Program Analysis & Testing

Class #5

With material from Jim Jones, Mauro Pezze, Michal Young, and Andreas Zeller.

Software Testing

Functional and Structural Testing
Functional vs. Structural Testing

- Based on a SW spec
- Cover as much specified behavior as possible
- Cannot reveal errors due to implementation details

- Based on the code
- Cover as much coded behavior as possible
- Cannot reveal errors due to missing paths

Structural Testing Example

**Specification:** function that inputs an integer and prints it

**Implementation:**

- Function foo contains a typo
- From the functional perspective, ints < 1024 and ints > 1024 are equivalent, but they are treated differently in the code
- The fault may be missed by black-box testing
- The fault would be easily revealed by white-box testing (e.g., by statement coverage)
**Functional Testing Example**

**Specification**: function that inputs an integer *param* and returns half its value if *param* is even, *param* otherwise

**Implementation**:

1. `int foo(int param) {`
2. `int result;`
3. `result = param/2;`
4. `return (result); }`

- Function `foo` works correctly only for even integers
- The fault may be missed by structural testing (100% coverage with any value)
- The fault would be easily revealed by black-box testing (typically, we would use at least one odd and one even input)
Advantages of Functional Testing

- Focus on the domain of the input data
- No need for the code
- Early functional test design can
  - Reveal spec problems
  - Assess testability
  - Give additional explanation of spec
  - May even serve as spec (e.g., in XP)
- Best for missing logic defects
- Applies at all granularity levels

Test Data Selection

Given a spec for a functionality F, and thus a domain D for F, how do I select test inputs for F from D?
Exhaustive Testing?

How long would it take (approximately) to test exhaustively the following program?

```c
printSum(int a, int b) {...}
```

- $2^{32} \times 2^{32} = 2^{64} \approx 10^{19}$ tests
- Assume 1 test per nanosecond ($10^9$ tests/sec)
- we get $10^{10}$ seconds…
- **About 600 years!**

Random Testing

- Pick possible inputs uniformly
- Treat all inputs as equally valuable
- Avoid designer bias
- Test designers can make the same logical mistakes and bad assumptions as program designers (especially if they are the same person)
So why not random?

- Defects are not distributed uniformly
  - Assume our `printSum` program fails if both inputs are 0
  - Random sampling is unlikely to choose $a = 0$ and $b = 0$
- We want bias!

A Systematic Approach

Deriving test cases from a functional specification is a complex analytical process
Brute force generation of test cases is generally an inefficient and ineffective approach
A systematic approach simplifies the overall problem by dividing the process in elementary steps
- Decoupling of different activities
- Dividing brain-intensive from steps that can be automated
- Monitoring of the testing process
Independently Testable Features

- Decompose system into independently testable features (ITF)
- An ITF need not correspond to units or subsystems of the software
- For system testing, ITFs are exposed through user interfaces or APIs

Example:

`printSum(int a, int b) {...}

What are the independently testable features?
Independently Testable Features

Functional specification ➔ Independently testable features

Consider a spreadsheet

What are some examples of independently testable features?

 Independently Testable Features

Functional specification ➔ Independently testable features

Representative values ➔ Model

Model ➔ Test case specifications

Test cases ➔ Test case specifications

generate

identify

derive

identify

derive

identify

derive
Representative Values

- Functional specification
  - identify
  - Independently testable features
    - Identify
    - derive
  - Representative values
    - identify
    - derive
  - Model
  - derive
  - Test cases
    - derive
    - generate
  - Test case specifications
    - derive

- Try to select inputs that are especially valuable
- Usually by choosing representatives of equivalence classes
Systematic Partition Testing

- Failure (valuable test case)
- No failure

- Failures are sparse in the space of possible inputs ...
- ... but dense in some parts of the space

- The space of possible input values (the haystack)

If we systematically test some cases from each part, we will include the dense parts.

Functional testing is one way of drawing lines to isolate regions with likely failures.

Equivalence Partitioning

**Basic idea:** identify test cases that may reveal classes of errors (e.g., erroneous handling of all inputs > 100)

- Partitioning the input domain in classes from which to derive test cases
- A class is a set of data whose components are likely to be treated homogeneously by the program
- Ideal case: all test cases in a class have the same outcome
Equivalence Partitioning

• How do we choose equivalence classes?
• Examine input conditions from the spec; input conditions induce equivalence classes of valid and invalid inputs

<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>

Boundary Values

Basic idea: errors tend to occur at the boundaries of the data domain ➞ select inputs that exercise such boundaries

Complementary to equivalence partitioning: select for each equivalence class one or more boundary values
Representative Values

- Functional specification
- Independently testable features
- Representative values
- Model
- Test cases
- Test case specifications

Model

- Functional specification
- Independently testable features
- Representative values
- Model
- Test cases
- Test case specifications
Models: FSMs

- Nodes: states of the system
- Edges: transitions between states
- Edge attributes: events and actions

Building an FSM from a spec

- Identify boundaries of the system
- Identify inputs to the system
- Identify states of the system (trade-off abstraction level/number of states)
- Identify outputs of the system
From an informal specification...

**Maintenance:** The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty, maintenance can be requested either by calling the maintenance toll free number or through the web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the item is shipped the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item is returned to the residence, the item is picked up at the customer site, otherwise, the customer shall be responsible to pick up the item.

If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall be responsible to pick up the item.

If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate.

If the maintenance contract number provided by the customer is not valid, the item is returned to the residence, the item is picked up at the customer site, otherwise, the customer shall be responsible to pick up the item.

Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise). If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.
...to a test suite

| TC1 | 0 2 4 1 0 |
| TC2 | 0 5 2 4 5 6 0 |
| TC3 | 0 3 5 9 6 0 |
| TC4 | 0 3 5 7 5 8 7 8 9 6 0 |

Meaning: From state 0 to state 2 to state 4 to state 1 to state 0

- Is this a thorough test suite?
- How can we judge?

"Covering" finite state machines

**State coverage:**
- Every state in the model should be visited by at least one test case

**Transition coverage:**
- Every transition between states should be traversed by at least one test case.
- This is the most commonly used criterions

Path coverage
- Various types
- Less used
State based testing: Considerations

Applicability
- Menu-driven Software
- Object-oriented software
- Device driver
- Installation software

Limitations?
State based testing: Considerations

Applicability
- Menu-driven Software
- Object-oriented software
- Device driver
- Installation software

Limitations?
- Problems in identifying states, mapping
- Problem in constructing oracles (What is the state of the system? How do you check events/actions?)

Other possible models
- Decision tables
- Flow graphs
- Grammars
- ...
Models – summary

Models are useful abstractions
- Emphasize key features and suppress details
- Convey structure and help us focus on one thing at a time

Can be used in systematic testing
- A model divides behavior into classes that we should exercise

Very general approach
Typically, one uses models and representative values to generate test cases. For input values enumerated in previous step, we now need to take care of combinations.
Combinatorial Testing

- Eliminate invalid combinations
  e.g., IIS only runs on Windows
- Cover all pairs of combinations
  e.g., MySQL on Windows and Linux
- Combinations typically generated automatically
  and (hopefully) tested automatically, too

Category-Partition Method


Black-box technique that can be used at different levels of granularity

Generates a set of test cases from a functional specification in several steps
A. Identify independently testable features
B. Identify categories
C. Partitioning the categories into choices
D. Identify constraints among choices
E. Produce and evaluate test-case specifications
F. Generate test cases from test-case specifications
Category-Partition Method: Example

**NAME**

grep - search a file for a pattern

**SYNOPSIS**

grep <pattern> <filename>

**DESCRIPTION**

The `grep` utility searches files for a pattern and prints all lines that contain that pattern on the standard output. A line that contains multiple occurrences of the pattern is printed only once.

The pattern is any sequence of characters. To include a blank in the pattern, the entire pattern must be enclosed in single quotes ('). To include a quote sign in the pattern, the quote sign must be escaped (\'). In general, it is safest to enclose the entire pattern in single quotes ‘…’.

---

Step E: Example

DEMO
Model

- Functional specification
- Independently testable features
- Representative values
- Model

Test cases
- Test case specifications

Test Cases

- Functional specification
- Independently testable features
- Representative values
- Model

Test cases
- Test case specifications
Test Cases

• Implement test cases in code
• Requires building scaffolding
  • Drivers
  • Stubs

Infrastructure for unit testing

XUnit: automated unit testing frameworks

• Available for multiple languages & environments
  • JUnit (Java)
  • cppUnit (C++)
  • nUnit (.NET languages)
  • dbUnit (database testing)
  • HTTPUnit (testing web sites)
  • PHPUnit (PHP)
  • ...
• Acts as a driver
• Automates test runs
• Automates result checks
Structure of tests

- Set fixture
- Invoke
- Check
- Cleanup

Characteristics of tests

- Simple/no logic
  - Need complex logic? Move it into a utility method or class
  - Difficult to write test? Unit is too large, break it down
- Many unit tests (easy), few functional system tests (complex)
- Design for testability helps
Erratic tests

Some tests are non-deterministic?

- Mock more
- Eliminate “resource optimism”
- Eliminate shared fixtures

Structural Testing
White-Box Testing

Selection of test suite is based on some elements in the code

Assumption: Executing the faulty element is a necessary condition for revealing a fault

We will consider several examples

• Control flow (statement, branch, basis path, path)
• Condition (simple, multiple)
• Loop
• Dataflow (all-uses, all-du-paths)
• Fault based (mutation)

Let’s go back to printSum

printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
    [else do nothing]
}
Coverage Criteria

Defined in terms of
• test requirements

Result in
• test specifications
• test cases

Selection VS adequacy/evaluation criteria

printSum: test requirements

printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol(“red”, result);
    else if (result < 0)
        printcol(“blue”, result);
    [else do nothing]
}

req #1

req #2

req #3
**printSum: test specifications**

```cpp
printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
    [else do nothing]
}
```

**printSum: test cases**

```cpp
printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
    [else do nothing]
}```
Statement Coverage

Test requirements: Statements in program

\[ C_{stmts} = \frac{\text{(number of executed statements)}}{\text{(number of statements)}} \]

printSum: statement coverage

```cpp
printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
}
```
printSum: statement coverage

```java
public void printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
}
```

Coverage: 0%

printSum: statement coverage

```java
public void printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
}
```

Coverage: 71%
printSum: statement coverage

```
printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
}
```

Coverage: 71%

---

printSum: statement coverage

```
printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
}
```

Coverage: 100%
Statement Coverage in Practice

Only about 1/3 of NASA statements were executed under test before software was released (Stucki 1973)
Microsoft reports 80-90% statement coverage
Boeing must get 100% statement coverage (feasible) for all software
Usually can about 85% coverage; 100% is harder
  • Unreachable code; dead code
  • Complex sequences
  • Not enough resources

Test Criteria

Path testing -> Boundary interior testing
  \[ \text{subsumes} \]
  Compound condition testing
  MC/DC testing
  Branch and condition testing
  Basic condition testing

Loop boundary testing
  Statement testing
```java
public class Main {
    public static void main(String[] args) {
        printSum(3, 9);
        printSum(-5, -8);
    }

    public static void printSum(int a, int b) {
        int result = a + b;
        if (result > 0)
            printcol("red", result);
        else if (result < 0)
            printcol("blue", result);
    }

    public static void printcol(String color, int result) {
        // Print result with specified color
    }
}
```
Test requirements: Branches in the program

\[ C_{\text{branches}} = \frac{\text{number of traversed branches}}{\text{number of branches}} \]

**printSum**: branch coverage

```c
int result = a + b;
if (result > 0)
    printcol(“red”, result);
else if (result < 0)
    printcol(“blue”, result);
[missing statement]
```

**Coverage:** ?
### printSum: branch coverage

```java
void printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
    [missing statement]
}
```

**Coverage:** 88%

<table>
<thead>
<tr>
<th>a == 3</th>
<th>a == -5</th>
<th>a == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b == 9</td>
<td>b == -8</td>
<td>b == 0</td>
</tr>
</tbody>
</table>

```java
void printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
    [missing statement]
}
```

**Coverage:** 88%
```java
printSum(int a, int b) {
    int result = a + b;
    if (result > 0)
        printcol("red", result);
    else if (result < 0)
        printcol("blue", result);
    [missing statement]
}
```

**Coverage:** 100%

---

**Test Criteria**

- Path testing
  - Boundary interior testing
  - LCSAJ testing
  - Loop boundary testing
- Compound condition testing
  - MC/DC testing
  - Branch and condition testing
- Basic condition testing
  - Statement testing

---

• 8/1/08
Branch Coverage: Example

1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0) || (y>0)  // !(x=0 || y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}

• Consider test cases
  \{(x=5,y=5), (x=5,y=-5)\}
Branch Coverage: Example

1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0)||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}

- Consider test cases
  \{(x=5,y=5), (x=5, y=-5)\}
- The test suite is adequate
  for branch coverage, but
  does not reveal the fault
  at statement 6
- Predicate 4 can be true
  or false operating on
  only one condition
  ➔ Basic condition coverage

Basic Condition Coverage

Test requirements: Truth values assumed by basic conditions

\[ C_{bc} = \frac{\text{(number of boolean values assumed by all basic conditions)}}{\text{(number of boolean values of all basic conditions)}} \]
### Basic Condition Coverage: Example

```c
void main() {
    float x, y;
    read(x);
    read(y);
    if (x == 0) || (y > 0) {
        y = y / x;
    } else {
        x = y + 2;
    }
    write(x);
    write(y);
}
```

- Consider test cases \{(x=0, y=-5), (x=5, y=5)\}
- The test suite is adequate for basic condition coverage, but is not adequate for branch coverage.

#### Branch and Condition Coverage

**Test requirements:** Branches and truth values assumed by basic conditions

```c
if((a || b) && c) {
    ...}
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Compound Condition Coverage

Also known as multiple condition coverage

**Test requirements:** All possible combinations of basic conditions

Very thorough, but also very expensive for non-trivial programs.
Modified Condition/Decision Coverage (MC/DC)

**Key idea:** Test important combinations of conditions, avoiding exponential blowup

A combination is “important” if each basic condition is shown to independently affect the outcome of each decision
Modified Condition/Decision Coverage (MC/DC)

MC/DC criterion: For each basic condition C, there are two test cases in which the truth values of all conditions except C are the same, and the compound condition as a whole evaluates to True for one of those test cases and False for the other.

MC/DC: Example

( a && b && c )

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
### MC/DC: Example

\[( a \&\& b \&\& c ) \]

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
### MC/DC: Example

(a && b && c)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
MC/DC: Example

\((a \&\& b \&\& c)\)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

MC/DC: Example

\((a \&\& b \&\& c)\)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
### MC/DC: Example

\((a \&\& b \&\& c)\)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

### MC/DC: Example

\((a \&\& b \&\& c)\)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

1  True  True  True  True
5  False  True  True  False
3  True  False  True  False
MC/DC: Example

\((a \&\& b \&\& c)\)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Modified Condition/Decision Coverage (MC/DC)

Tradeoff between number of required test cases and thoroughness of the test

Required by both US and European quality standards in aviation
Other criteria

Path coverage: cover each path in the program

Loop coverage (given n): cover all paths that contain at most n iterations of any loop in the program

Data-flow coverage: cover data-flow relations in the program (du pairs)
Structural Testing: Summarizing

- Formal: works on a formal model
- Two broad classes
  - Practical criteria (test suite size linear in the program size)
  - Theoretical criteria (too many or infinite number of test cases)
- Comparable criteria (subsume relation)
- Most widely used as adequacy criteria because fully automatable!

Is path coverage enough?

Generate a set of test cases to achieve path coverage on the following program

1. int i;
2. read( i );
3. print( 10 / i );

Does it reveal the fault at line 3?

It may seem a tautology, but only exhaustive testing is exhaustive!
Issues with structural testing

Generate a set of test cases to achieve statement coverage on the following program

1. read(j);
2. int i=0, j;
3. if( ( j > 5 ) && ( i > 0 ) )
4. print( i );
5. print( j );

- Statements may not be executed due to defensive programming or code reuse
- Conditions may not be satisfiable due to interdependent conditions
- Paths may not be executable due to interdependent decisions

How does this problem affect coverage criteria?
How can we address the problem?

Software Testing

Tools
Tools

- Program Analysis Tools
- Instrumentation Tools
- Analysis tools
  - SOOT
- Static Checkers
  - Codesurfer
  - Eclipse (built-in)
  - FindBugs
  - PMD
- Testing Tools
  - JUnit

Software Testing

Debugging
Debugging Phases

- Run some test cases
- Observe failure
- Locate bugs
- Identify possible fixes
- Choose best fix
- Implement fix

Debugging Phases

- Run some test cases
- Observe failure
- **Locate bugs – fault localization**
- Identify possible fixes
- Choose best fix
- Implement fix
Current Practice

- Print statements
- Symbolic debugger
  - Breakpoints
  - Single stepping
- Limitations
  - Manual
  - Lots of data to observe
  - Single execution

Automated Debugging Techniques

- Static Slicing
- Dynamic Slicing
- Critical Slicing
- Dicing
- Nearest Neighbors
- Tarantula
- Statistical Debugging
- Cause Transitions
Automated Debugging Techniques

- Static Slicing
- Dynamic Slicing
- Critical Slicing
- Dicing
- Nearest Neighbors
- **Tarantula**
- Statistical Debugging
- Cause Transitions

Slicing

Weiser proposed slicing
Suggested its use for debugging
Slice backward from the output that caused failure

Limitations
Example

```c
mid() {
    int x,y,z,m;
    1: read("Enter 3 numbers:",x,y,z); 
    2: m = z; 
    3: if (y<z) 
    4:    if (x<y) 
    5:        m = y; 
    6:    else if (x<z) 
    7:        m = y; // bug 
    8:    else 
    9:    if (x>y) 
   10:        m = y; 
   11: else if (x>z) 
   12:        m = x; 
   13: print("Middle number is:", m); 
}
```

Test Cases

<table>
<thead>
<tr>
<th>Test</th>
<th>3.1.3</th>
<th>1.2.3</th>
<th>3.2.1</th>
<th>5.5.5</th>
<th>5.3.4</th>
<th>2.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pass Status: P P P P F

Slicing

Weiser proposed slicing
Suggested its use for debugging
Slice backward from the output that caused failure

Limitations
- Much of program included
- No execution data is used
- Must identify responsible output statement and variable
# Dynamic Slicing

Korel and Laski introduce dynamic slicing

Agrawal suggests using dynamic slices for fault localization

Slice backward from failure-causing output

## Limitations

```
mid() {
  int x,y,z,m;
  1: read("Enter 3 numbers:" ,x,y,z);
  2: m = z;
  3: if (y<z)
      4: if (x<y)
          5: m = y;
      6: else if (x<z)
          7: m = y; // bug
      8: else
          9: if (x>y)
              10: m = y;
          11: else if (x>z)
              12: m = x;
  13: print("Middle number is:" , m);
}
```

### Example

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Pass Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.35</td>
<td>P</td>
</tr>
<tr>
<td>3.23</td>
<td>P</td>
</tr>
<tr>
<td>3.21</td>
<td>P</td>
</tr>
<tr>
<td>5.55</td>
<td>P</td>
</tr>
<tr>
<td>5.34</td>
<td>P</td>
</tr>
<tr>
<td>2.13</td>
<td>F</td>
</tr>
</tbody>
</table>

Test Cases:

- 3,3,5: P
- 1,2,3: P
- 3,2,1: P
- 5,5,5: P
- 5,3,4: P
- 2,1,3: F
Dynamic Slicing

Korel and Laski introduce dynamic slicing
Agrawal suggests using dynamic slices for fault localization
Slice backward from failure-causing output

Limitations
• Still much of program
• Only one execution
• Must identify failure-causing output

Tarantula

Jones, Harrold, and Stasko suggest use of “fuzzy slices”
Inclusion of statement in slice is not a Boolean function, but instead a “grade of membership”

Intuition: Statements that are primarily executed by failed test cases are more suspicious than statements that are primarily executed by passed test cases

\[
\text{suspiciousness}(s) = \frac{\text{failed}(s)}{\text{total failed}} + \frac{\text{failed}(s)}{\text{total passed}}
\]
Tarantula

\[
suspiciousness(s) = \frac{\text{failed}(s)}{\text{total failed}} + \frac{\text{passed}(s)}{\text{total passed}}
\]

```c
mid() {
    int x,y,z,m;
    1: read("Enter 3 numbers:",x,y,z);
    2: m = z;
    3: if (y<z) { susp(1) = \frac{1}{1+\frac{1}{1}} = 0.5 }
    4:   if (x<y) { susp(5) = \frac{1}{\frac{1}{1}+\frac{1}{1}} = 0.7 }
    5:     m = y; // bug
    6:   else if (x<z) { susp(7) = \frac{1}{\frac{1}{1}+\frac{1}{1}} = 0.8 }
    7:     m = y; // bug
    8: else { susp(13) = \frac{1}{\frac{1}{1}+\frac{1}{1}} = 0.5 }
    9: if (x>y) { susp(1) = \frac{1}{1+\frac{1}{1}} = 0.5 }
    10:   m = y; // bug
    11: else if (x>z) { susp(7) = \frac{1}{\frac{1}{1}+\frac{1}{1}} = 0.8 }
    12:     m = x; // bug
    13: print("Middle number is:", m);
}
```

Pass Status: P P P P P F

Experimental Results

- Tarantula on Space (~6000 LOC)
- Tarantula on Siemens (~500 LOC)
Tarantula Tool

For statement $s$:

- **Hue** summarizes pass/fail results of test cases that executed $s$
- **Brightness** presents the “confidence” of the hue assigned to $s$

```
mid() {
  int x,y,z,m;
  1:   read("Enter 3 numbers:\n",x,y,z);
  2:   m = z;
  3:   if (y<z)
  4:     if (x<y)
  5:       m = y;
  6:     else if (x<z)
  7:       m = y;
  8:   else
  9:     if (x>y)
 10:       m = y;
 11:     else if (x>z)
 12:       m = x;
 13:   print("Middle number is:\n", m);
}
```

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>3.35</th>
<th>3.23</th>
<th>3.21</th>
<th>5.55</th>
<th>5.34</th>
<th>2.13</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td>Pass Status</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
</tr>
</tbody>
</table>
SeeSoft view

each pixel represents a character in the source

```c
mid() {
    int x, y, z, m;
    read("Enter 3 numbers:", x, y, z);
    m = z;
    if (y<z)
        if (x<y)
            m = y;
        else if (x<z)
            m = y;
    else
        if (x>y)
            m = y;
        else if (x>z)
            m = x;
    print("Middle number is:", m);
}
```
System-level View

**TreeMap view**
each node
- represents a file
- is divided into blocks representing color of statements

Tarantula
Summary

• Presented techniques
  • Static Slicing
  • Dynamic Slicing
  • Tarantula

• Why are we still using print statements and breakpoints?