Let’s do activity
• Create all possible test inputs for the given program

```java
/**
 * Determine if the argument is a leap year in the Gregorian calendar
 * Assumes that arguments are in Gregorian calendar range (1582 and onwards)
 *
 * @param year value in range for Gregorian calendar
 * @return true iff year is a leap year
 */

public static boolean isLeap(int year)
{
    if (year % 4 != 0) return false;
    if (year % 400 == 0) return true;
    if (year % 100 == 0) return false;

    return true;
}
```

• It is impossible to provide all possible inputs
• Therefore, we need some rules to help us decide which inputs to enter and give us an idea if we test enough
Coverage Criteria

- Describe a finite subset of test cases out of the vast/infinite number of possible tests we should execute.
- Divide the input space to maximize the number of faults found per test case.
- Provide useful rules for when to stop testing.
Benefits of Coverage Criteria

Adequate
• Have I got enough tests?

Guidance
• Where should I test more?

Automation
• Generate test that satisfies a test requirement
Model-Driven Test Design

Test Design

- model / structure
- test requirements
- refined requirements / test specs

Implementation Abstraction Level
- software artifact

Design Abstraction Level
- test requirements

Test Automation
- input values
- test results
- test scripts
- test cases

Test Evaluation
- pass / fail

[AO, p.30]
Changing Notions in Testing

Old view (phase)

- Requirements Analysis
- Architectural Design
- Subsystem Design
- Detailed Design
- Implementation
- Acceptance Test
- System Test
- Integration Test
- Module Test
- Unit Test

New view (structures and criteria)

- Input space (sets)
  - A: \{0, 1, >1\}
  - B: \{undergraduate, graduate\}
  - C: \{1000, 2000, 3000, 4000\}

- Graphs
- Logical expressions
  - (not X or not Y) and A and B
- Syntax structures (grammar)
  - if (x > y)
    - \( z = x - y \);
  - else
    - \( z = 2 \times x \);

[AO, p 21]
Test design is largely the same at each phase

- Creating the **structure** is different
- Choosing **values** and automating the tests is different

Tester defines a structure of the software and then find ways to cover it

Structures can be extracted from lots of software artifacts

- **Graphs** – from UML use cases, finite state machines, source code, ...
- **Logical expressions** – from decisions in program source, guards on transitions, conditionals in use cases, ...

Input space (sets)

- A: \(\{0, 1, >1\}\)
- B: \(\{\text{undergraduate, graduate}\}\)
- C: \(\{1000, 2000, 3000, 4000\}\)

Syntax structures (grammar)

\[
\begin{align*}
\text{if} (x > y) \\
\quad z &= x - y; \quad \text{else} \\
\quad z &= 2 \times x;
\end{align*}
\]

(not X or not Y) and A and B
Test Coverage Criteria

Coverage Criterion

- A rule or collection of rules that impose test requirements on a test set

Test requirement

- A specific element of a software artifact that a test case must satisfy or cover
- Depends on the specific artifact under test

Test case

- A set of test inputs, execution conditions, and expected results, developed for a particular test scenario to verify whether the system under test satisfies a specific requirement

Test set

- A set of test cases
Example: Blow Pop Coverage

### Flavors
- Cherry
- Blue razz berry
- Strawberry
- Sour apple
- Grape
- Watermelon

### Colors
- Red (Cherry, strawberry, watermelon)
- Blue (Blue razz berry)
- Green (Sour apple)
- Purple (Grape)

Possible coverage criteria:

C1: Taste one blow pop of **each flavor**

(deciding if red blow pop is cherry, strawberry, or watermelon is a controllability problem)

C2: Taste one blow pop of **each color**
# Example: Blow Pop Coverage

## Flavors

- Cherry
- Blue razz berry
- Strawberry
- Sour apple
- Grape
- Watermelon

### Test requirements for $C_1$

<table>
<thead>
<tr>
<th>Test</th>
<th>Flavors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tr_1$</td>
<td>Cherry</td>
</tr>
<tr>
<td>$tr_2$</td>
<td>Blue razz berry</td>
</tr>
<tr>
<td>$tr_3$</td>
<td>Strawberry</td>
</tr>
<tr>
<td>$tr_4$</td>
<td>Sour apple</td>
</tr>
<tr>
<td>$tr_5$</td>
<td>Grape</td>
</tr>
<tr>
<td>$tr_6$</td>
<td>Watermelon</td>
</tr>
</tbody>
</table>

$$TR_1 = \{\text{Cherry, Blue razz berry, Strawberry, Sour apple, Grape, Watermelon}\}$$

## Colors

- Red (Cherry, strawberry, watermelon)
- Blue (Blue razz berry)
- Green (Sour apple)
- Purple (Grape)

### Test requirements for $C_2$

<table>
<thead>
<tr>
<th>Test</th>
<th>Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tr_1$</td>
<td>Red</td>
</tr>
<tr>
<td>$tr_2$</td>
<td>Blue</td>
</tr>
<tr>
<td>$tr_3$</td>
<td>Green</td>
</tr>
<tr>
<td>$tr_4$</td>
<td>Purple</td>
</tr>
</tbody>
</table>

$$TR_2 = \{\text{Red, Blue, Green, Purple}\}$$
Example: Source Code

```java
public static int numZero(int[] x) {
    // Effects: if x == null, throw NullPointerException
    // else return the number of occurrences of 0 in x
    int count = 0; // line1
    for (int i=1; i<x.length; i++) { // line2
        if (x[i] == 0) { // line3
            count++; // line4
        }
    }
    return count; // line5
}
```

Test requirements for **line coverage**

```text
TR = {line1, line2, line3, line4, line5}
```

Test requirements for **branch coverage**

```text
TR = {NPE-B1, B1, !B1, B2, !B2}
```
Coverage

• Given a set of test requirements $TR$ for coverage criterion $C$, a test set $T$ satisfies $C$ coverage if and only if for every test requirement $tr$ in $TR$, there is at least one test $t$ in $T$ such that $t$ satisfies $tr$

• Adequate test
• Minimal test
• Minimum test
**C1: Flavor criterion**

\[ TR1 = \{\text{Cherry, Blue razz berry, Strawberry, Sour apple, Grape, Watermelon}\} \]

**C2: Color criterion**

\[ TR2 = \{\text{Red, Blue, Green, Purple}\} \]

**Test sets**

\[ T1 = \{\text{one Cherry, one Blue razz berry, three Strawberries, one Sour apple, two Grapes, four Watermelons}\} \]

\[ T2 = \{\text{one Blue razz berry, one Sour apple, two Grapes, three Watermelons}\} \]

Satisfy C1?  
Satisfy C2?
Coverage Level

- It is sometimes expensive to satisfy a coverage criterion.
- Testers compromise by trying to achieve a certain coverage level.

Coverage level = \( \frac{\text{number of test requirements satisfied by } T}{\text{Size of } TR} \)
Blow Pop Coverage (continue)

**C1: Flavor criterion**

\[ TR1 = \{\text{Cherry, Blue razz berry, Strawberry, Sour apple, Grape, Watermelon}\} \]

**C2: Color criterion**

\[ TR2 = \{\text{Red, Blue, Green, Purple}\} \]

Test sets

**T1**

- one Cherry
- one Blue razz berry
- three Strawberries
- one Sour apple
- two Grapes
- four Watermelons

Satisfy C1? Coverage level 6 / 6
Satisfy C2? Coverage level 4 / 4

**T2**

- one Blue razz berry
- one Sour apple
- two Grapes
- three Watermelons

Satisfy C1? Coverage level 4 / 6
Satisfy C2? Coverage level 4 / 4
Infeasible Test Requirement

Example:
```java
/**
 * @param s1, s2, s3: sides of the putative triangle
 * @return enum describing type of triangle
 */

public static Triangle triangle (int s1, int s2, int s3)
```

Imagine if we have the following test requirements
TR = {all sides > 0, all sides = 0, all sides < 0}

- Some test requirements are infeasible (i.e., cannot be satisfied)
  - No test case values exist that meet the test requirements
  - Example: dead code
  - Detection of infeasible test requirements is undecidable for most test criteria

- 100% coverage is usually impossible in practice
Two Ways to Use Test Criteria

• Directly generate test case values to satisfy the criterion
  • Often assumed by the research community
  • Most obvious way to use criteria
  • Very hard without automated tools

• Generate test values externally and measure against the criterion -- to evaluate existing test sets
  • Usually favored by industry
  • Sometimes misleading
    • If tests do not reach 100% coverage, what does that mean?
    • We don’t have enough data to tell how much 99% coverage is worse than 100% coverage
Implementation of Test Criteria

Generator

- A procedure that automatically generate values to satisfy a criterion
- Automated test generation tools

Recognizer

- A procedure that decides whether a set of test case values satisfies a criterion
- Coverage analysis tools

It is possible to recognize whether test cases satisfy a criterion far more than it is possible to generate tests that satisfy the criterion
Comparing Criteria

Criteria Subsumption

- A test criterion $C_1$ subsumes $C_2$ if and only if every set of test cases that satisfies criterion $C_1$ also satisfies $C_2$
- Must be true for every test set

$C_1$ subsumes $C_2$ (superset)

$C_1$ subsumes $C_2$ (many-to-one)

$C_1$ subsumes $C_2$ (one-to-one)
**Blow Pop Coverage (Subsume)**

**C1: Flavor criterion**

\[ TR1 = \{\text{Cherry, Blue razz berry, Strawberry, Sour apple, Grape, Watermelon}\} \]

\[ C1 \text{ subsumes } C2 \]

**C2: Color criterion**

\[ TR2 = \{\text{Red, Blue, Green, Purple}\} \]

Test sets

\[ T1 = \{\text{one Cherry, one Blue razz berry, three Strawberries, one Sour apple, two Grapes, four Watermelons}\} \]

Satisfy C1? Satisfy C2?

Coverage level 6 / 6 4 / 4

\[ T2 = \{\text{one Blue razz berry, one Sour apple, two Grapes, three Watermelons}\} \]

Satisfy C1? Satisfy C2?

Coverage level 4 / 6 4 / 4
Characteristics of a Good Coverage Criterion

- It should be fairly easy to compute test requirements automatically
- It should be efficient to generate test values
- The resulting tests should reveal as many faults as possible

Additional notes:

- Subsumption is only a rough approximation of fault revealing capability
- Researchers still need to give us more data on how to compare coverage criteria
Advantages of Using Criteria

- Yield fewer tests that are more effective at finding faults
  - Design test inputs that are more likely to find problems

- Increase traceability
  - Answer the “why” for each test
  - Support regression testing

- Provide stopping rules for testing – “how many test” are needed

- Support test automation

- Make testing more efficient and effective

- Provide greater assurance that the software is of high quality and reliability

How do we start applying these ideas in practice

More comprehensive
Less overlap
How to Improve Testing?

- Test engineers need more and better software tools

- Test engineers need to adopt practices and techniques that lead to more efficient and effective testing
  - More education
  - Different management organizational strategies

- Testing / QA teams need more technical expertise
  - Developer expertise has been increasing dramatically

- Testing / QA teams need to specialize more
Changes in Practice

- **Reorganize** test and QA teams to make effective use of individual abilities – one math-head can support many testers

- **Retrain** test and QA teams
  - Use a process like MDTD
  - Learn more testing concepts

- Encourage researchers to
  - **Invent** processes and techniques
  - **Embed** theoretical ideas in tools
  - Demonstrate **economic value** of criteria testing
    - **Which** criteria should be used and **when**?
    - **When** does the extra effort pay off?

- **Get involved** in curricular design efforts through industrial advisory boards
Summary

• Many companies still use “monkey testing”
  • A human sits at the keyboard, wiggles the mouse and bangs the keyboard
  • No automation
  • Minimal training required

• Some companies automate human-designed tests

• But companies that use both automation and criteria-based testing save money, find more faults, and build better software
What’s Next?

Structures for Criteria-Based Testing

Four structures for modeling software

- **Input space**
  - Source
  - Design
  - Specs
  - Use cases

- **Graph**
  - Source
  - Design
  - Specs
  - Use cases

- **Logic**
  - Source
  - Specs
  - FSMs
  - DNF

- **Syntax**
  - Source
  - Models
  - Integration
  - Inputs