# Software Architecture and Engineering

## *Test Case Selection*

**Peter Müller**

Chair of Programming Methodology

Spring Semester 2012

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

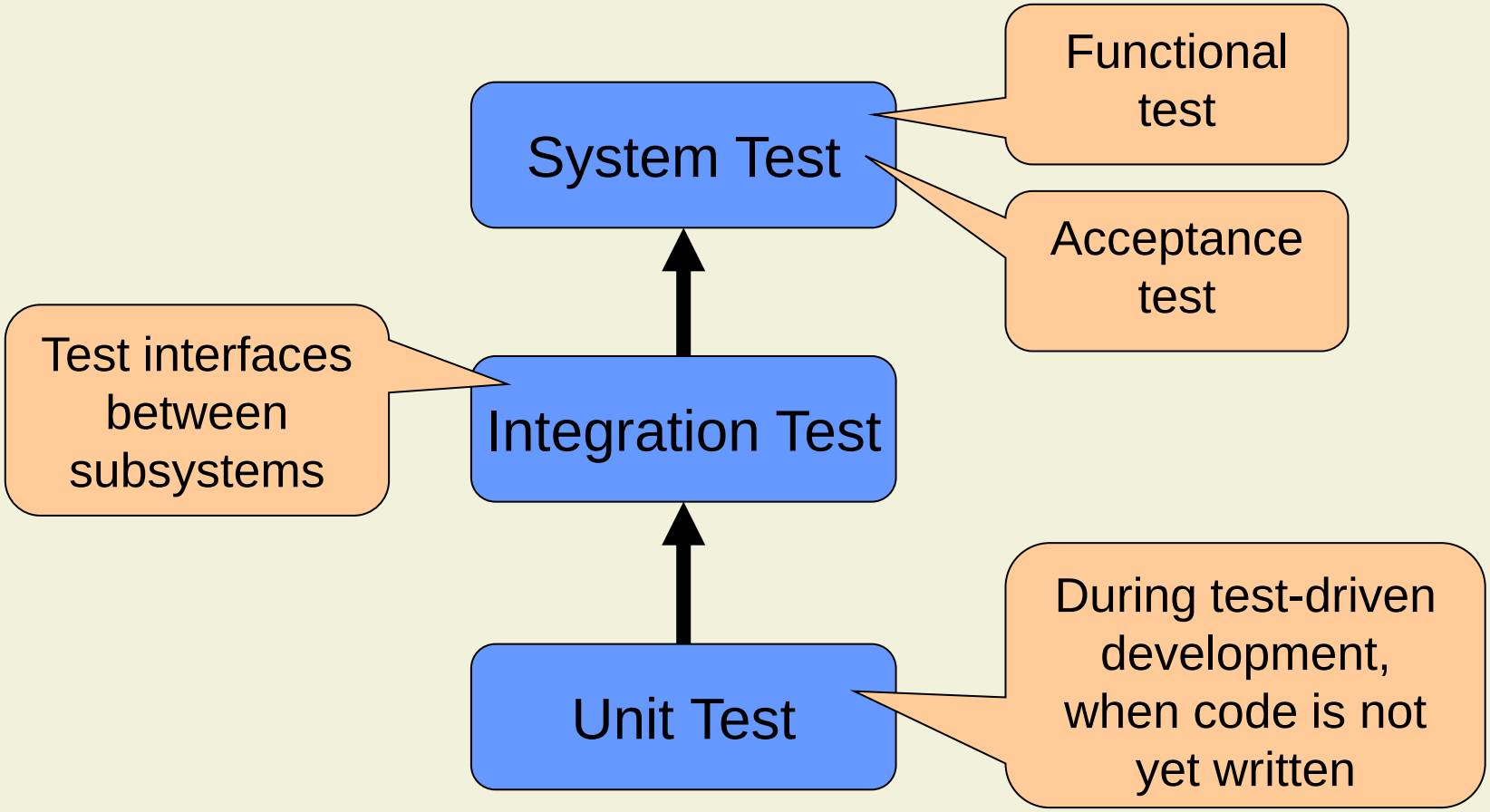
# 8. Test Case Selection

## 8.1 Functional Testing

## 8.2 Structural Testing

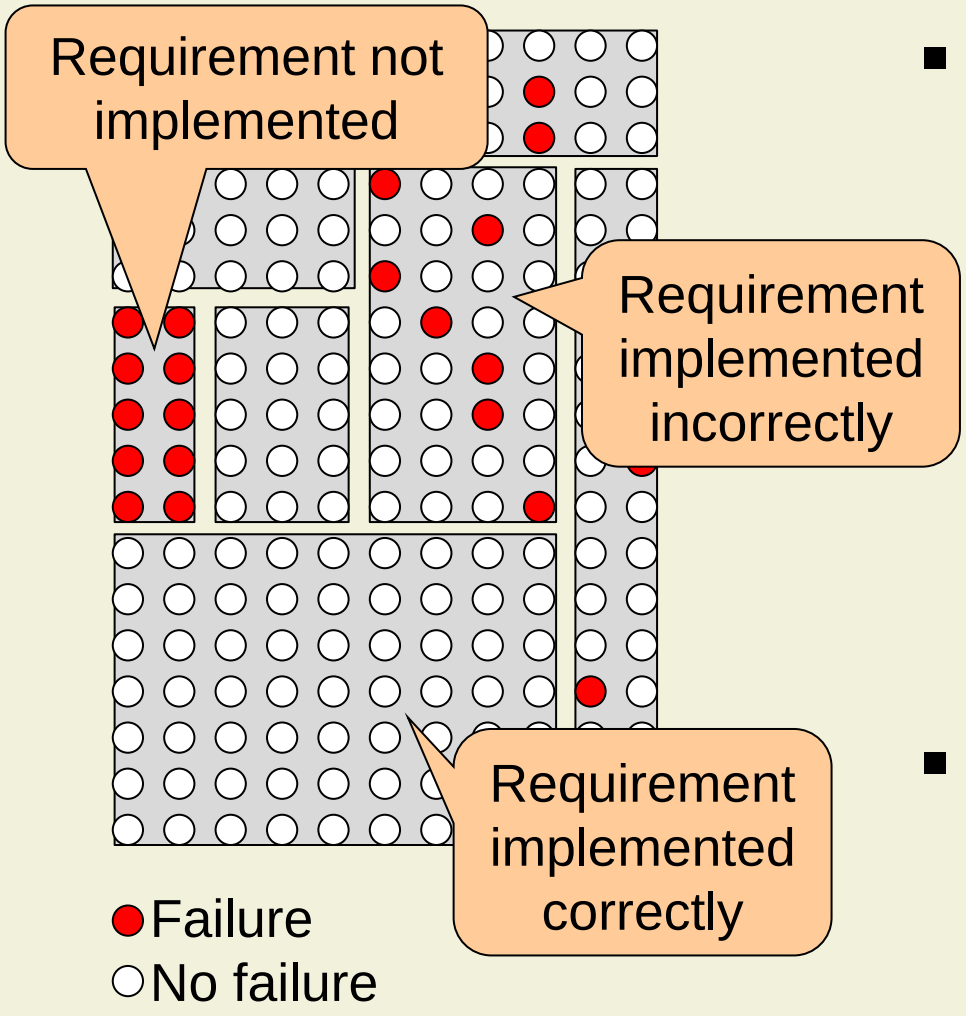
# Applications of Functional Testing

- Black-box test a unit against its requirements



# 8. Test Case Selection

# Finding Representative Inputs




- Divide inputs into **equivalence classes**
  - Each possible input belongs to one of the equivalence classes
  - Goal: some classes have higher density of failures
- Choose test cases for each equivalence class

# Equivalence Classes: Example

Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

| month | |
|---|---|
| Month with 28 or 29 days | month = 2 |
| Months with 30 days | month ∈ {4, 6, 9, 11} |
| Months with 31 days | month ∈ {1, 3, 5, 7, 8, 10, 12} |

| year | |
|---|---|
| Leap years | (year **mod** 4 = 0 and year **mod** 100 ≠ 0) or year **mod** 400 = 0 |
| Non-leap years | year **mod** 4 ≠ 0 or (year **mod** 100 = 0 and year **mod** 400 ≠ 0) |

Invalid inputs missing

# Equivalence Classes: Example (cont'd)

Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

| month | |
|---|---|
| Month with 28 or 29 days | month = 2 |
| Months with 30 days | month ∈ {4, 6, 9, 11} |
| Months with 31 days | month ∈ {1, 3, 5, 7, 8, 10, 12} |
| Invalid | month < 1 or month > 12 |

| year | |
|---|---|
| Leap years | (year **mod** 4 = 0 and year **mod** 100 ≠ 0) or year **mod** 400 = 0 |
| Non-leap years | year **mod** 4 ≠ 0 or (year **mod** 100 = 0 and year **mod** 400 ≠ 0) |

Partitioning seems too coarse

# Equivalence Classes: Example (cont'd)

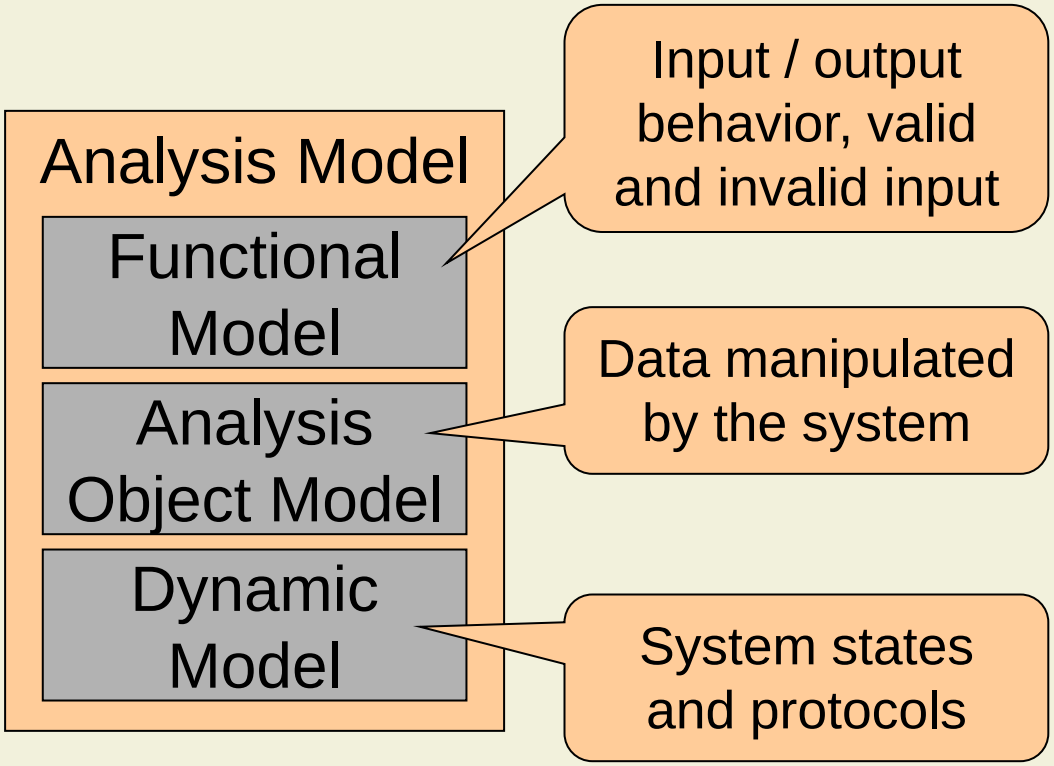
Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

| month | |
|---|---|
| Month with 28 or 29 days | month = 2 |
| Months with 30 days | month ∈ {4, 6, 9, 11} |
| Months with 31 days | month ∈ {1, 3, 5, 7, 8, 10, 12} |
| Invalid | month < 1 or month > 12 |

| year | |
|---|---|
| Standard leap years | year **mod** 4 = 0 and year **mod** 100 ≠ 0 |
| Standard non-leap years | year **mod** 4 ≠ 0 |
| Special leap years | year **mod** 400 = 0 |
| Special non-leap years | year **mod** 100 = 0 and year **mod** 400 ≠ 0 |

# Sources of Information

- Use **analysis knowledge** to determine test cases that check requirements



# Using the Functional Model

- The functional model describes the input-output behavior of the whole system
  - Valid and invalid inputs, entry conditions
  - Expected results, exit conditions

- Basis for functional system testing

# Using the Functional Model: Example

## Actor steps

1. Authenticate **(use case Authenticate)**
3. Client selects “Withdraw CHF”
5. Client enters amount

Inputs

Expected outputs

## System Steps

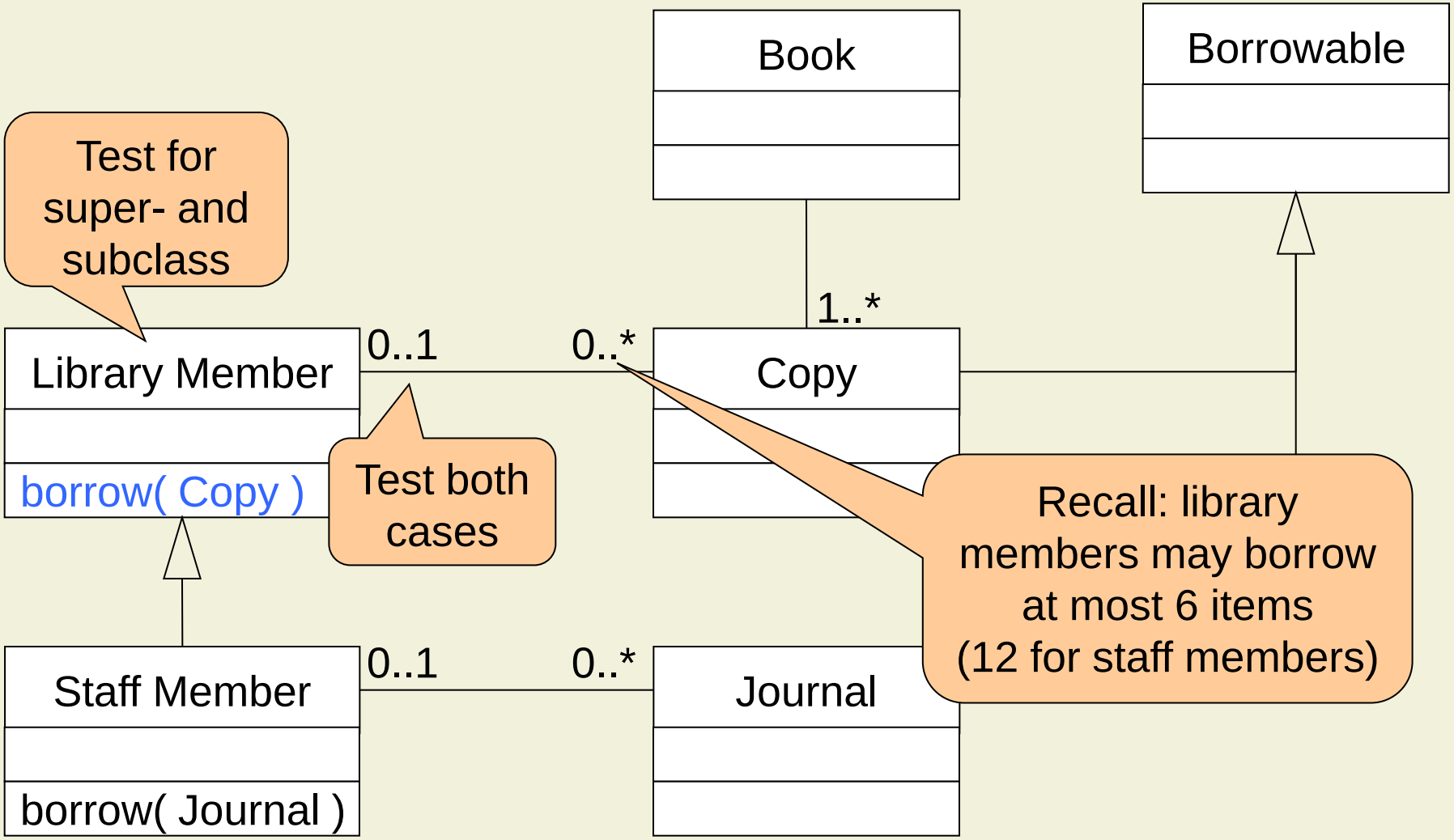
2. Bankomat displays options
4. Bankomat queries amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF

# Using the Analysis Object Model

- The analysis object model contains the main concepts manipulated by the system, their properties and relationships
  - Useful to determine equivalence classes
  - Useful to set up state of objects

- Relevant information
  - Classes and attributes
  - Subtypes
  - Aggregations and multiplicities

# Using the Analysis Object Model: Example



# Equivalence Classes

- Consider the operation member.borrow( copy )

| member | |
|---|---|
| Library member | 0 – 5 borrowed copies |
| Staff member | 0 – 11 borrowed items |
| Invalid library member | more than 5 borrowed copies |
| Invalid staff member | more than 11 borrowed items |

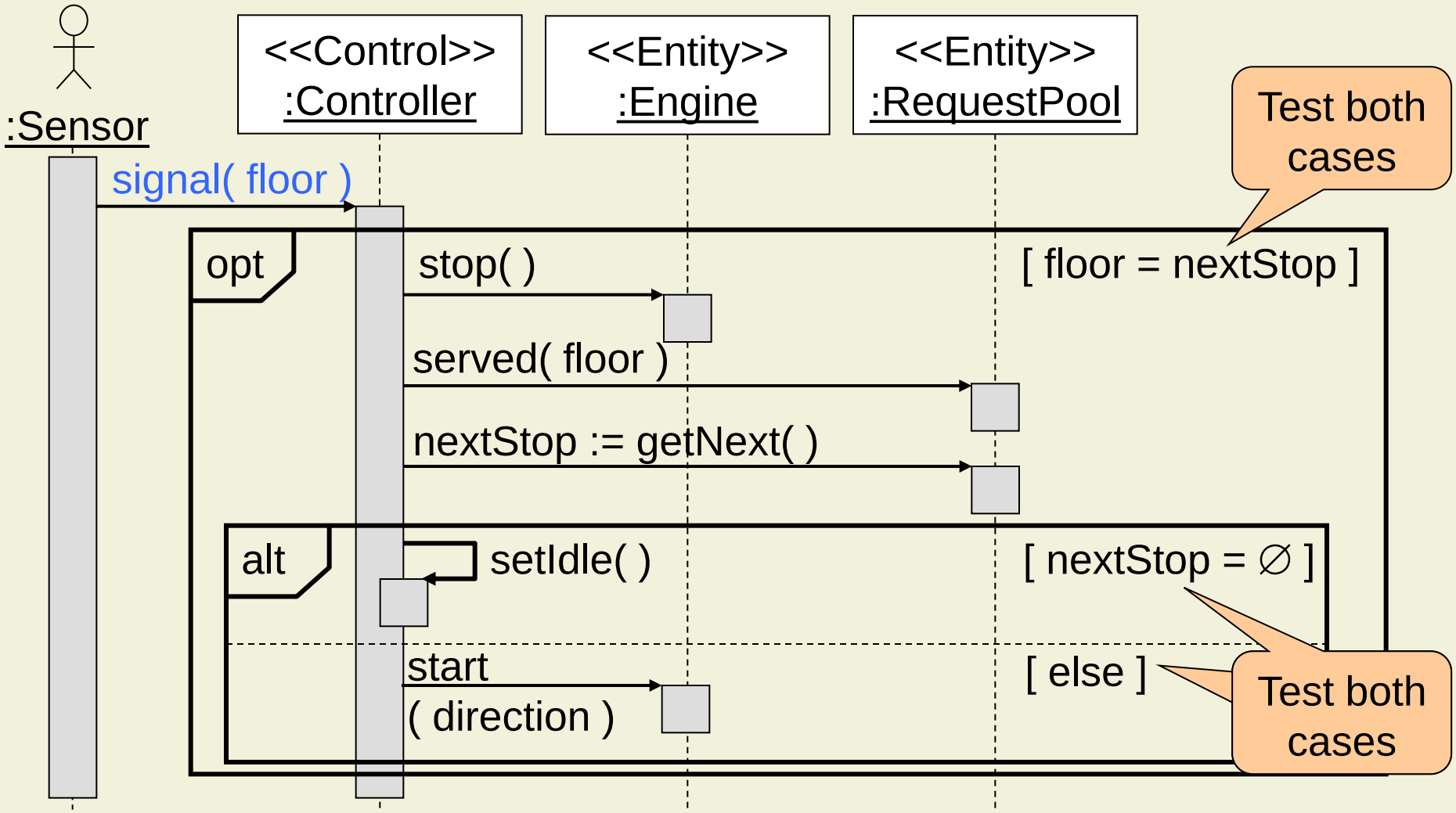
| copy | |
|---|---|
| Copy on shelf | copy has zero library members |
| Copy borrowed by library member | copy has one library member |
| Copy borrowed by staff member | copy has one staff member |
| Invalid | **null** |

# Using the Dynamic Model

- Sequence diagrams describe protocols for object interactions
  - Benefit for testing is similar to use cases
  - Especially useful for integration testing

- State diagrams describe state-dependent behavior
  - Different states typically require different equivalence classes
  - State defines valid input and expected output
  - Expected output includes successor state
  - Useful for protocols, GUIs, and objects

# Using the Dynamic Model: Example 1

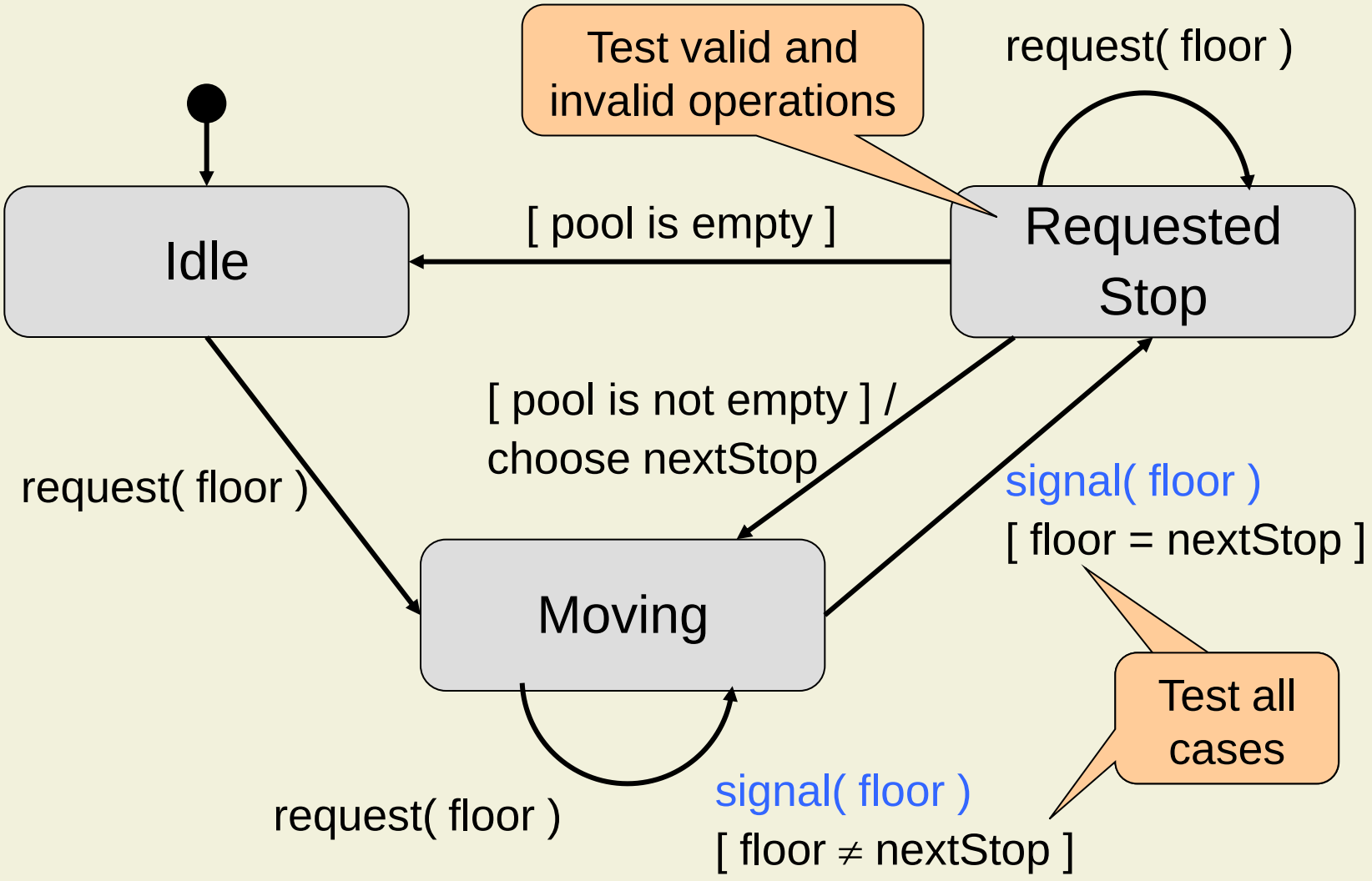


# Equivalence Classes

- Consider the operation controller.signal( floor )

| controller | |
|---|---|
| Serve only request | floor = nextStop, and there is no further request |
| Serve first request | floor = nextStop, and there are further requests |
| Serve no request | floor ≠ nextStop |

# Using the Dynamic Model: Example 2



# Equivalence Classes

- Consider the operation controller.signal( floor )

| controller | |
|---|---|
| Reach floor | controller is in state Moving and floor = nextStop |
| Keep moving | controller is in state Moving and floor ≠ nextStop |
| Invalid state | controller is in state Idle or RequestedStop |

# 8. Test Case Selection

## 8.1 Functional Testing

### 8.1.1 Partition Testing

### 8.1.2 Selecting Representative Values

### 8.1.3 Combinatorial Testing

## 8.2 Structural Testing

# Selecting Representative Values

- Once we have partitioned the input values, we need to select **concrete values** for the test cases **for each equivalence class**

- Input from a range of valid values
  - Below, within, and above the range
  - Also applies to multiplicities on aggregations

- Input from a discrete set of valid values
  - Valid and invalid discrete value
  - Instances of each subclass

# Boundary Testing

Given an integer x, determine the absolute value of x

| x | |
|---|---|
| Valid | all values |

```
int abs( int x ) {
  if( 0 <= x ) return x;
  return –x;
}
```

Negative result for x==Integer.MIN_VALUE

- Large number of errors tend to occur at **boundaries of the input domain**
  - Overflows
  - Comparisons (‘<‘ instead of ‘<=‘, etc.)
  - Missing emptiness checks (e.g., collections)
  - Wrong number of iterations

# Boundary Testing: Example

- Select elements at the “edge” of each equivalence class (in addition to values in the middle)
  - Ranges: lower and upper limit
  - Empty sets and collections

| month | |
|---|---|
| Month with 28 or 29 days | month = 2 |
| Months with 30 days | month ∈ {4, 6, 9, 11} |
| Months with 31 days | month ∈ {1, 3, 5, 7, 8, 10, 12} |
| Invalid | month < 1 or month > 12 |

There is only one value

Choose all values

Choose 1 and 12 plus one more

Choose MIN_VALUE, 0, 13, MAX_VALUE

# Boundary Testing: Example (cont'd)

| year | |
|---|---|
| Standard leap years | year **mod** 4 = 0 and year **mod** 100 ≠ 0 |
| Standard non-leap years | year **mod** 4 ≠ 0 |
| Special leap years | year **mod** 400 = 0 |
| Special non-leap years | year **mod** 100 = 0 and year **mod** 400 ≠ 0 |

Choose for instance -200.004, -4, 4, 2012, 400.008

Choose for instance -200.003, -1, 1, 2011, 400.009

Choose for instance -200.000, 0, 2000, 400.000

Choose for instance -200.100, 1900, 400.100

# Parameterized Unit Test for Leap Years

```
[ Test ]
public void TestDemo29(
    [ Values( -200004, -200000, -4, 0,4, 2000, 2012, 400000, 400008 ) ]
    int year )
{
  int d = Days( 2, year );
  Assert.IsTrue( d == 29 );
}
```

Only one value

All selected values for leap years and special leap years

Expected result

- Analogous test cases for February in non-leap year, months with 30 days, and months with 31 days

# Parameterized Unit Test for Invalid Inputs

Expected result: an exception

All selected invalid values for month

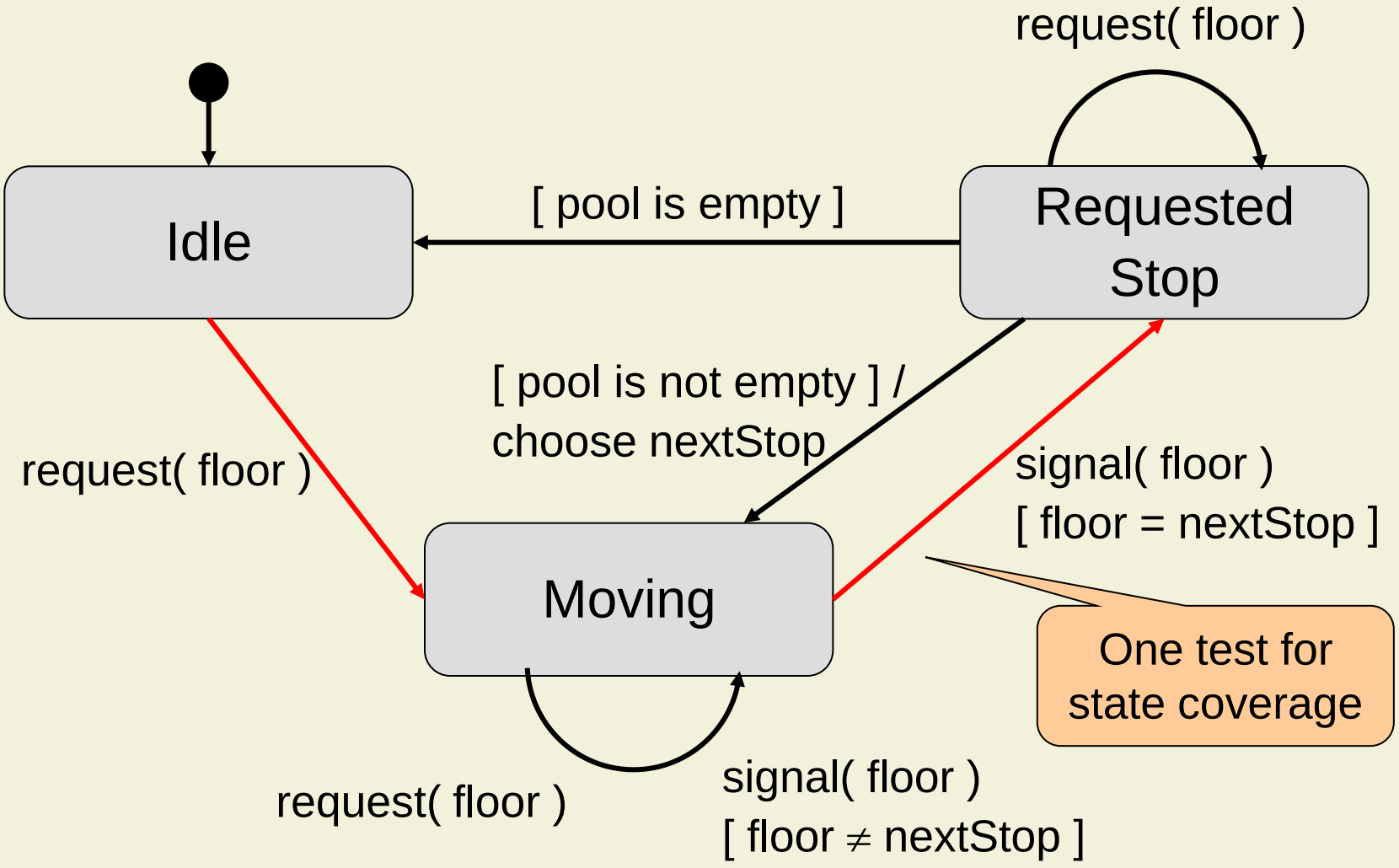
All selected values for year

```
[ Test ]
[ ExpectedException( typeof(ArgumentException) ) ]
public void TestDemoInvalid(
    [ Values( int.MinValue, 0, 13, int.MaxValue ) ] int month,
    [ Values( -200100, -200004, -200003, -200000, -4, -1, 0, 1, 4, 1900,
       2000, 2011, 2012, 400000, 400008, 400009, 400100 ) ] int year ) {
  int d = Days( month, year );
}
```

# Coverage of State Diagrams

- We define coverage criteria to measure how thorough a state diagram is tested
- Path Coverage: execute each possible path
  - Not feasible with many nested conditionals
  - Impossible for most loops
- State Coverage: visit each state
  - A minimum criterion
- Transition Coverage: execute each edge
  - Thorough testing

# Coverage Example



# Coverage Example (cont'd)

- Consider the operation controller.signal( floor )

| controller | |
|---|---|
| Reach floor | controller is in state Moving and floor = nextStop |
| Keep moving | controller is in state Moving and floor ≠ nextStop |
| Invalid state | controller is in state Idle or RequestedStop |

One test case for state coverage

Two test cases for transition coverage: empty and non-empty pool

Not required for state coverage

Not required for coverage

# 8. Test Case Selection

## 8.1 Functional Testing

### 8.1.1 Partition Testing

### 8.1.2 Selecting Representative Values

### 8.1.3 Combinatorial Testing

## 8.2 Structural Testing

# Combinatorial Testing

- Combining equivalence classes and boundary testing leads to many values for each input
  - Twelve values for month and 17 values for year in the Leap Year example
- Testing all possible combinations leads to a combinatorial explosion (12 x 17 = 204 tests)

- Reduce test cases to make effort feasible
  - Semantic constraints
  - Combinatorial selection
  - Random selection

# Eliminating Combinations

- Inspect test cases for unnecessary combinations
  - Especially for invalid values
  - Use problem domain knowledge

| month | |
|---|---|
| Month with 28 or 29 days | month = 2 |
| Months with 30 days | month ∈ {4, 6, 9, 11} |
| Months with 31 days | month ∈ {1, 3, 5, 7, 8, 10, 12} |
| Invalid | month < 1 or month > 12 |

Test all combinations with year

Behavior is independent of year

- Reduces test cases from 204 to 17 + 4 + 3 + 4 = 28

# Eliminating Combinations: NUnit Example

```
[ Test, Sequential ]
[ ExpectedException( typeof(ArgumentException) ) ]
public void TestDemoInvalid(
    [ Values( int.MinValue, 0, 13, int.MaxValue ) ] int month,
    [ Values( -200100, -200004, -200003, -200000 ) ] int year ) {
  int d = Days( month, year );
}
```

All selected invalid values for month

One value for year for each value for month

# Selecting Object References

- Objects are different from values because they have identity

```
a1 = new Account( 1000 );
a2 = new Account( 1000 );
a1.transfer( a2, 500 );
```

```
a1 = new Account( 1000 );
a1.transfer( a1, 500 );
```

Might behave differently (e.g., deadlock)

- When selecting test data for objects, one has to consider object identities and aliasing
- Referenced objects lead to combination problem

# Selecting Object References: Example

| member | |
|---|---|
| Library member | 0 – 5 borrowed copies |
| Staff member | 0 – 11 borrowed items |
| Invalid library member | more than 5 borrowed copies |
| Invalid staff member | more than 11 borrowed items |

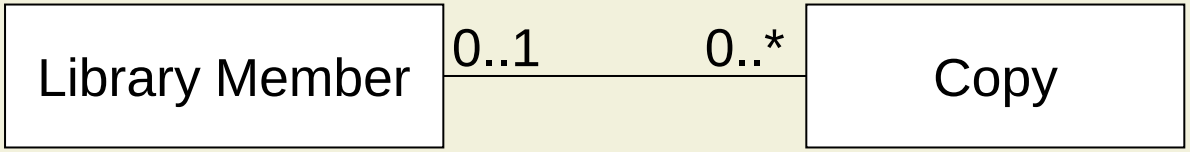
Empty list,
list of 1 and 5 (11) copies,
list without duplicates,
list with duplicates

List of 6 (12) copies and very large collection

- This is a case of combinatorial testing since it combines the Library Member and the collection

# Semantic Constraints for Objects

- Object invariants restrict the possible instances of a class diagram
  - Expressed as comment or OCL constraint
- In our example, assume the following invariants




  - A Library Member m contains a Copy c in its collection of borrowed items if and only if c's Library Member is m
  - For each Library Member, the collection of borrowed items contains no duplicates
  - For each Library Member, the collection of borrowed items contains at most 6 copies (12 items for staff)

# Selecting Object References: Example (cont'd)

Empty list,
list of 1 and 5 (11) copies
without duplicates

| member | | copy | |
|---|---|---|---|
| Library member | 0 – 5 borrowed copies | Copy on shelf | copy has zero library members |
| Staff member | 0 – 11 borrowed items | Copy borrowed by library member | copy has one library member |
| Invalid library member | more than 5 borrowed copies | Copy borrowed by staff member | copy has one staff member |
| Invalid staff member | more than 11 borrowed items | | **null** |

List of 6 (12) copies

copy is borrowed by «member»,
copy is not borrowed by «member»

# Roots Example

Given three values, a, b, c, compute all solutions of the equation $ax^2 + bx + c = 0$

| | a | b | c |
|---|---|---|---|
| Valid | any value | any value | any value |
| Invalid | infinity, NaN | infinity, NaN | infinity, NaN |

Boundary testing:
a, b, c ∈
{ Double.MIN_VALUE, -5, 0, 5, Double.MAX_VALUE }

- $5^3 = 125$ test cases for valid inputs

# Roots Example (cont'd)

Given three values, a, b, c, compute all solutions of the equation $ax^2 + bx + c = 0$

Look at dependencies between inputs

| Two solutions | One solution | No solution |
|---|---|---|
| $a \neq 0$ and $b^2-4ac > 0$ | $a = 0$ and $b \neq 0$<br>or<br>$a \neq 0$ and $b^2-4ac = 0$ | $a = 0$, $b = 0$, and $c \neq 0$<br>or<br>$a \neq 0$ and $b^2-4ac < 0$ |

Semantic constraints on combinations

Partitioning seems too coarse

# Roots Example (cont'd)

Given three values, a, b, c, compute all solutions of the equation $ax^2 + bx + c = 0$

| | Two solutions | One solution | No solution |
|---|---|---|---|
| **Linear equation** | | $a = 0$ and $b \neq 0$ | $a = 0$, $b = 0$, and $c \neq 0$ |
| **(Truly) quadratic equation** | $a \neq 0$ and $b^2-4ac > 0$ | $a \neq 0$ and $b^2-4ac = 0$ | $a \neq 0$ and $b^2-4ac < 0$ |

Not all inputs are covered: a=b=c=0

# Roots Example (cont'd)

Given three values, a, b, c, compute all solutions of the equation $ax^2 + bx + c = 0$; report an error if all three values are zero

| | Two solutions | One solution | No solution |
|---|---|---|---|
| **Linear equation** | | $a = 0$ and $b \neq 0$ | $a = 0$, $b = 0$, and $c \neq 0$ |
| **(Truly) quadratic equation** | $a \neq 0$ and $b^2-4ac > 0$ | $a \neq 0$ and $b^2-4ac = 0$ | $a \neq 0$ and $b^2-4ac < 0$ |
| **Invalid input** | $a = 0$, $b = 0$, $c = 0$ | | |

# Roots Example: Summary

- Classifying the combinations according to semantic constraints did not reveal any irrelevant test cases
- But we did identify an omission in the specification
  - It is common that testers clarify the specification
- One option is to manually choose a manageable number of test cases such that there is at least one test case for each semantic constraint
  - Note that omitting test cases might leave errors such as arithmetic overflow undetected

# Semantic Constraints: Discussion

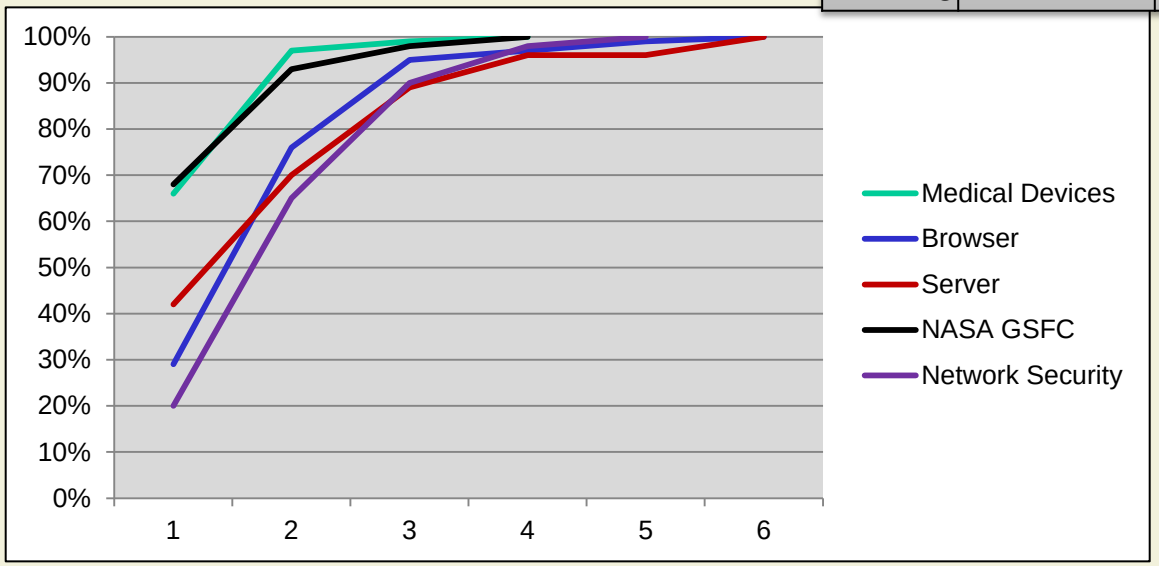
- Semantic constraints potentially reduce the number of test cases
  - They also help increasing the coverage

- But too many combinations remain
  - Especially when there are many input values, for instance, for the fields of objects

# Influence of Variable Interactions

- Empirical evidence suggests that most errors do not depend on the interaction of many variables

| Vars | Medical Devices | Browser | Server | NASA GSFC | Network Security |
|---|---|---|---|---|---|
| 1 | 66% | 29% | 42% | 68% | 20% |
| 2 | 97% | 76% | 70% | 93% | 65% |
| 3 | 99% | 95% | 89% | 98% | 90% |
| 4 | 100% | 97% | 96% | 100% | 98% |
| 5 | | 99% | 96% | | 100% |
| 6 | | 100% | 100% | | |




- Interactions of two or three variables trigger most errors

# Pairwise-Combinations Testing

- Instead of testing all possible combinations of all inputs, focus on **all possible combinations of each pair of inputs**
  - Pairwise-combinations testing is identical to combinatorial testing for two or less inputs
- Example: Consider a method with four boolean parameters
  - Combinatorial testing requires $2^4 = 16$ test cases
  - Pairwise-combinations testing requires 5 test cases: TTTT, TFFF, FTFF, FFTF, FFFT
- Can be generalized to k-tuples (k-way testing)

# Pairwise-Combinations Testing: Complexity

- For n parameters with d values per parameter, the number of test cases grows logarithmically in n and quadratic in d
  - Handles larger number of parameters, for instance, fields of objects
  - The number d can be influenced by the tester
- Result holds for large n and d, and for all k in k-way testing

# Pairwise-Combinations Testing: Example

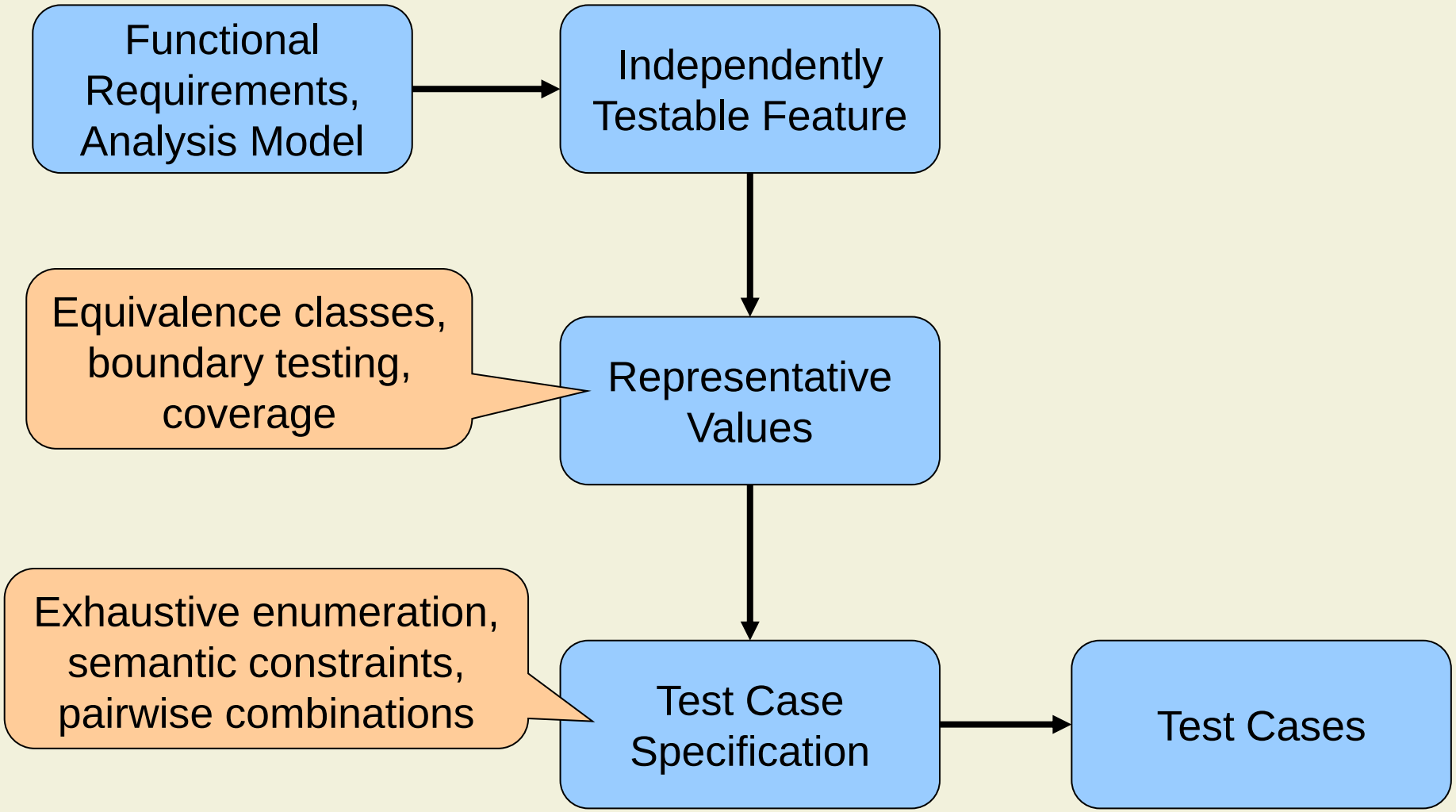
| Two solutions | One solution | No solution |
|---|---|---|
| | a = 0 and b ≠ 0 | a = 0, b = 0, and c ≠ 0 |
| a ≠ 0 and $b^2$-4ac > 0 | a ≠ 0 and $b^2$-4ac = 0 | a ≠ 0 and $b^2$-4ac < 0 |
| a = 0, b = 0, c = 0 | | |

- Three parameters, five values each
  - Double.MIN_VALUE, -5, 0, 5, Double.MAX_VALUE
  - $5^3$ = 125 test cases for combinatorial testing
  - 25 test cases for pairwise-combinations testing
- Bug is still detected (depends only on a and b)
- Some cases depend on three parameters, e.g., invalid input

# Pairwise-Combinations Testing: Discussion

- Pairwise-combinations testing (or k-way testing) **reduces the number of test cases significantly** while detecting most errors
- Pairwise-combinations testing is especially important when **many system configurations** need to be tested
  - Hardware, operating system, database, application server, etc.
- Should be **combined with other approaches** to detect errors that are triggered by more complex interactions among parameters

# Functional Testing: Summary



# 8. Test Case Selection

8.1 Functional Testing

**8.2 Structural Testing**

# Motivating Example

Given a non-null array of integers, sort the array in-place in ascending order

```
public void sort( int[ ] a ) {
  if( a == null || a.length < 2 ) // array is trivially sorted
    return;
  // check if array is already sorted
  for( int i = 0; i < a.length – 1; i++ )
    if( a[ i ] < a[ i + 1 ] )
      break;
  if( i >= a.length – 1 ) // array is already sorted
    return;
  // use quicksort to sort the array in ascending order
}
```

Error: check for sortedness should use '>'

# Motivating Example: Functional Testing

Given a non-null array of integers, sort the array in-place in ascending order

| | a |
|---|---|
| Valid | any non-null array |
| Invalid | **null** |

Choose for instance
{ }, { 1 }, { 1, 2, 3 }

- The requirements give no clue that one should test with an array that is sorted in descending order

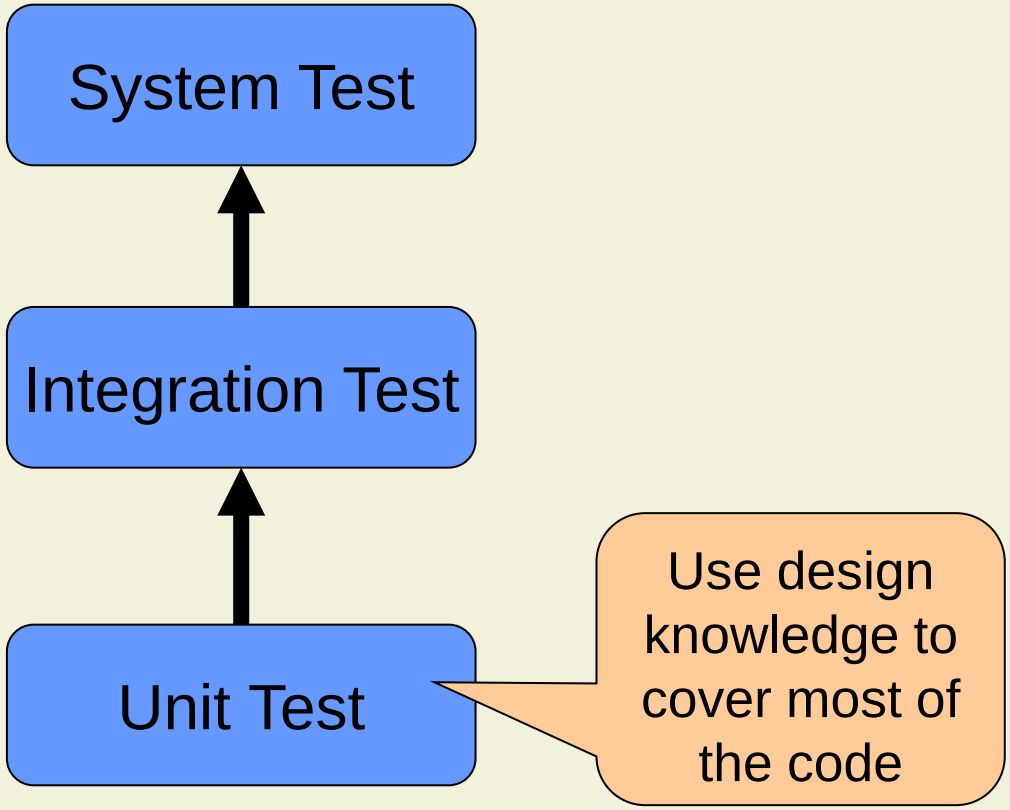
# Motivating Example: Discussion

- Detailed design and coding introduce many behaviors that are not present in the requirements
  - Choice of data structures
  - Choice of algorithms
  - Optimizations such as caches

- Functional testing generally does not thoroughly exercise these behaviors
  - No data structure specific test cases, e.g., rotation of AVL-tree
  - No test cases for optimizations, e.g., cache misses

# Applications of Structural Testing

- White-box test a unit to cover a large portion of its code



# 8. Test Case Selection

# Basic Blocks

- A **basic block** is a sequence of statements such that the code in a basic block:
  - has **one entry point**: no code within it is the destination of a jump instruction anywhere in the program
  - has **one exit point**: only the last instruction causes the program to execute code in a different basic block

- Whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order

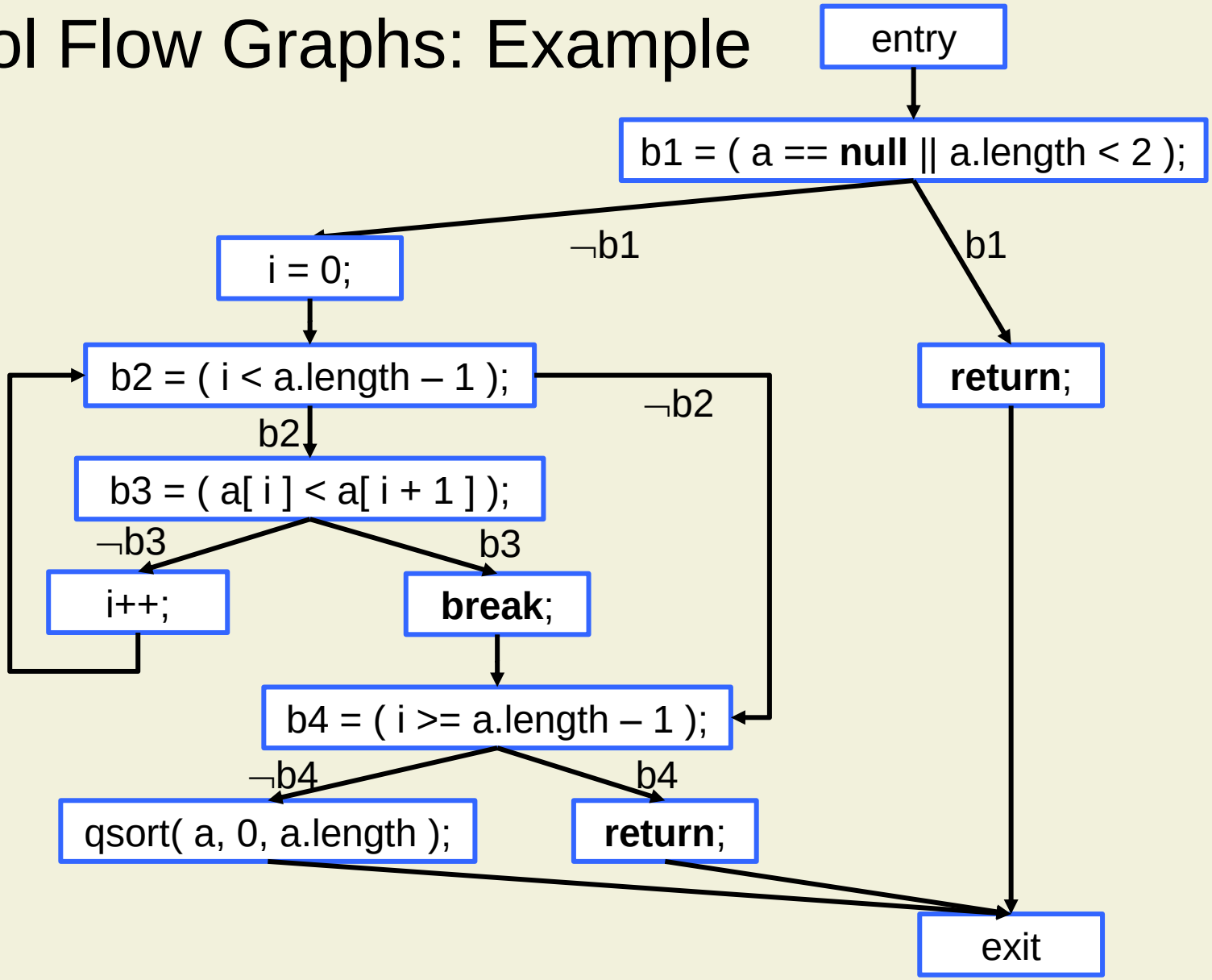
# Basic Blocks: Example

```
public void sort( int[ ] a ) {
  if( a == null || a.length < 2 )
    return;
  for( int i = 0; i < a.length – 1; i++ ) {
    if( a[ i ] < a[ i + 1 ] )
      break;
  }
  if( i >= a.length – 1 )
    return;
  qsort( a, 0, a.length );
}
```

# Intraprocedural Control Flow Graphs

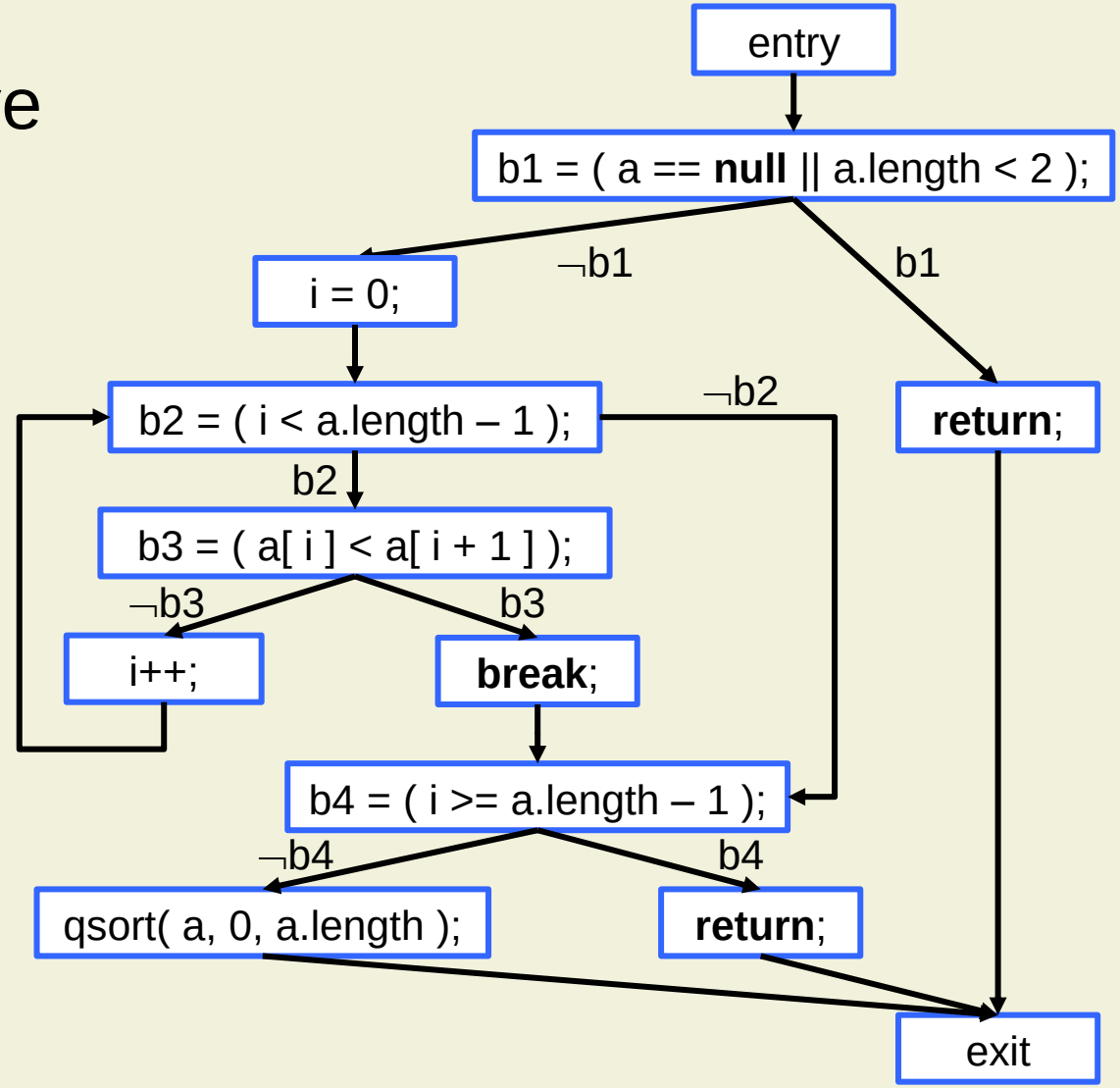
- An **intraprocedural control flow graph** (CFG) of a procedure p is a graph (N,E) where:
- N is the set of basic blocks in p plus designated entry and exit blocks
- E contains
  - an edge from a to b with condition c iff the execution of basic block a is succeeded by the excution of basic block b if condition c holds
  - an edge (entry, a, true) if a is the first basic block of p
  - edges (b, exit, true) for each basic block b that ends with an (implicit) return statement

# Control Flow Graphs: Example



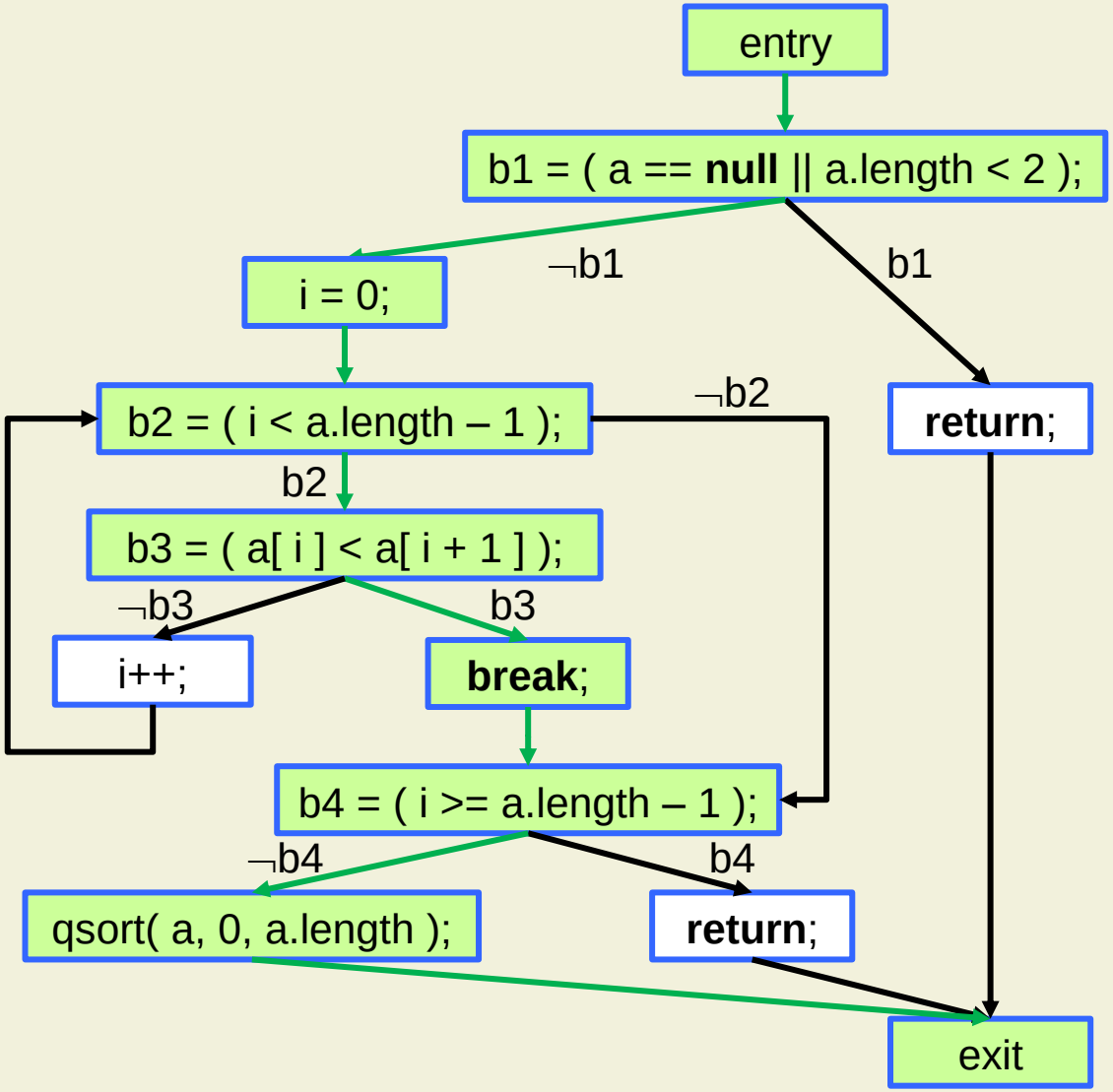
# Test Coverage

- The CFG can serve as an **adequacy criterion** for test cases
- The more parts are executed, the higher the chance to uncover a bug
- "parts" can be nodes, edges, paths, etc.



# Test Coverage: Example

- Consider the input
  a = { 3, 7, 5 }



# Statement Coverage

- Assess the quality of a test suite by measuring how much of the CFG it executes

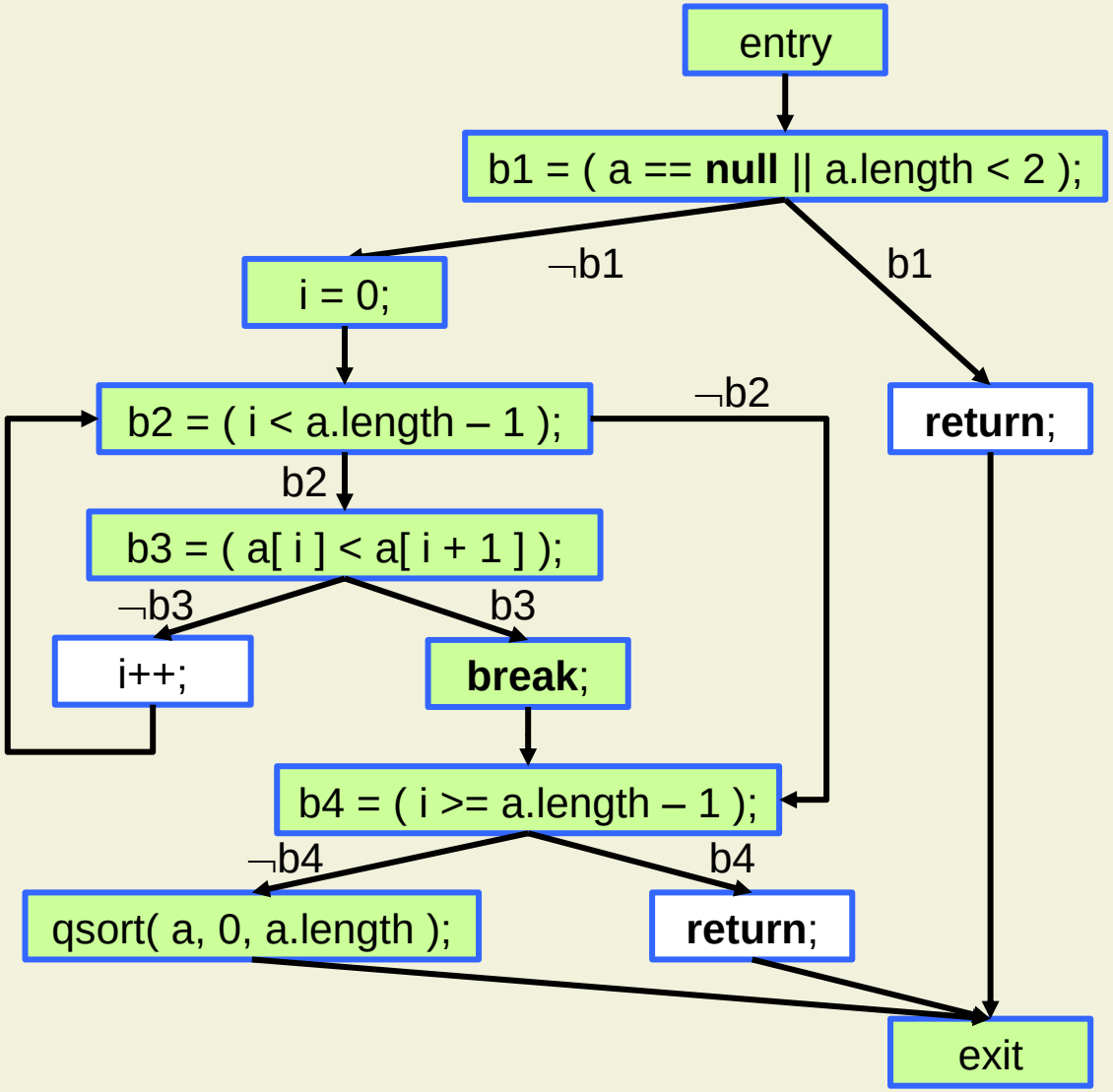
- Idea: one can detect a bug in a statement only by executing the statement

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

  - Can also be defined on basic blocks

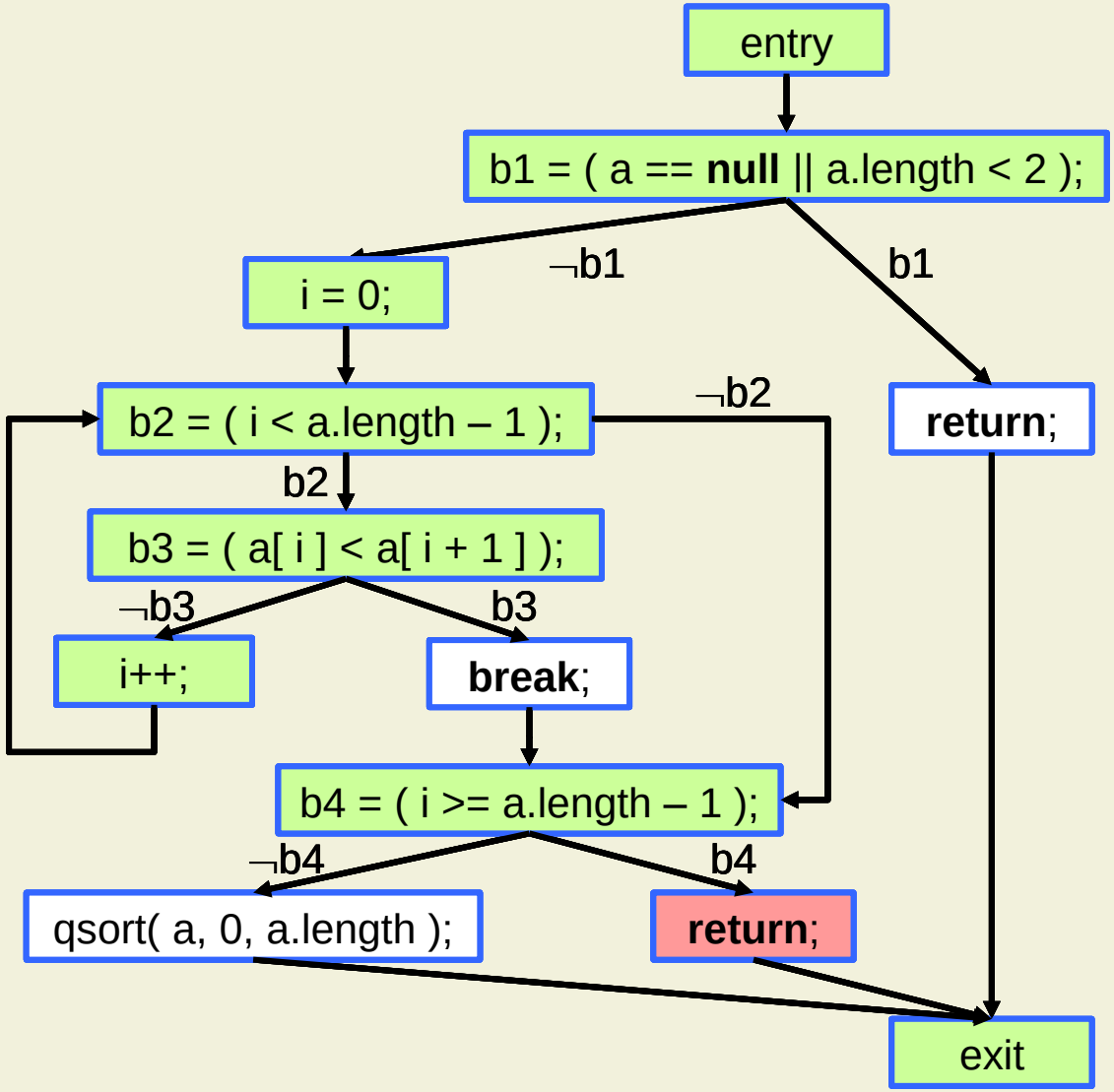
# Statement Coverage: Example

- Consider the input a = { 3, 7, 5 }
- This single test case executes 7 out of 10 basic blocks
- Statement coverage: 70%



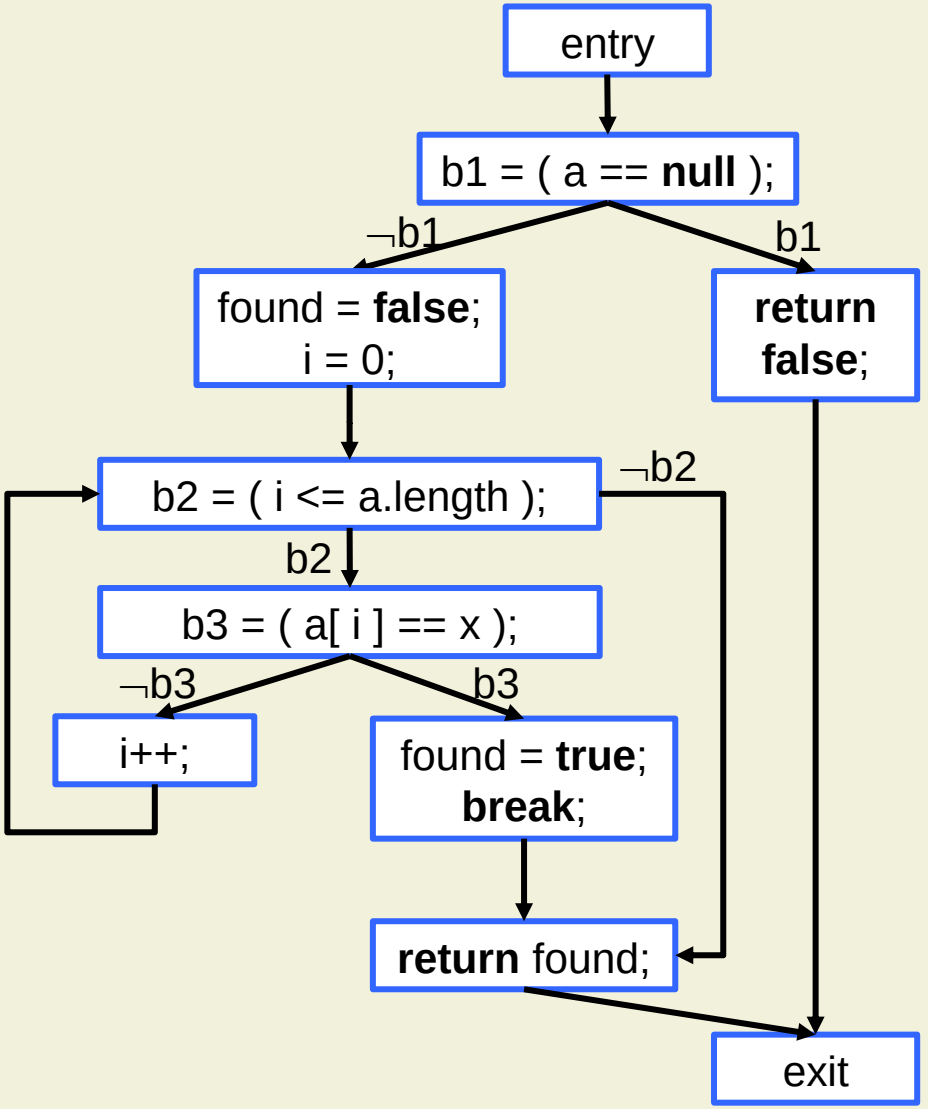
# Statement Coverage: Example (cont'd)

- We can achieve 100% statement coverage with three test cases
  - a = { 1 }
  - a = { 5, 7 }
  - a = { 7, 5 }
- The last test case detects the bug



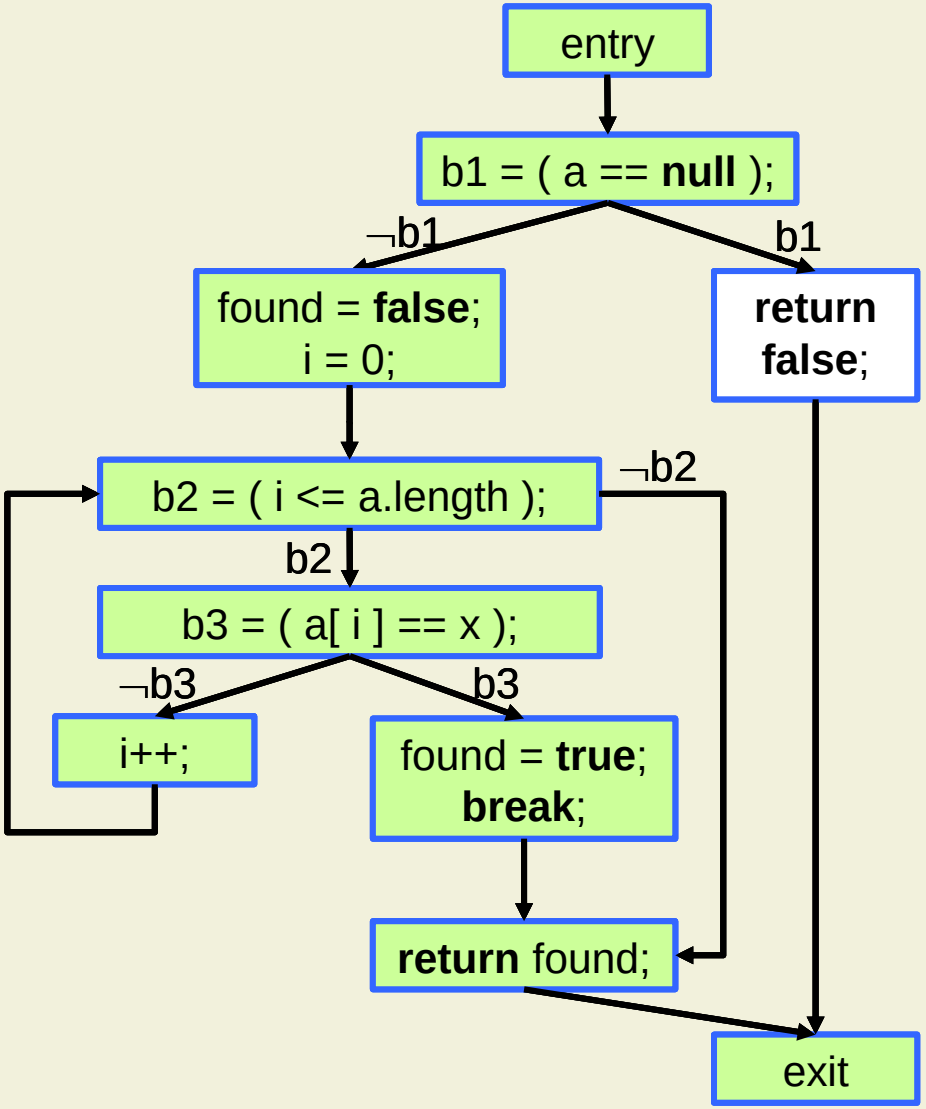
# Statement Coverage: Discussion

```
boolean contains( int[ ] a, int x ) {
  if( a == null )  return false;
  boolean found = false;
  for( int i = 0; i <= a.length; i++ ) {
    if( a[ i ] == x ) {
      found = true;
      break;
    }
  }
  return found;
}
```



# Statement Coverage: Discussion (cont'd)

- We can achieve 100% statement coverage with two test cases
  - a = **null**
  - a = { 1, 2 }, x = 2
- The test cases do not detect the bug!
- More thorough testing is necessary



# Branch Coverage

- Idea: test all possible branches in the control flow

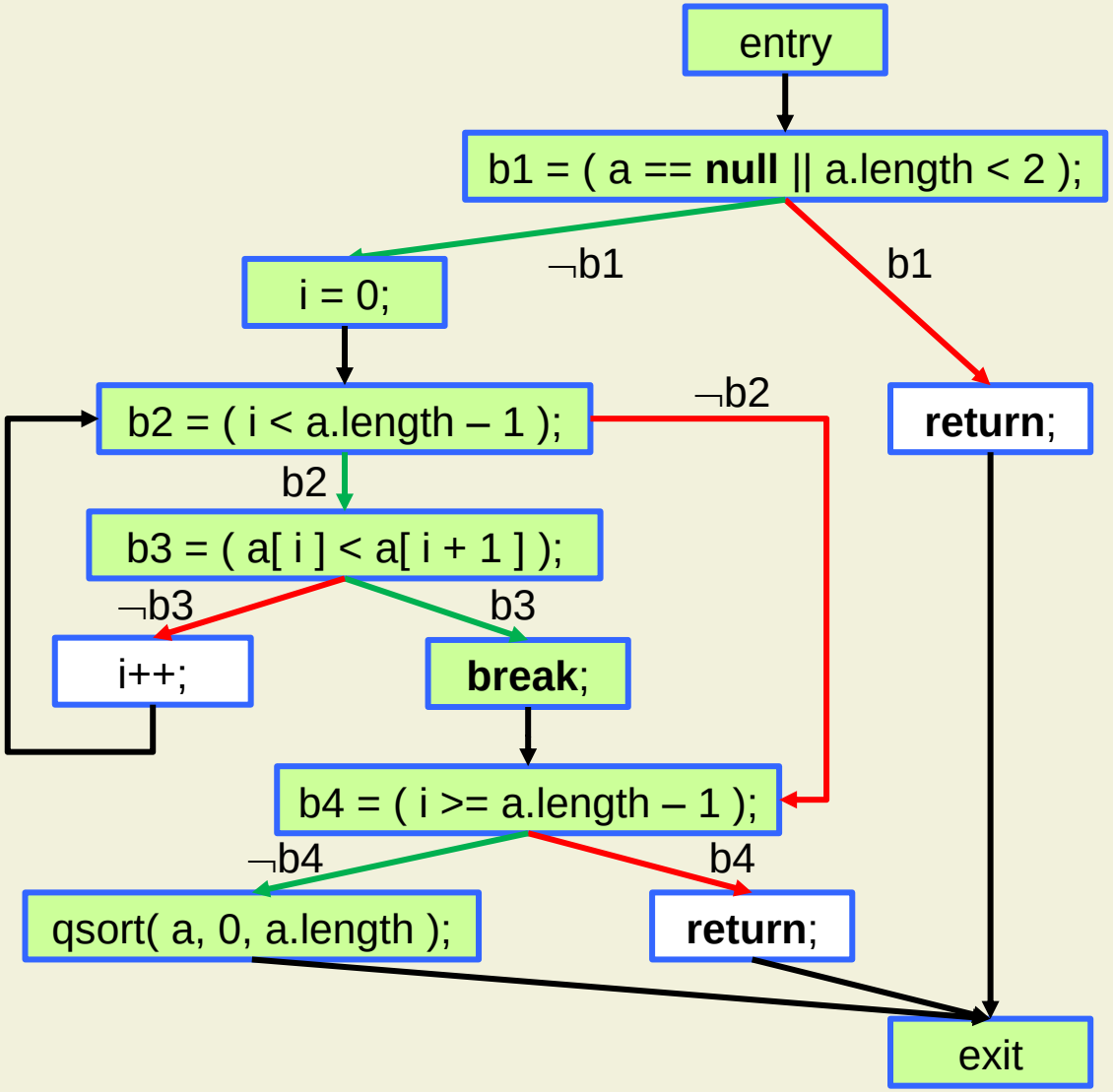
- An edge (m, n, c) in a CFG is a branch iff there is another edge (m, n', c') in the CFG with n ≠ n'

$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

  - Conveniently define branch coverage to be 100% if the code contains no branches

# Branch Coverage: Example 1

- Consider the input a = { 3, 7, 5 }
- This single test case executes 4 out of 8 branches
- Branch coverage: 50%
- Three test cases needed for 100% branch coverage

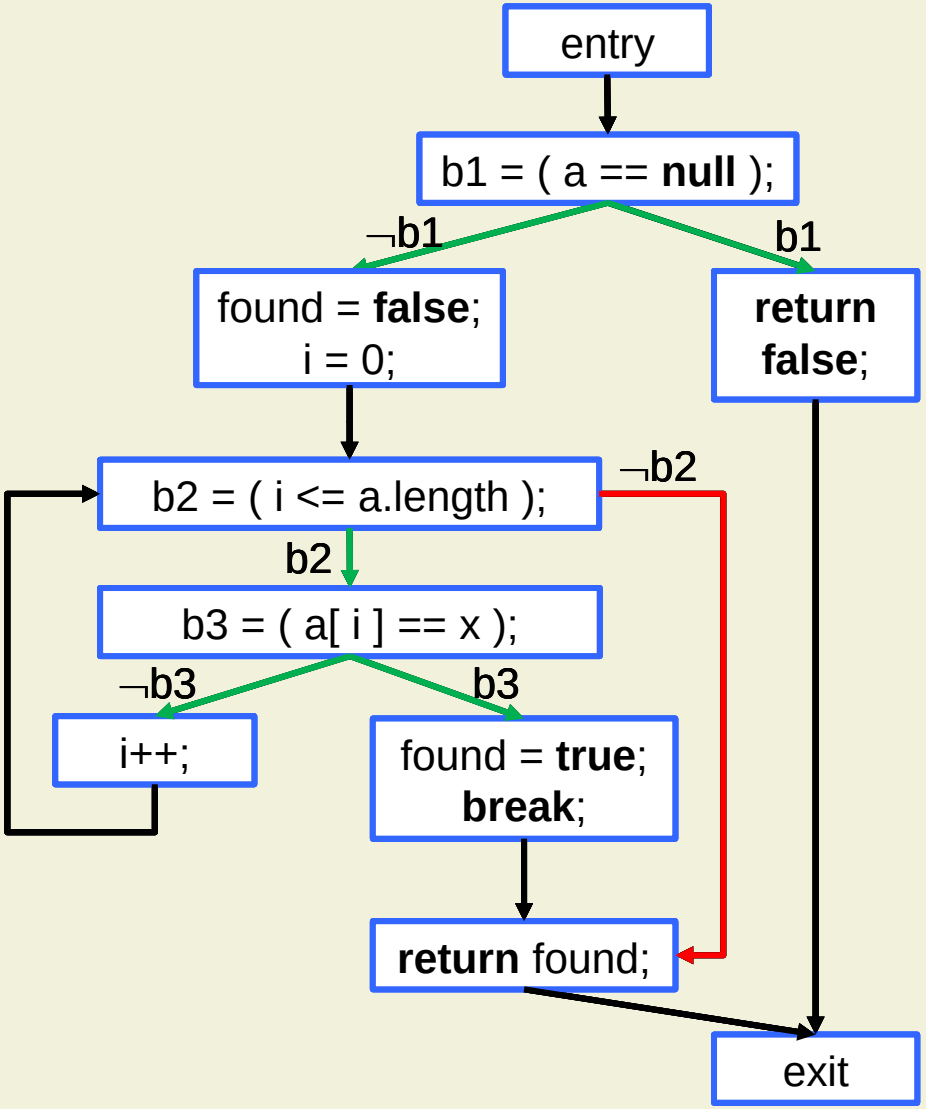


# Branch Coverage: Example 2

- The two test cases
  - a = **null**
  - a = { 1, 2 }, x = 2

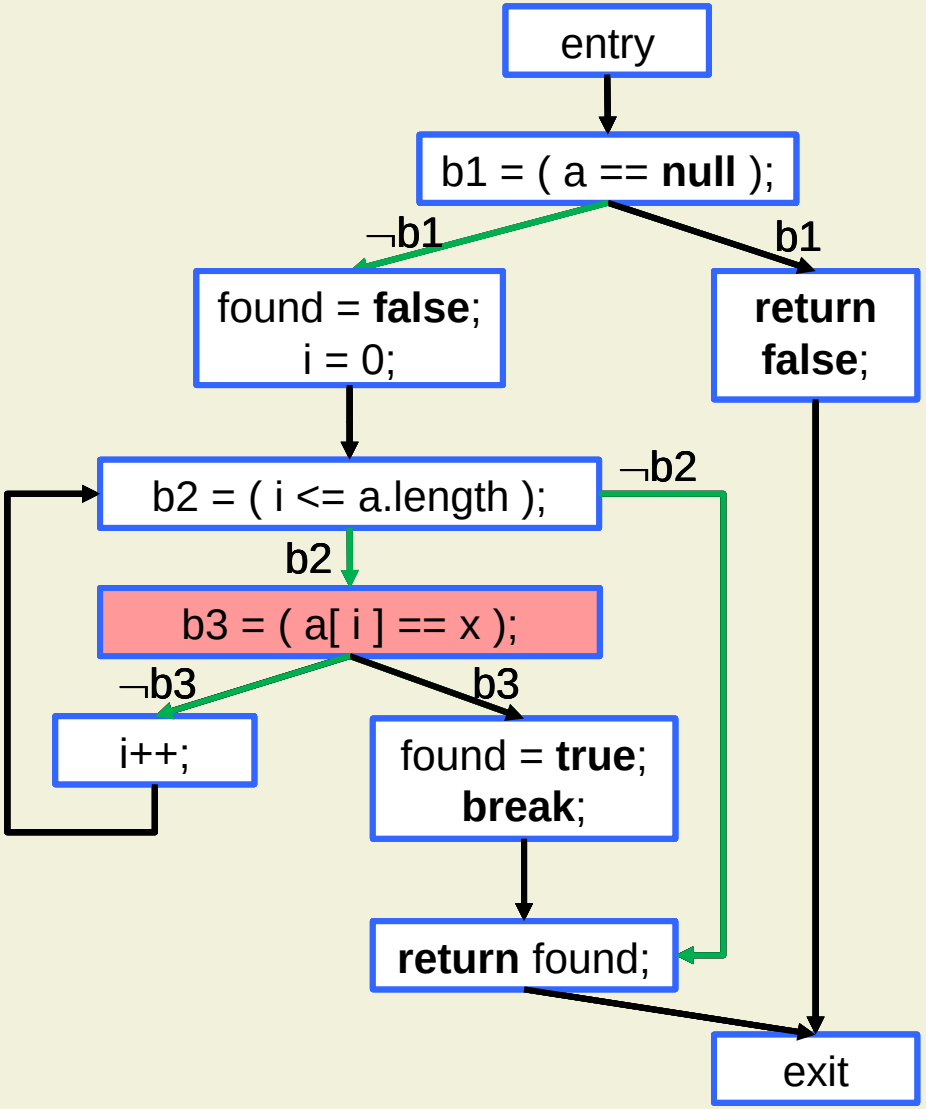
  execute 5 out of 6 branches

- Branch coverage: 83%



# Branch Coverage: Example 2 (cont'd)

- Achieving 100% branch coverage would require a test case that runs the loop to the end
  - a = **null**
  - a = { 1 }, x = 1
  - a = { 1 }, x = 3
- The last test case detects the bug



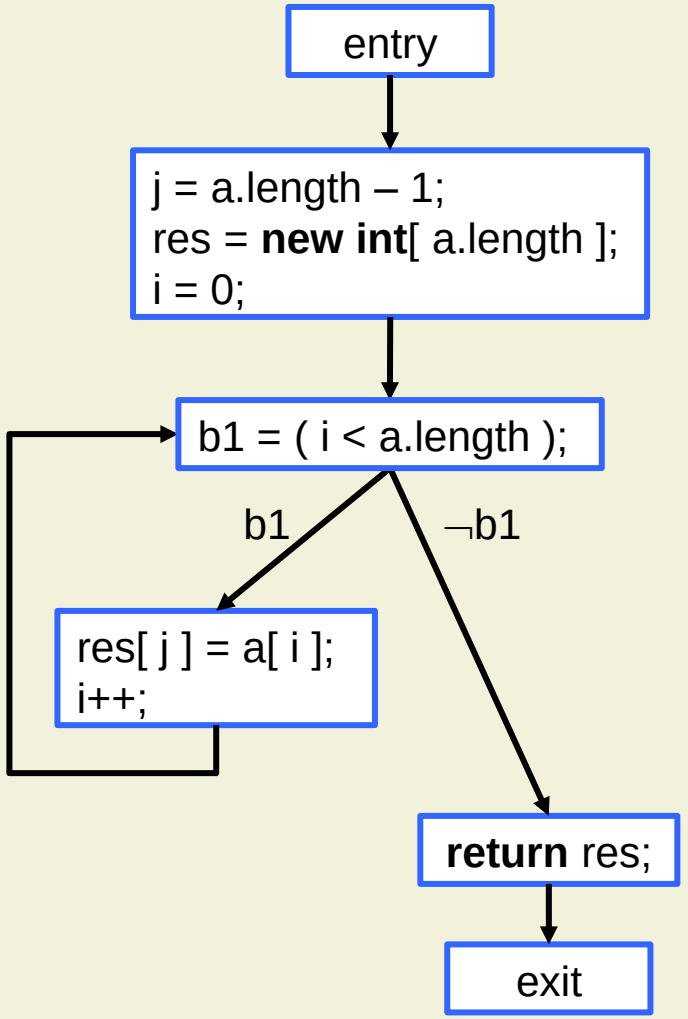
# Branch Coverage: Discussion

- Branch coverage leads to more thorough testing than statement coverage
  - Complete branch coverage implies complete statement coverage
  - But “at least n% branch coverage” does not generally imply “at least n% statement coverage”

- Most widely-used adequacy criterion in industry

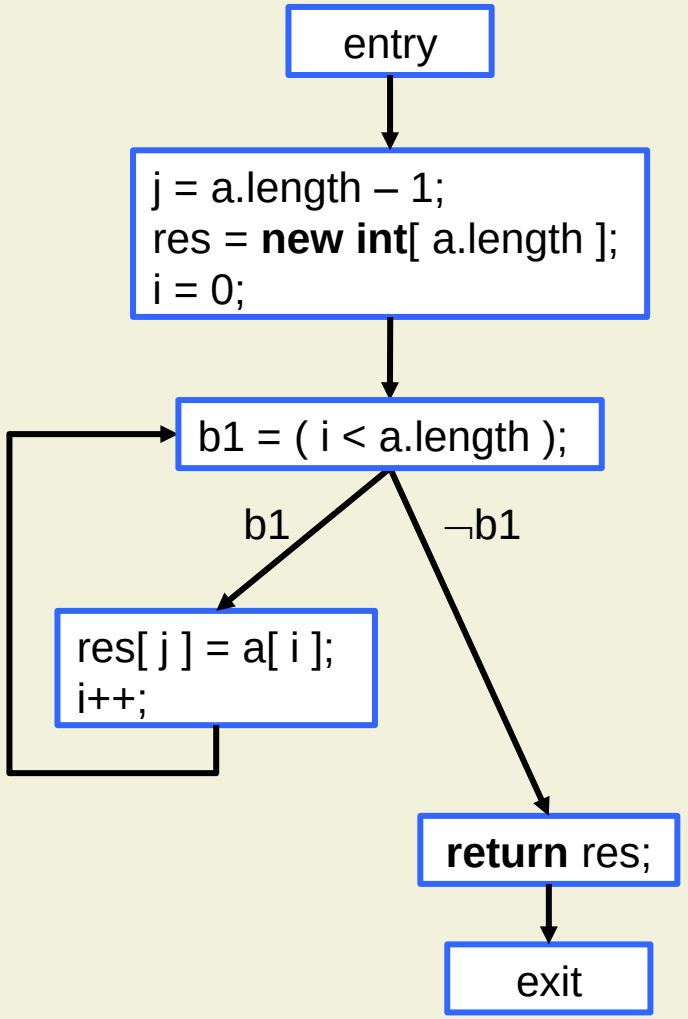
# Branch Coverage: Discussion (cont'd)

```
int[ ] reverse( int[ ] a ) {
  int j = a.length – 1;
  int[ ] res = new int[ a.length ];
  for( int i = 0; i < a.length; i++ ) {
    res[ j ] = a[ i ];
  }
  return res;
}
```



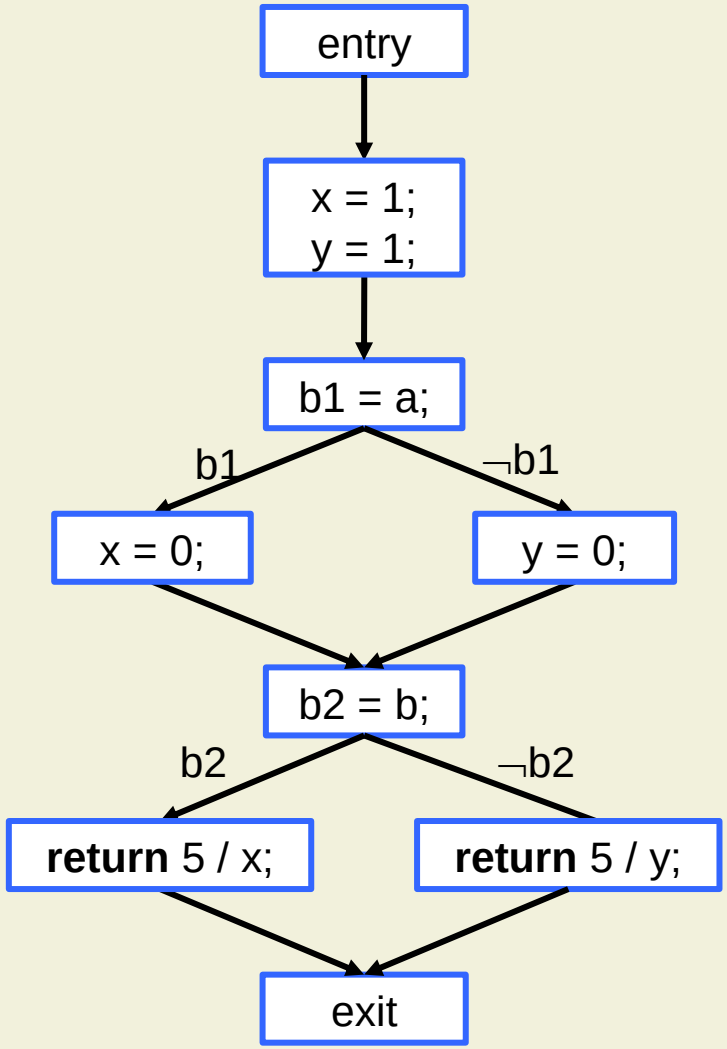
# Branch Coverage: Discussion (cont'd)

- We can achieve 100% branch coverage with one test case
  - a = { 1 }
- The test case does not detect the bug!
- More thorough testing is necessary



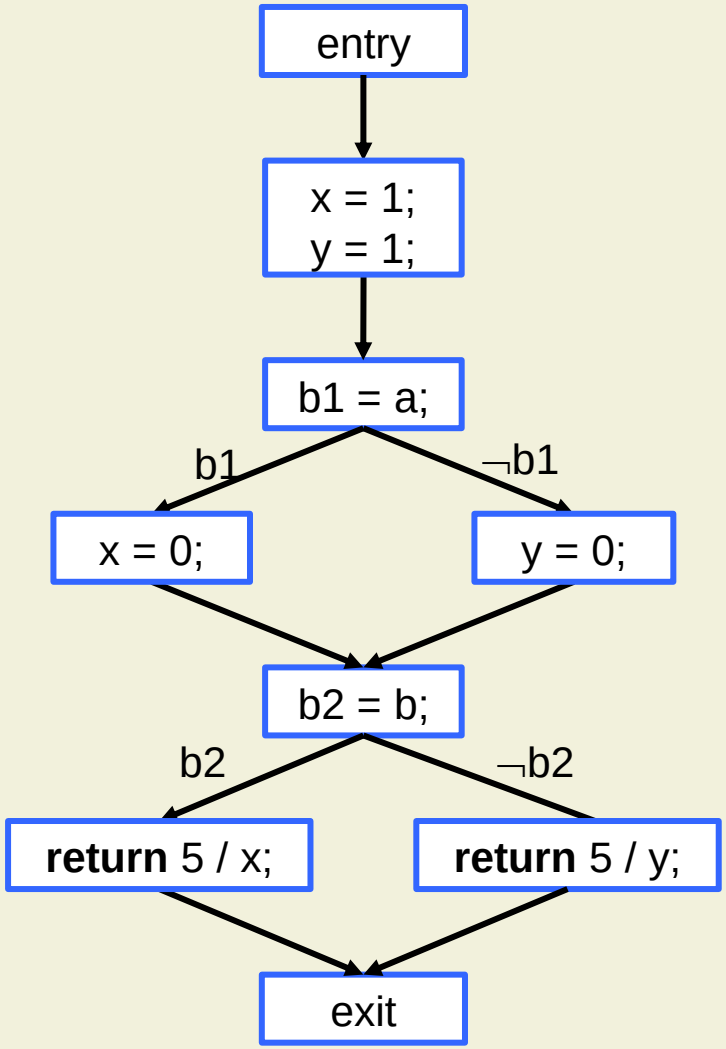
# Branch Coverage: Discussion (cont'd)

```
int foo( boolean a, boolean b ) {
  int x = 1;
  int y = 1;
  if( a )
    x = 0;
  else
    y = 0;
  if( b )
    return 5 / x;
  else
    return 5 / y;
}
```



# Branch Coverage: Discussion (cont'd)

- We can achieve 100% branch coverage with two test cases
  - a = **true**, b = **false**
  - a = **false**, b = **true**
- The test cases do not detect the bug!
- More thorough testing is necessary



# Path Coverage

- Idea: test all possible paths through the CFG

- A path is a sequence of nodes $n_1$, …, $n_k$ such that
  - $n_1$ = entry
  - $n_k$ = exit
  - There is an edge ($n_i$, $n_{i+1}$, c) in the CFG

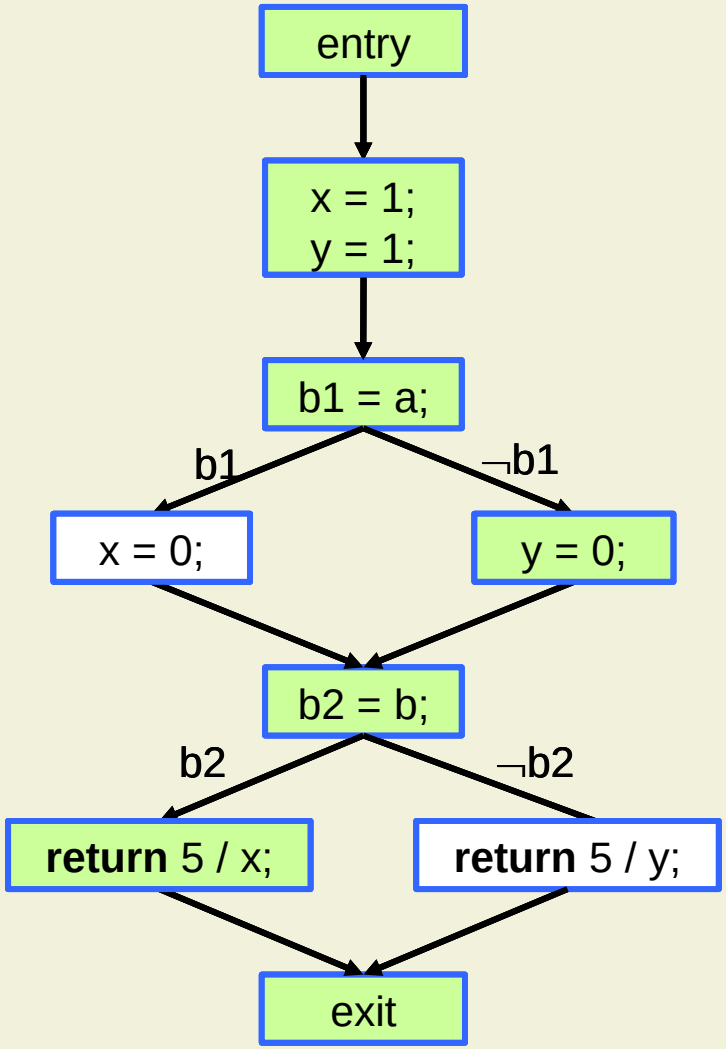
$$\text{Path Coverage} = \frac{\text{Number of executed paths}}{\text{Total number of paths}}$$

# Path Coverage: Example 1

- The two test cases
  - a = **true**, b = **false**
  - a = **false**, b = **true**

  execute two out of four paths

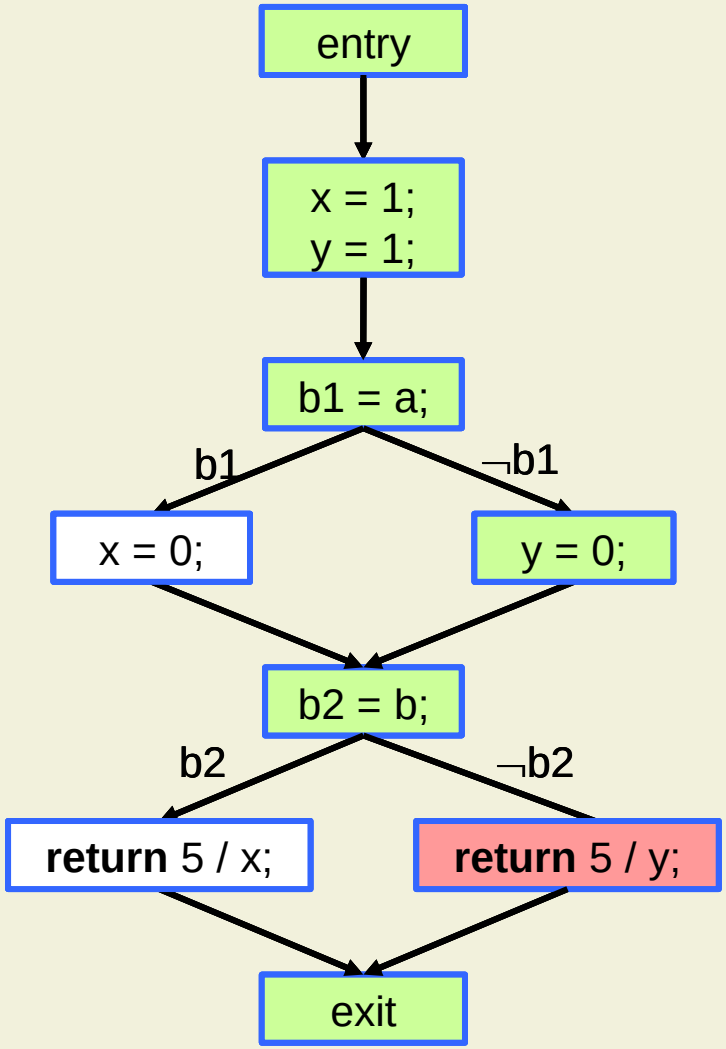
- Path coverage: 50%



# Path Coverage: Example 1 (cont'd)

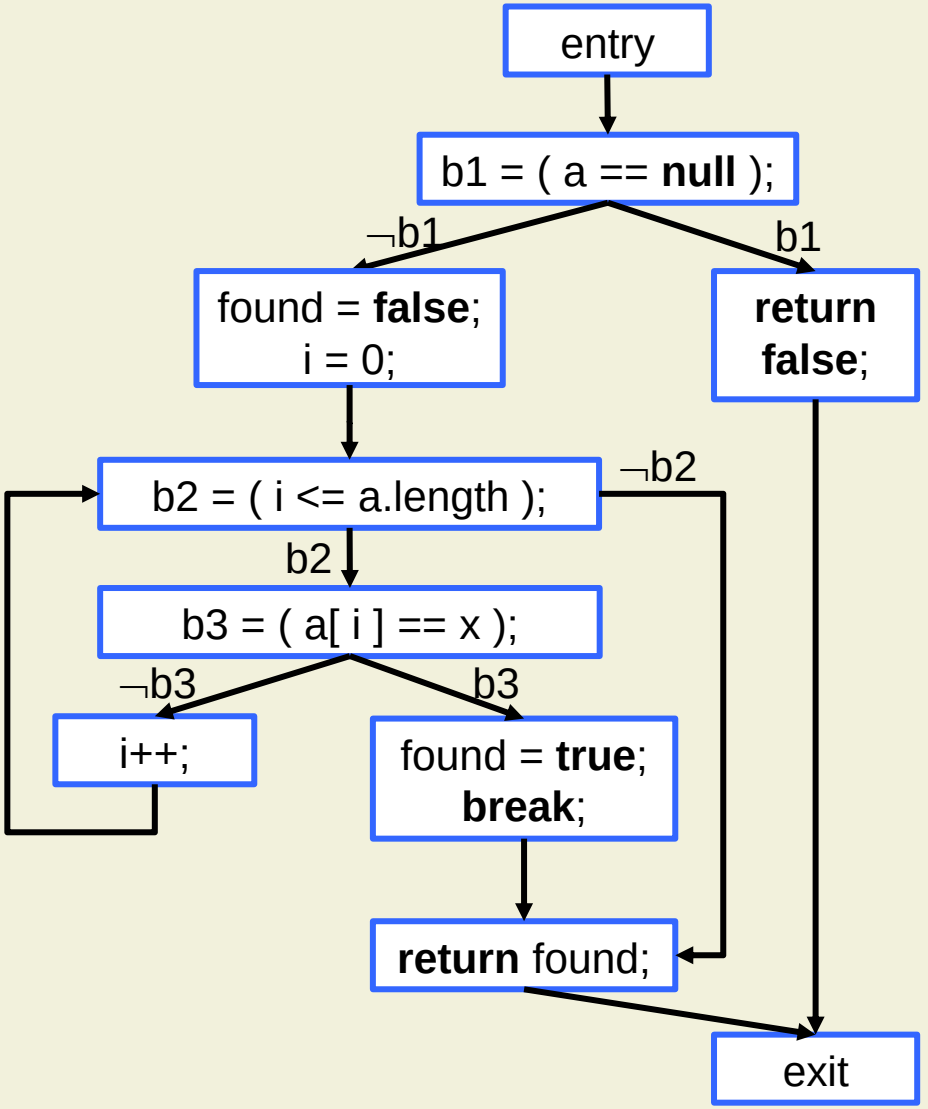
- We can achieve 100% path coverage with four test cases
  - a = **true**, b = **false**
  - a = **false**, b = **true**
  - a = **true**, b = **true**
  - a = **false**, b = **false**

- The two additional test cases detect the bugs



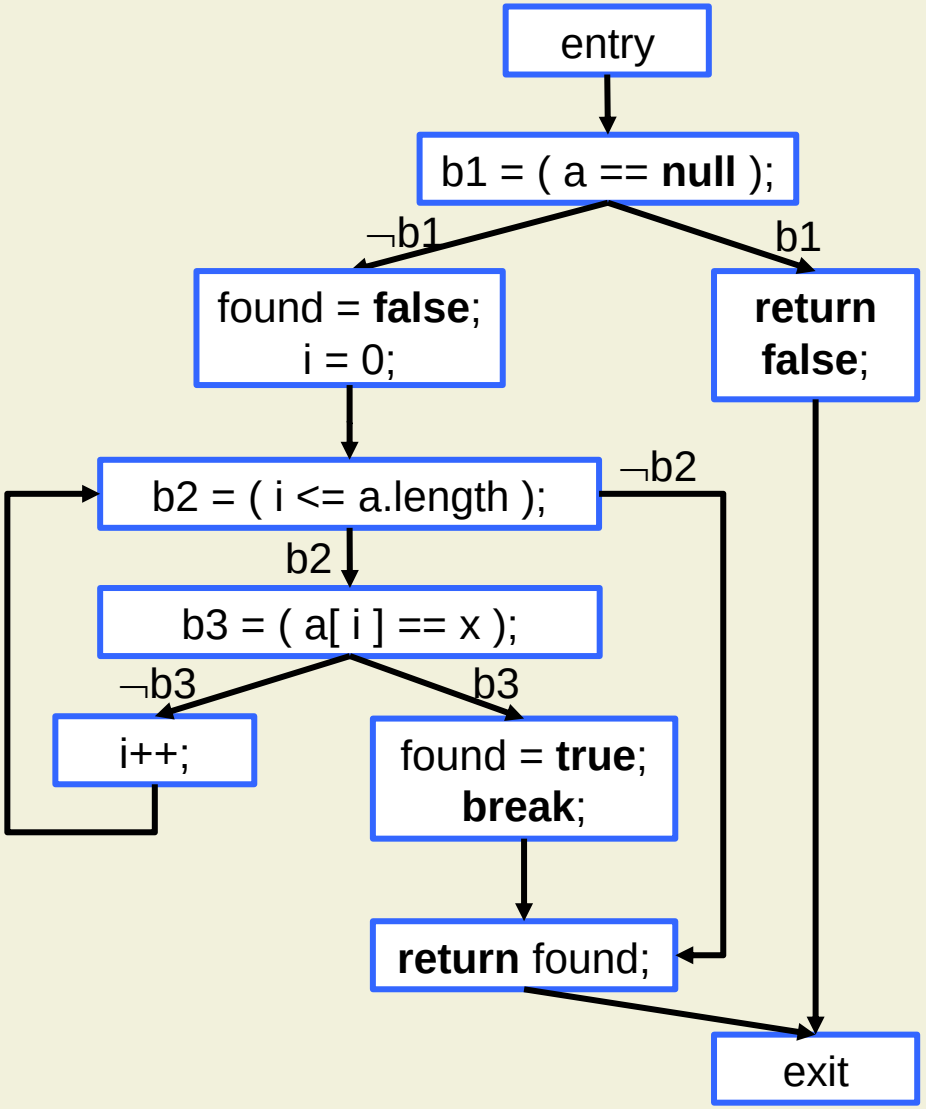
# Path Coverage: Example 2

```
boolean contains( int[ ] a, int x ) {
  if( a == null )  return false;
  boolean found = false;
  for( int i = 0; i <= a.length; i++ ) {
    if( a[ i ] == x ) {
      found = true;
      break;
    }
  }
  return found;
}
```



# Path Coverage: Example 2 (cont'd)

- Number of loop iterations is not known statically
- An arbitrarily large number of test cases is needed for complete path coverage



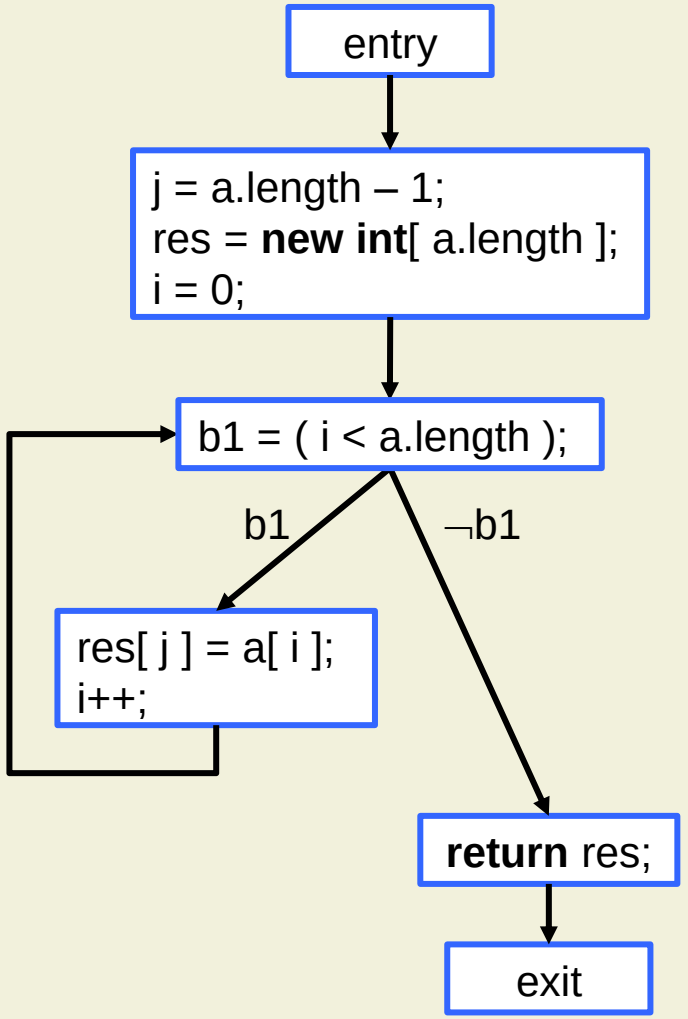
# Path Coverage: Discussion

- Path coverage leads to more thorough testing than both statement and branch coverage
  - Complete path coverage implies complete statement coverage and complete branch coverage
  - But “at least n% path coverage” does not generally imply “at least n% statement coverage” or “at least n% branch coverage”

- Complete path coverage is not feasible for loops
  - Unbounded number of paths

# Branch Coverage: Discussion (cont'd)

```
int[ ] reverse( int[ ] a ) {
  int j = a.length – 1;
  int[ ] res = new int[ a.length ];
  for( int i = 0; i < a.length; i++ ) {
    res[ j ] = a[ i ];
  }
  return res;
}
```



# Loop Coverage

- Idea: for each loop, test zero, one, and more than one iterations

$$\text{Loop Coverage} = \frac{\text{Number of executed loops with 0, 1, and more than 1 iterations}}{\text{Total number of loops} * 3}$$

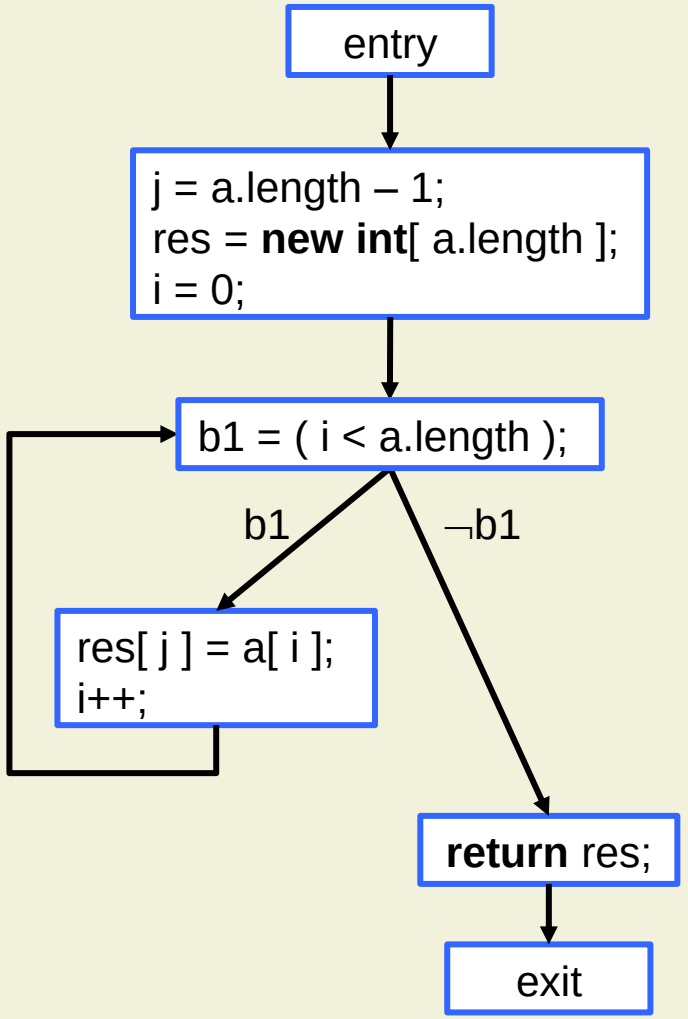
- Loop coverage is typically combined with other adequacy criteria such as statement or branch coverage

# Loop Coverage: Example

- The test case
  - a = { 1 }

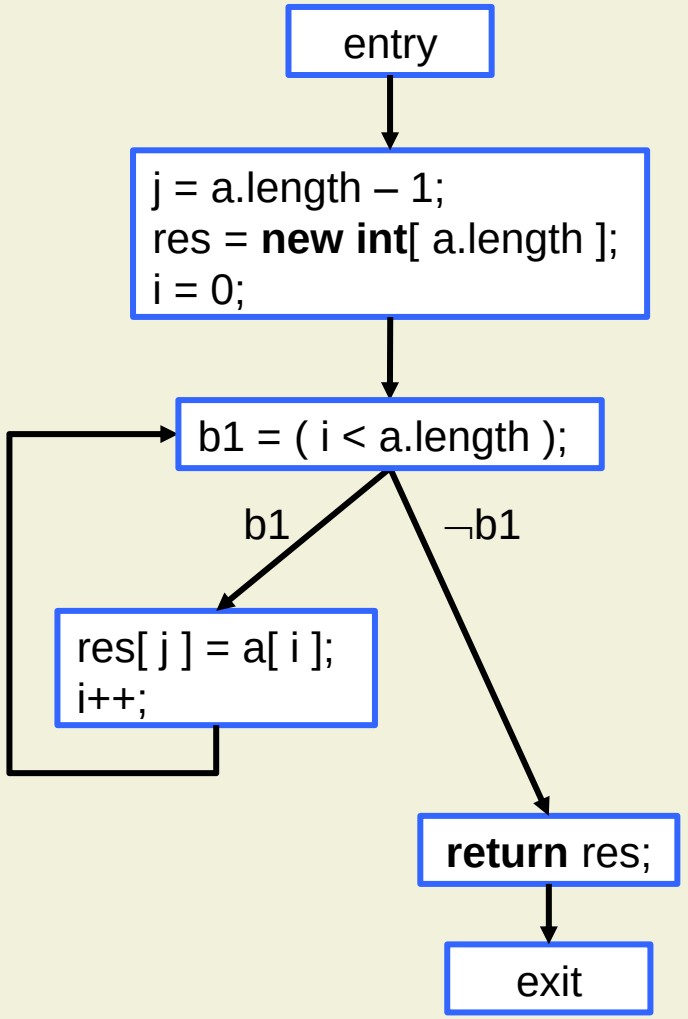
  executes one out of three possible cases for the loop

- Loop coverage: 33%



# Loop Coverage: Example

- We can achieve 100% loop coverage with three test cases
  - a = { }
  - a = { 1 }
  - a = { 1, 2 }
- The last test case detects the bug



# Measuring Coverage

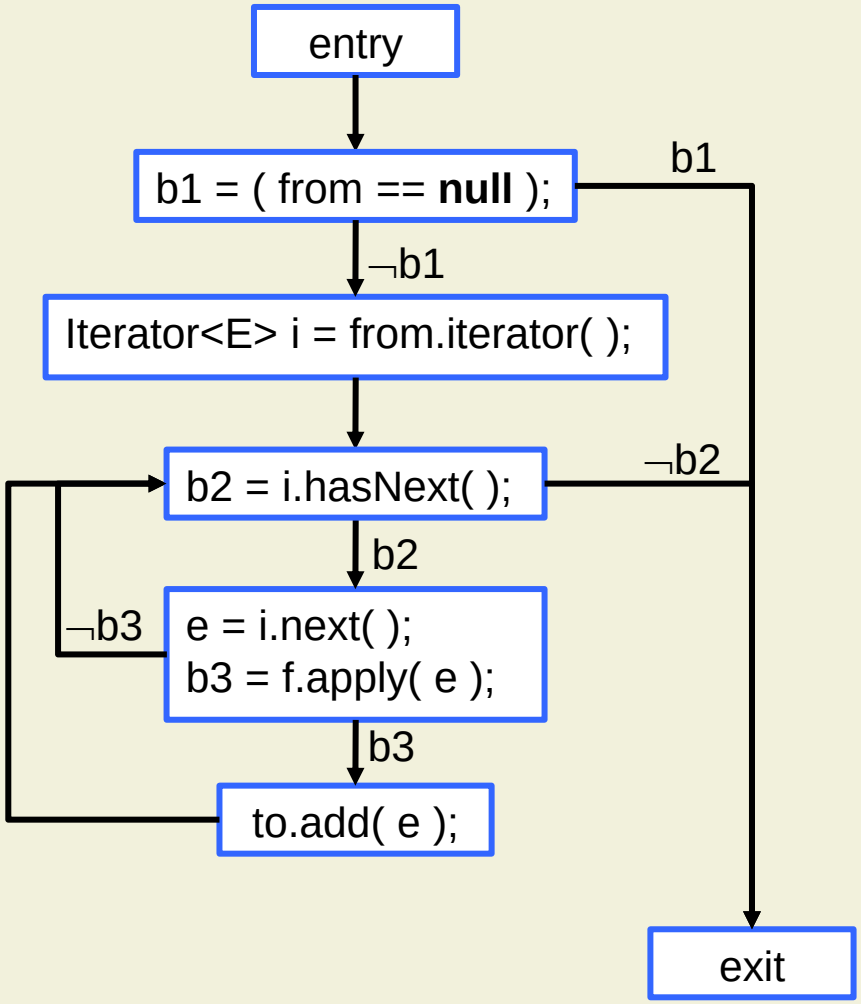
- Coverage information is collected while the test cases execute
- Use code instrumentation or debug interface to count executed basic blocks, branches, etc.

```
int foo( boolean a, boolean b ) {
  int x = 1;  int y = 1;
  if( a ) {
    executedBranches[ 0 ]++;  x = 0;
  } else {
    executedBranches[ 1 ]++;  y = 0;
  }
  if( b ) {
    executedBranches[ 2 ]++;
    return 5 / x;
  } else {
    executedBranches[ 3 ]++;
    return 5 / y;
  }
}
```

# 8. Test Case Selection

# CFG: Method Calls

```
static <E> void filter(
              Collection<E> from,
              Filter<E> f,
              Collection<E> to ) {
  if( from == null ) return;
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
  }
}
```



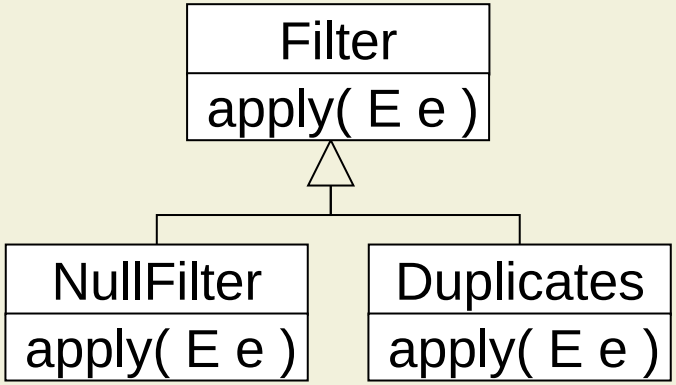
# Dynamically-Bound Method Calls

```
static <E> void filter(
            Collection<E> from,
            Filter<E> f,
            Collection<E> to ) {
  if( from == null ) return;
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
  }
}
```

- Intraprocedural CFGs treat method calls as simple statements
- Yet, calls **invoke different code** depending on the dynamic type of the receiver
- Testing should **cover the possible behaviors**

# Testing Dynamically-Bound Method Calls

- A dynamically-bound method call can be regarded as a case distinction on the type of the receiver




f.apply( e )

```
if( type( f ) == Filter )
  f.Filter::apply( e );
else if( type( f ) == NullFilter )
  f.NullFilter::apply( e );
else // type( f ) == Duplicates
  f.Duplicates::apply( e );
```

- Now we can apply branch testing

# Testing Dynamically-Bound Calls (cont'd)

- Treating dynamically-bound method calls as branches leads to a **combinatorial explosion**
- Use semantic constraints and pairwise-combinations testing

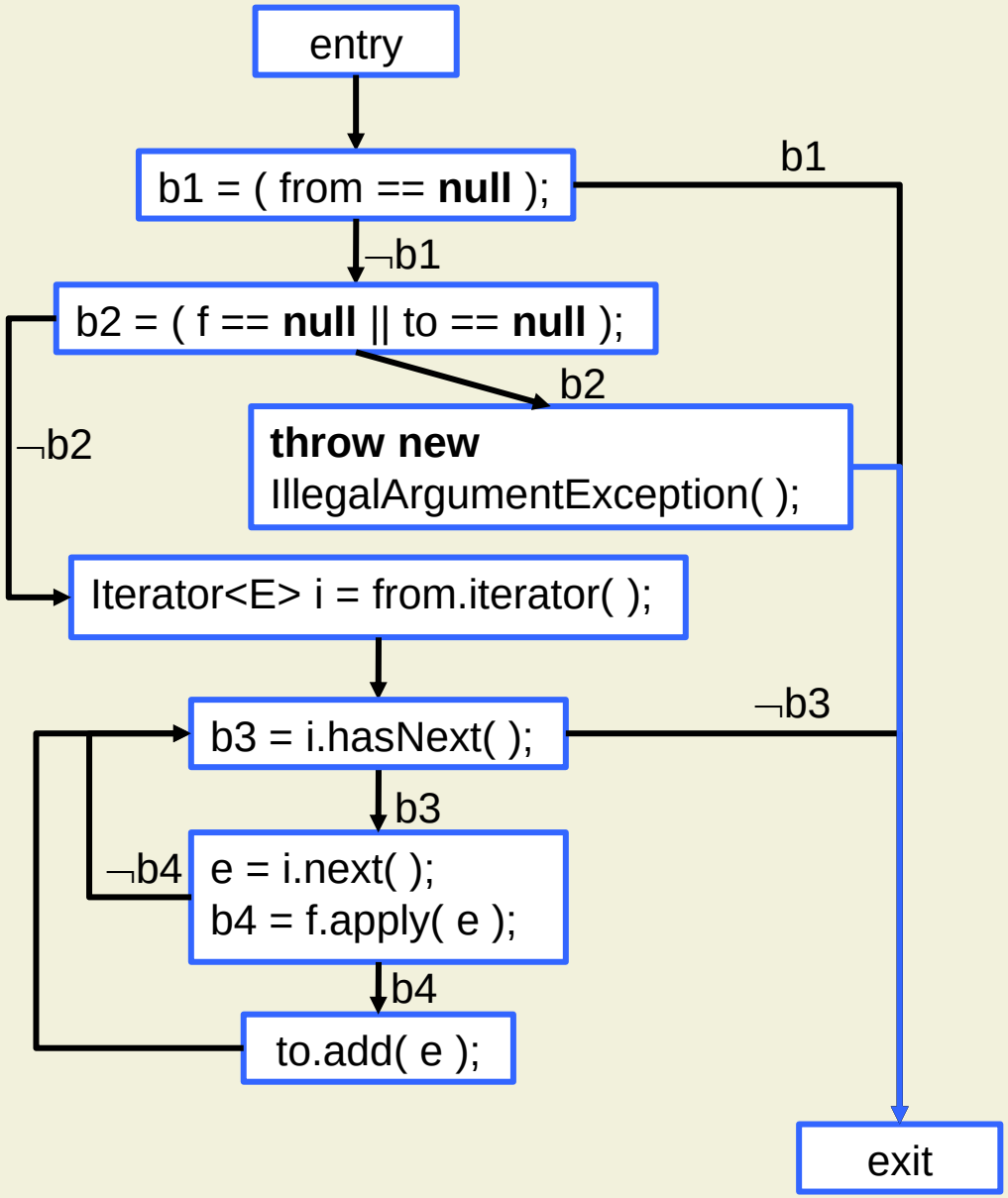
```
static <E> void filter(
            Collection<E> from,
            Filter<E> f,
            Collection<E> to ) {
  if( from == null ) return;
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
```

Several different Filter classes in the program

java.util contains dozens of collection classes

# Exceptions

```
static <E> void filter(
            Collection<E> from,
            Filter<E> f,
            Collection<E> to ) {
  if( from == null ) return;
  if( f == null || to == null )
    throw new
      IllegalArgumentException( );
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
  }
}
```



# CFG: Exceptions

- Exceptions add a control flow edge from the basic block where the exception is thrown to the exit block or the block where the exception is caught
- Idea: Cover exceptional control flow like normal control flow during testing
  - Test oracle is checked when method terminates normally

```
[ Test ]
[ ExpectedException( typeof(ArgumentException) ) ]
public void TestDemoInvalid( … ) {
  int d = Days( month, year );
}
```

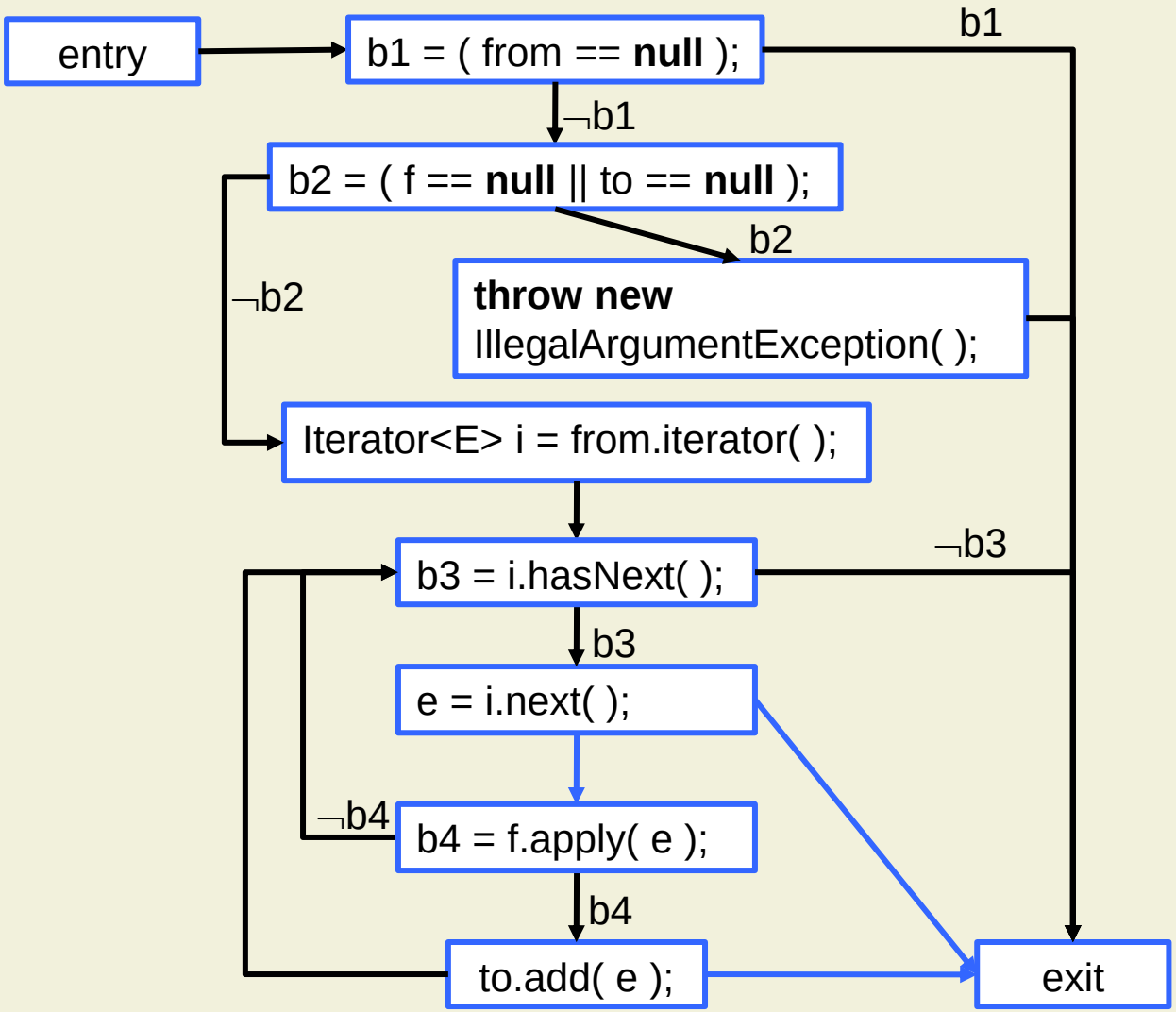
# Example: Documented Exceptions

```
static <E> void filter(
            Collection<E> from,
            Filter<E> f,
            Collection<E> to ) {
  if( from == null ) return;
  if( f == null || to == null )
    throw new
      IllegalArgumentException( );
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
  }
}
```

Might throw:
- NoSuchElementException

Might throw:
- UnsupportedOperationException
- ClassCastException
- NullPointerException
- IllegalArgumentException
- IllegalStateException

# Example: Documented Exceptions (cont'd)



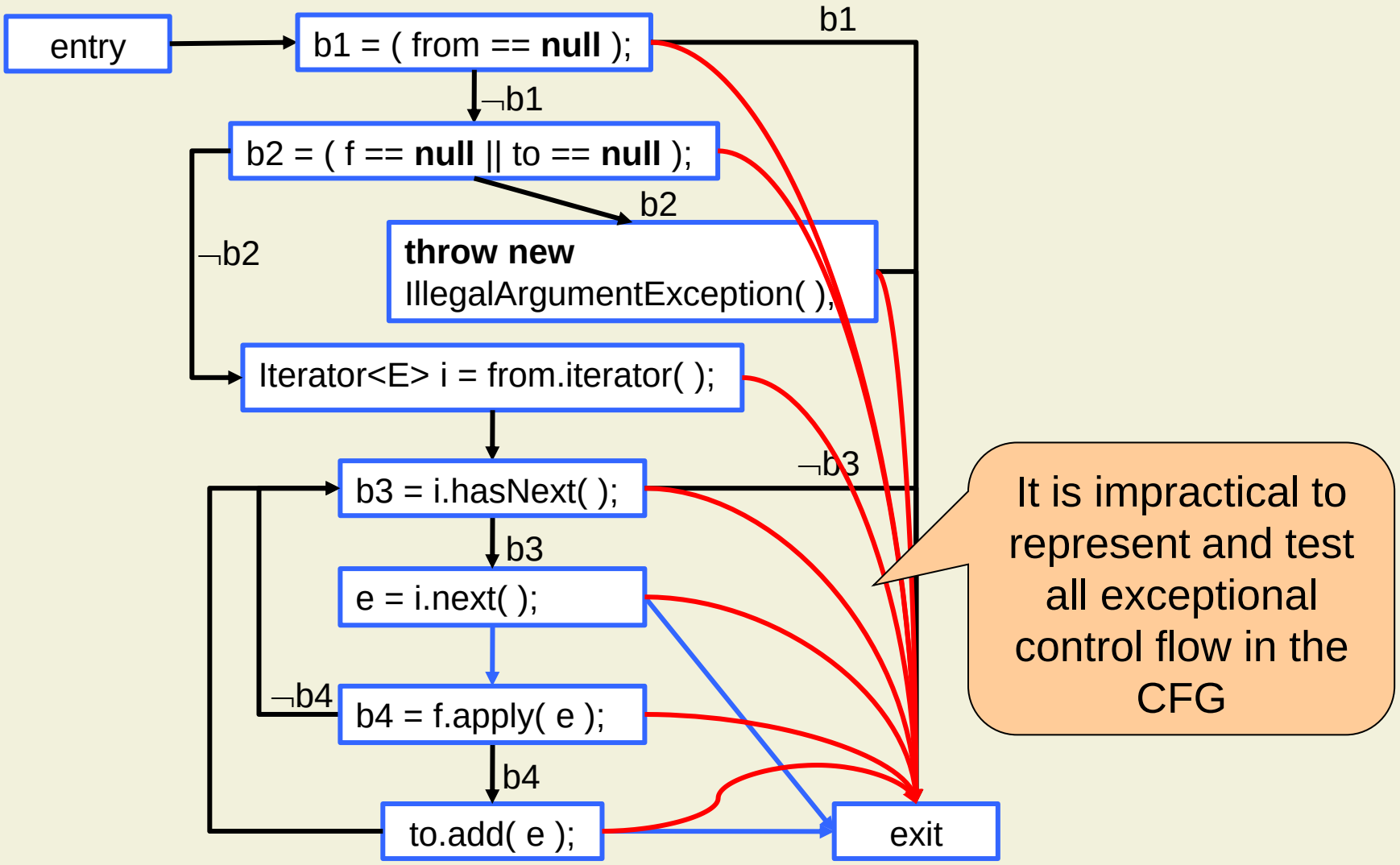
# Example: Undocumented Exceptions

```
static <E> void filter(
            Collection<E> from,
            Filter<E> f,
            Collection<E> to ) {
  if( from == null ) return;
  if( f == null || to == null )
    throw new
      IllegalArgumentException( );
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
  }
}
```

The example might also throw:

- ConcurrentModificationException
- NoClassDefFoundError
- NoSuchMethodError
- OutOfMemoryError
- StackOverflowError
- ThreadDeath
- VirtualMachineError
- etc.

# Example: Undocumented Exceptions (cont'd)

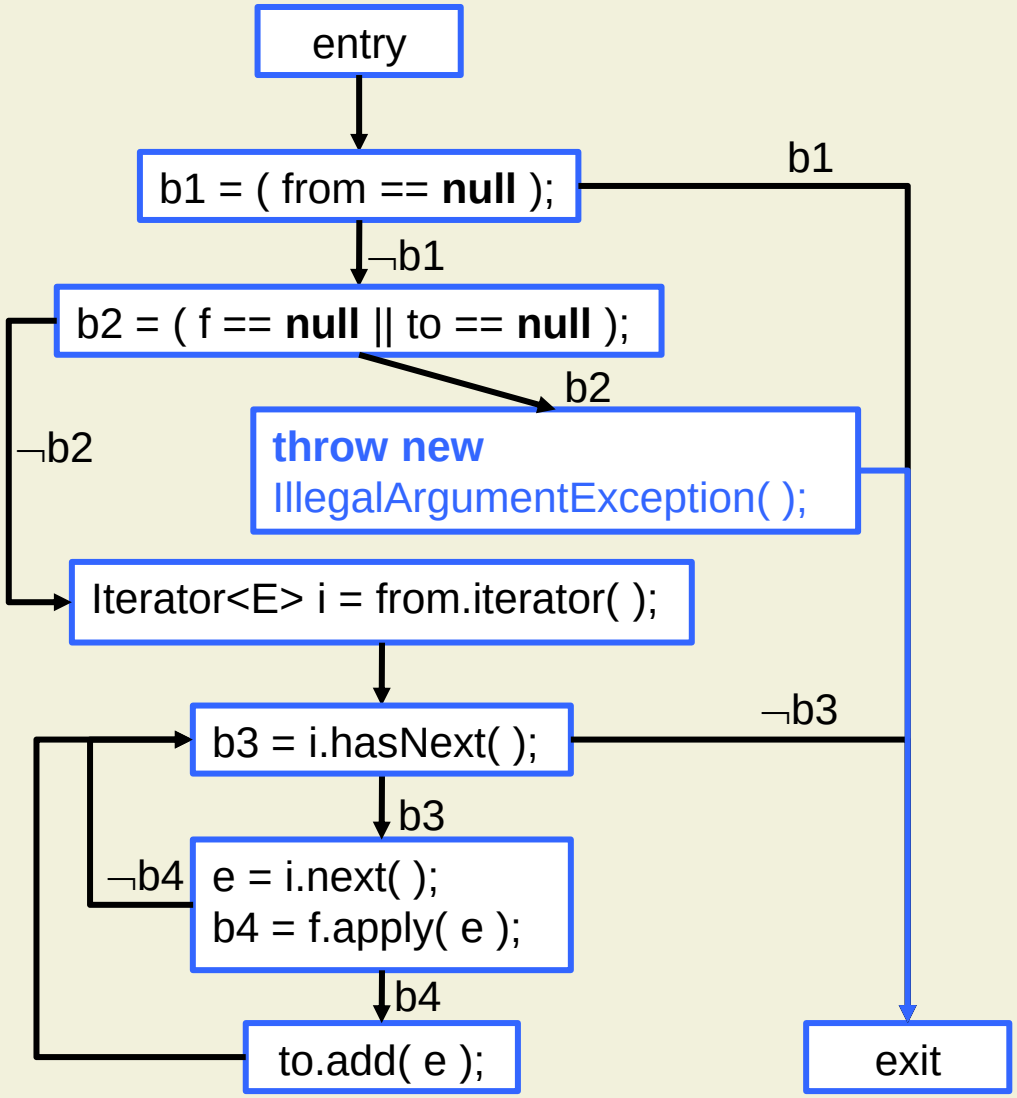


# Checked vs. Unchecked Exceptions

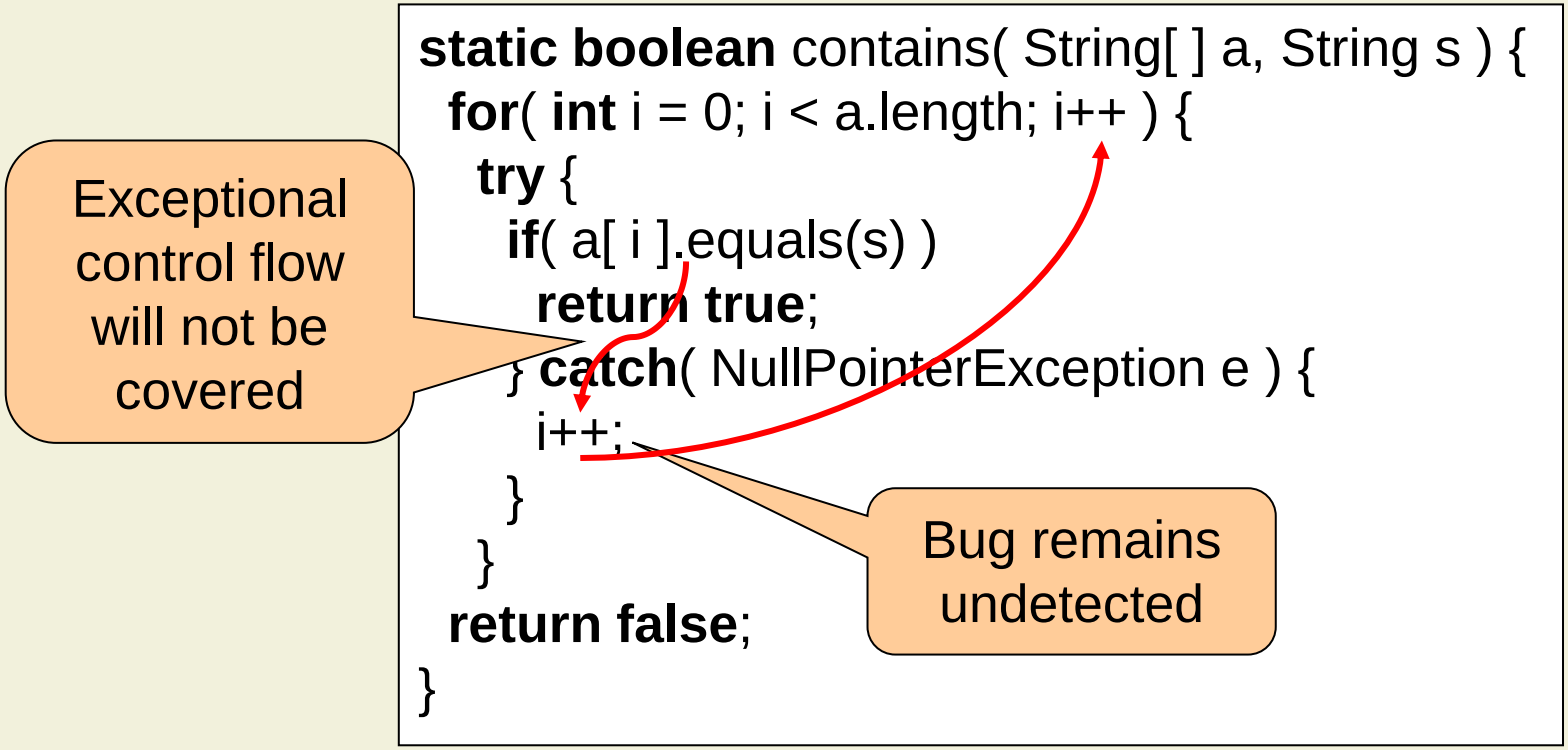
- Many programming languages distinguish between checked and unchecked exceptions
- **Checked exceptions** represent invalid conditions outside the immediate control of the program
  - Invalid user input, database problems, network outages, absent files
- **Unchecked exceptions** represent defects in the program or the execution environment
  - Illegal arguments, null-pointer dereferencing, division by zero, assertion violation, etc.
  - In Java: all subclasses of RuntimeException and Error

# Testing Unchecked Exceptions

- Unchecked exceptions are not supposed to occur
- When computing the CFG, **ignore unchecked exceptions** thrown by other methods and virtual machine
  - But consider **throw** statements

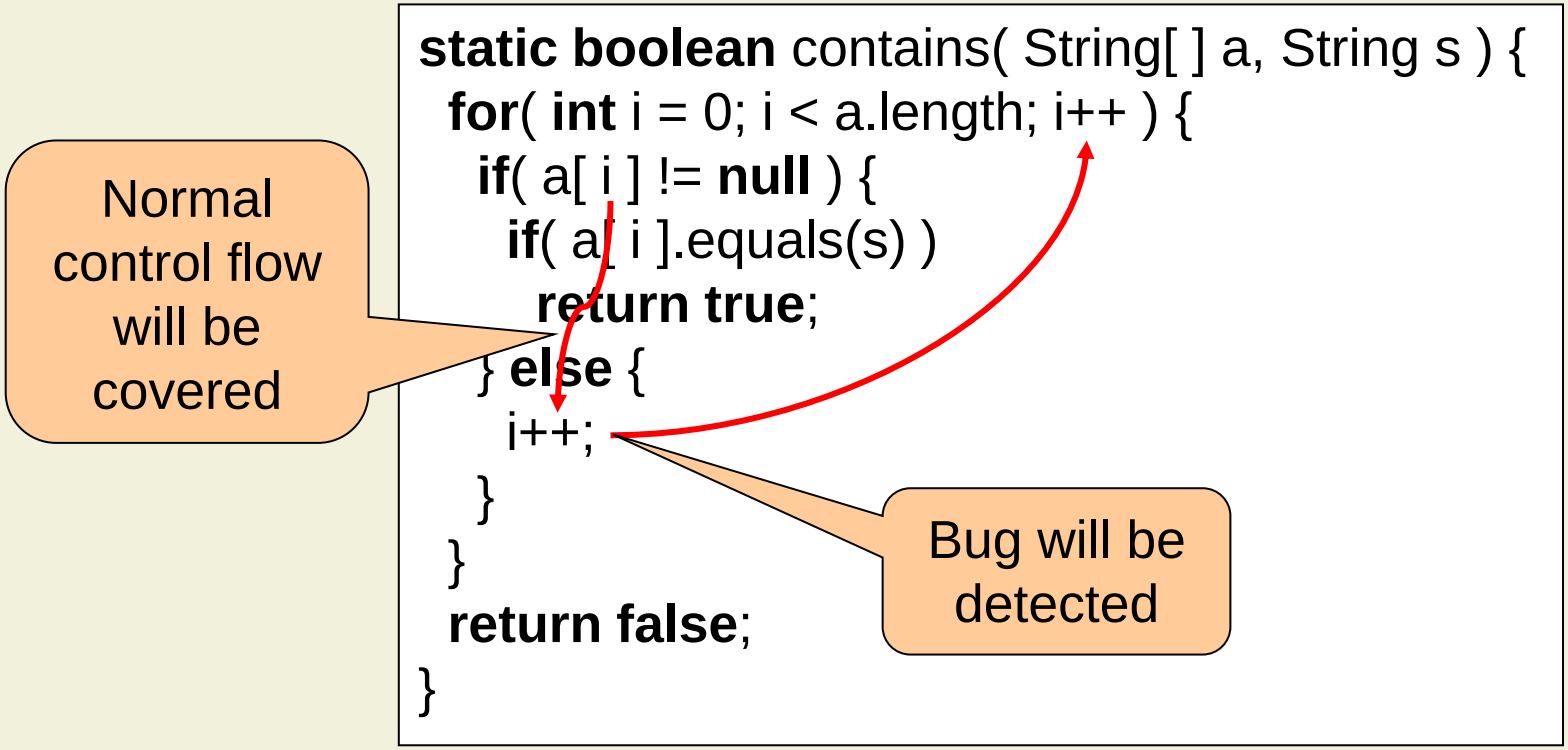


# Unchecked Exceptions: Bad Example




- Never use unchecked exceptions to encode control flow!

# Bad Example Fixed



# Testing Checked Exceptions

- Checked exceptions represent **regular control flow that needs to be tested**
  - Include control flow in CFG, testing, and coverage

- In Java, checked exceptions are declared in method signatures

```
interface RemoteBuffer extends Remote {
  void put( String s ) throws RemoteException;
}
```

- For each call, add appropriate control flow edges

# Checked Exceptions: Example

```
class Producer {
  RemoteBuffer b;

  void produce( ) throws RemoteException {
    boolean retried = false;
    boolean success = false;
    while( !success ) {
      try {
        b.put( "Product“ );
        success = true;
      } catch( RemoteException e ) {
        if( retried )  throw e;
      }
    }
  }
}
```

Exceptional control flow will be covered

Bug will be detected

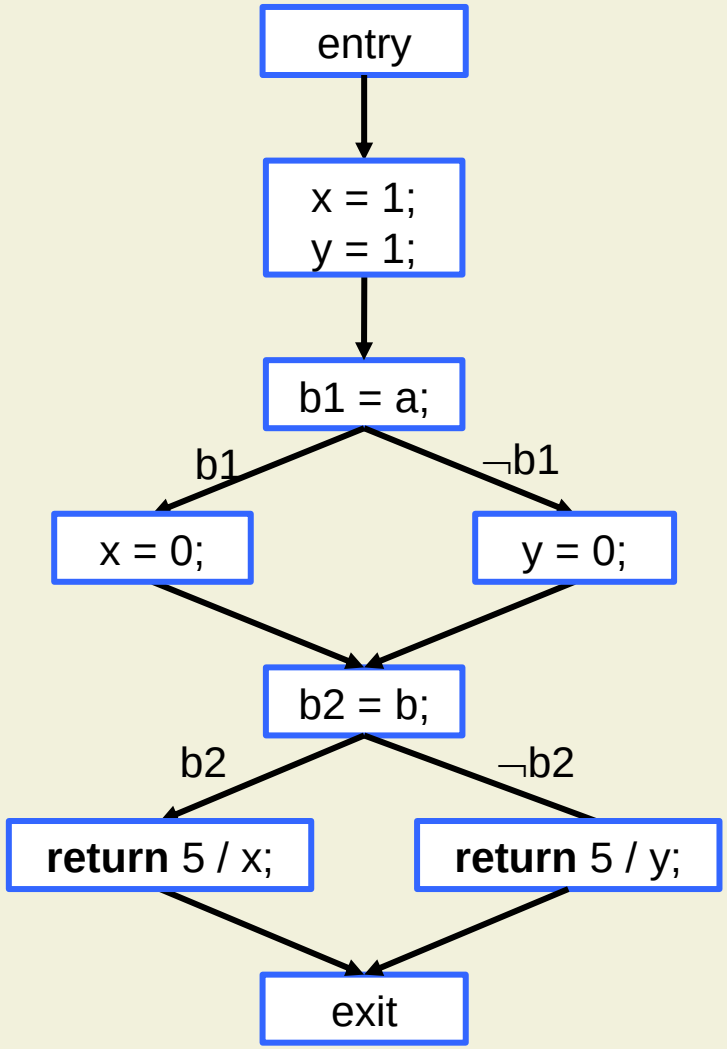
# Testing Exceptions: Summary

- Checked exceptions encode the program's reaction to invalid conditions in the environment
  - Test like normal control flow
- Unchecked exceptions represent defects
  - Test unchecked exceptions explicitly thrown by method under test (argument validation, precondition check)
  - Unchecked exceptions thrown by methods being called indicate defect in method under test (precondition violation) or in the called method
  - Unchecked exceptions thrown by virtual machine indicate defect in method under test (e.g., infinite recursion) or deployment error (e.g., class not found)

# 8. Test Case Selection

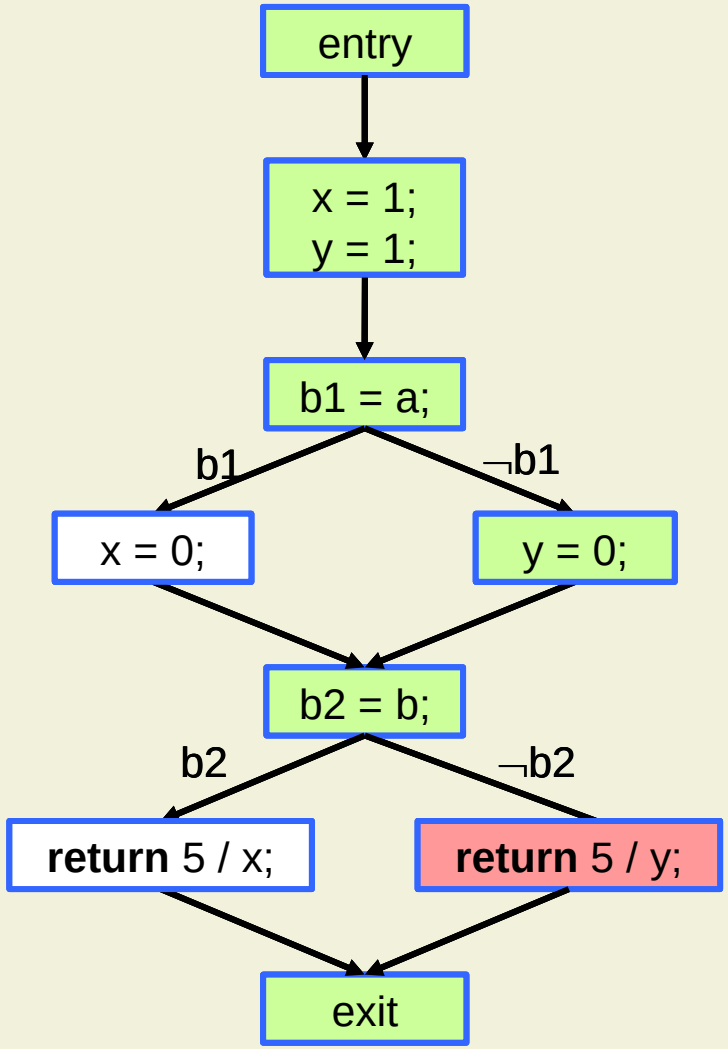
# Example Revisited

```
int foo( boolean a, boolean b ) {
  int x = 1;
  int y = 1;
  if( a )
    x = 0;
  else
    y = 0;
  if( b )
    return 5 / x;
  else
    return 5 / y;
}
```



# Data Flow Testing

- Testing all paths is not feasible
  - Number grows exponentially in the number of branches
  - Loops
- Idea: Test those paths where a computation in one part of the path affects the computation of another



# Variable Definition and Use

- A **variable definition** for a variable v is a basic block that assigns to v
  - v can be a local variable, formal parameter, field, or array element

- A **variable use** for a variable v is a basic block that reads the value from v
  - In conditions, computations, output, etc.

# Definition-Clear Paths

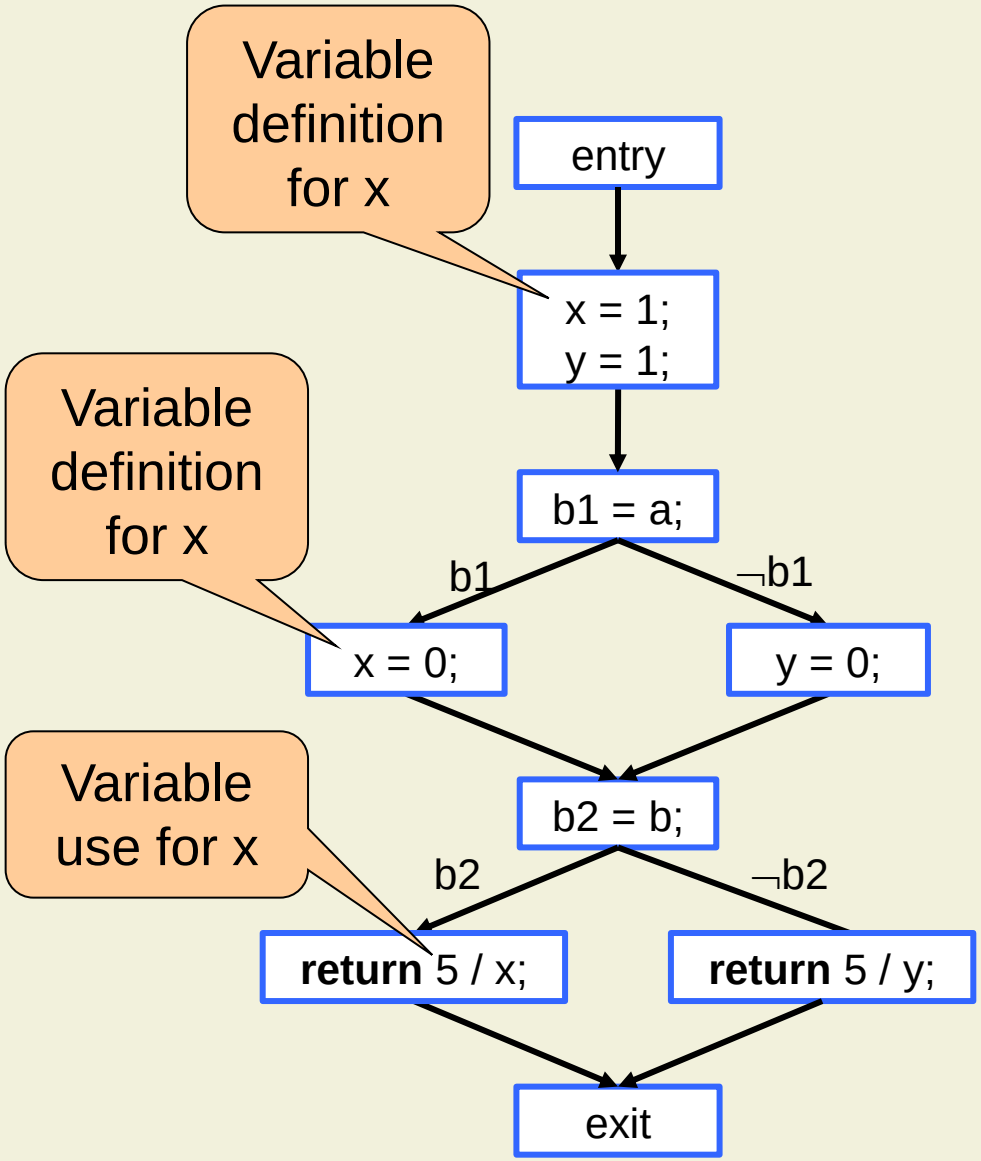
- A **definition-clear path** for a variable v is a path $n_1, \ldots, n_k$ in the CFG such that:
  - $n_1$ is a variable definition for v
  - $n_k$ is a variable use for v
  - No $n_i$ $(1 < i \leq k)$ is a variable definition for v
    ($n_k$ may be a variable definition if each assignment to v occurs after a use)

- Note: definition-clear paths do not go from entry to exit (in contrast to our earlier definition of path)

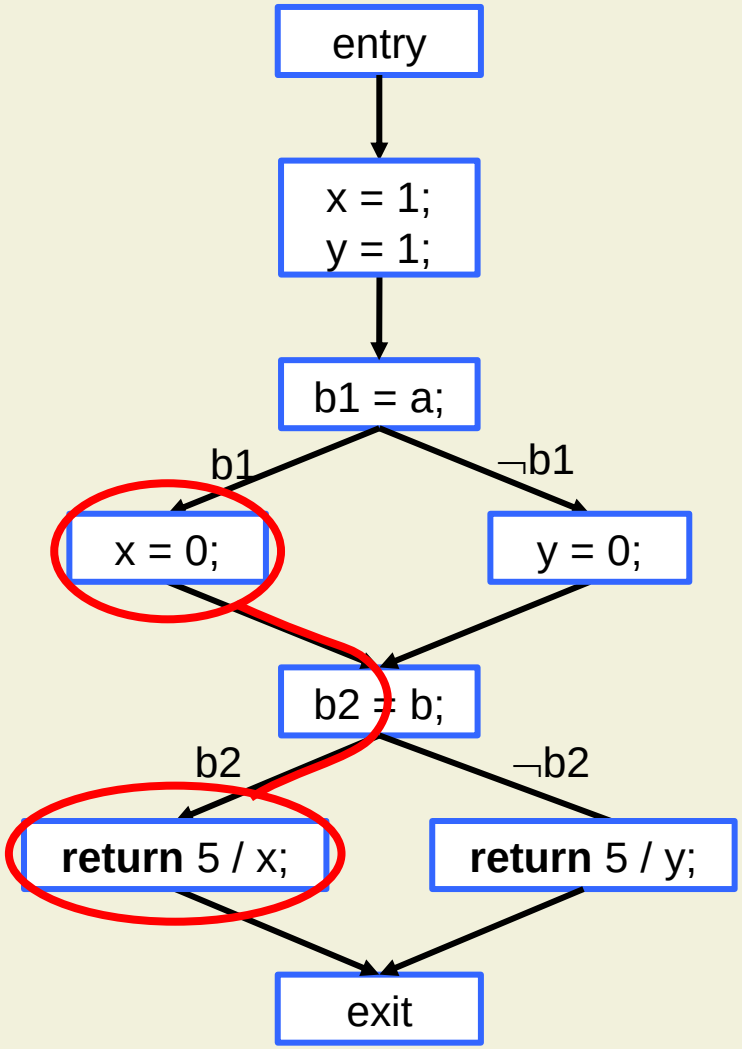
# Definition-Use Pairs

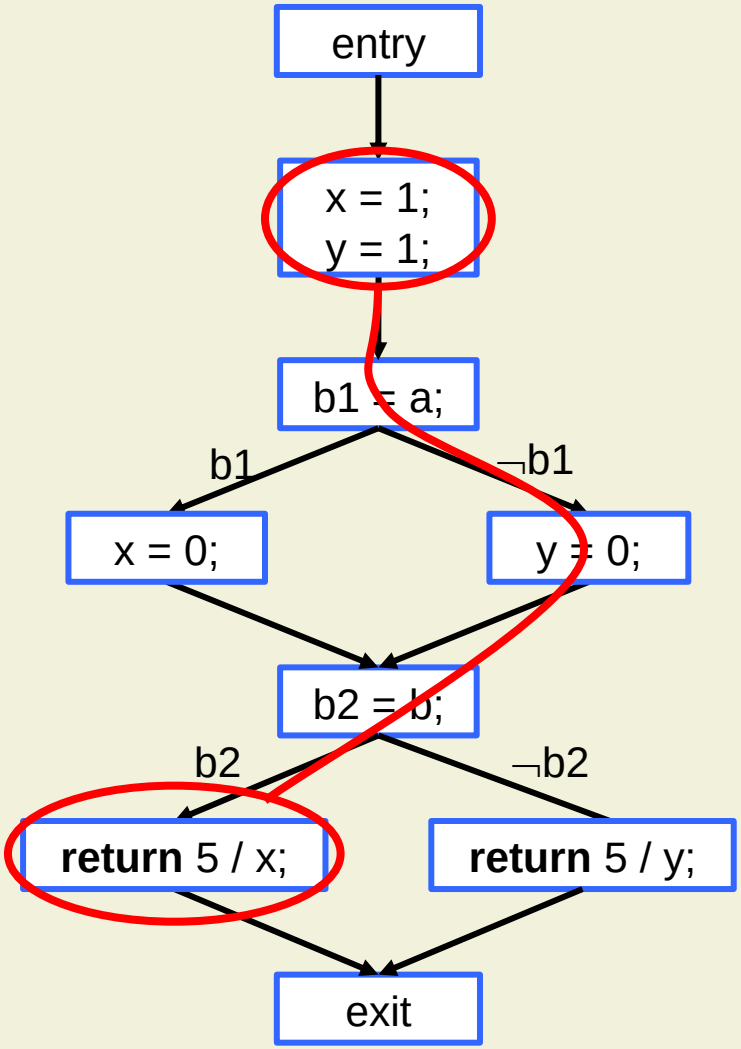
- A **definition-use pair** for a variable v is a pair of nodes (d,u) such that there is a definition-clear path d, …, u in the CFG

- We say **DU-pair** for definition-use pair



# Definition-Use Pairs: Examples



# DU-Pairs Coverage

- Idea: test all paths that provide a value for a variable use

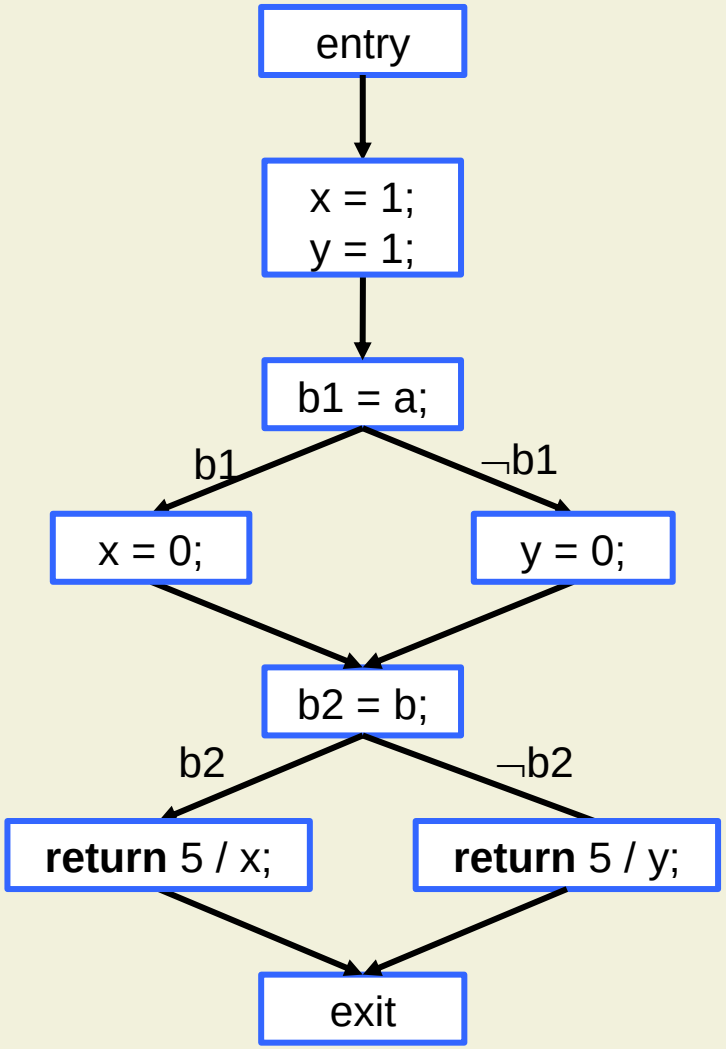
$$\text{DU-Pairs Coverage} = \frac{\text{Number of executed DU-Pairs}}{\text{Total number of DU-Pairs}}$$

# DU-Pairs Coverage: Example

- The two test cases
  - a = **true**, b = **false**
  - a = **false**, b = **true**

  achieve 100% branch coverage, but only 50% DU-pairs coverage

- In this example, DU-pairs coverage is equivalent to path coverage



# Determining all DU-Pairs

- DU-Pairs are computed using a static **reaching-definitions** analysis
- For each node n and for each variable v, compute all variable definitions for v that possibly reach n via a definition-clear path

- The reaching definitions at a node n are:
  - The reaching definitions of n's predecessors in the CFG
  - minus the definitions killed by one of n'd predecessors
  - plus the definitions made by one of n'd predecessors

# Reaching Definitions: Algorithm

- Input
  - pred( n ) = { m | (m,n,c) is an edge in the CFG }
  - succ( m ) = { n | (m,n,c) is an edge in the CFG }
  - gen( n ) = { $v_n$ | n is a variable definition for v }
  - kill( n ) = { $v_m$ | n is a variable definition for v and m ≠ n }

- We compute via fixpoint iteration
  - Reach( n ): The reaching definitions at the beginning of n
  - ReachOut( n ): The reaching definitions at the end of n

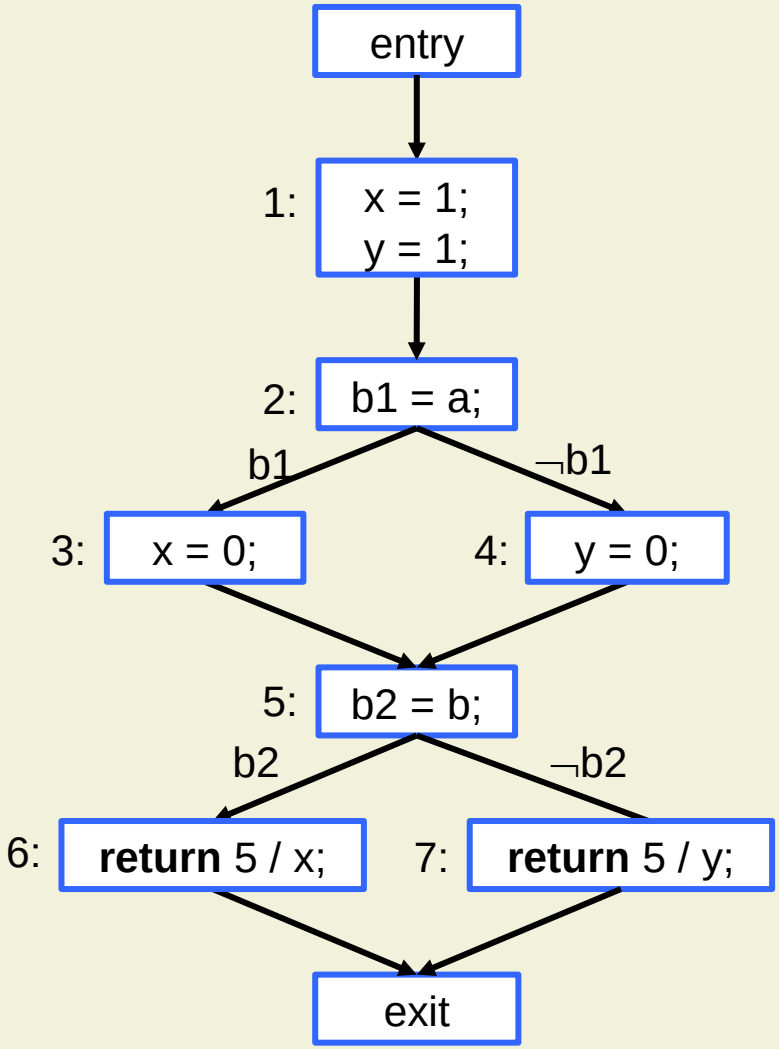
# Reaching Definitions: Algorithm (con't)

**foreach** node n **do** ReachOut( n ) := ∅ **end**

worklist := nodes

**while** worklist ≠ ∅ **do**
  n := **any**( worklist )
  remove n from worklist
  Reach( n ) := $U_{m \in pred(n)}$ ReachOut( m )
  ReachOut( n ) := Reach( n ) \ kill( n ) ∪ gen( n )
  **if** ReachOut( n ) has changed **then**
    worklist := worklist ∪ succ( n )
  **end**
**end**

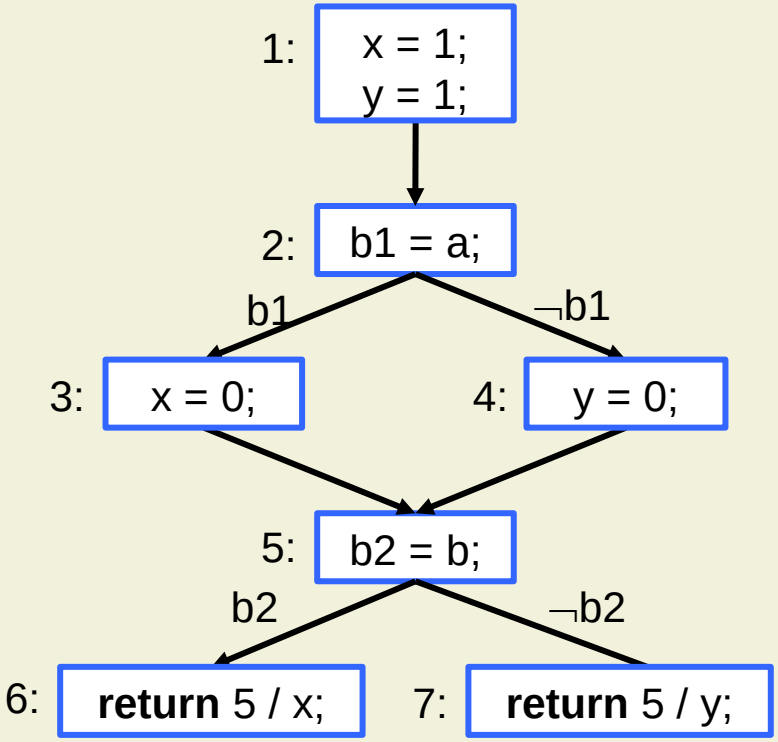
# Reaching Definitions: Example




| n | Reach( n ) | ReachOut( n ) |
|---|---|---|
| 1 | ∅ | |
| 2 | $x_1$, $y_1$ | $x_1$, $y_1$ |
| 3 | $x_1$, $y_1$ | $x_3$, $y_1$ |
| 4 | $x_1$, $y_1$ | $x_1$, $y_4$ |
| 5 | $x_1$, $x_3$, $y_1$, $y_4$ | $x_1$, $x_3$, $y_1$, $y_4$ |
| 6 | $x_1$, $x_3$, $y_1$, $y_4$ | $x_1$, $x_3$, $y_1$, $y_4$ |
| 7 | $x_1$, $x_3$, $y_1$, $y_4$ | $x_1$, $x_3$, $y_1$, $y_4$ |

# From Reaching Definitions to DU-Pairs

- The set of DU-pairs is easily determined as
  { (d,u) | u is a variable use for v and $v_d \in$ Reach(u) }




| n | Reach( n ) |
|---|---|
| 1 | ∅ |
| 2 | $x_1$, $y_1$ |
| 3 | $x_1$, $y_1$ |
| 4 | $x_1$, $y_1$ |
| 5 | $x_1$, $x_3$, $y_1$, $y_4$ |
| 6 | $x_1$, $x_3$, $y_1$, $y_4$ |
| 7 | $x_1$, $x_3$, $y_1$, $y_4$ |

- DU-pairs for x:
  (1,6), (3,6)
- DU-pairs for y:
  (1,7), (4,7)

# Data Flow Testing Example

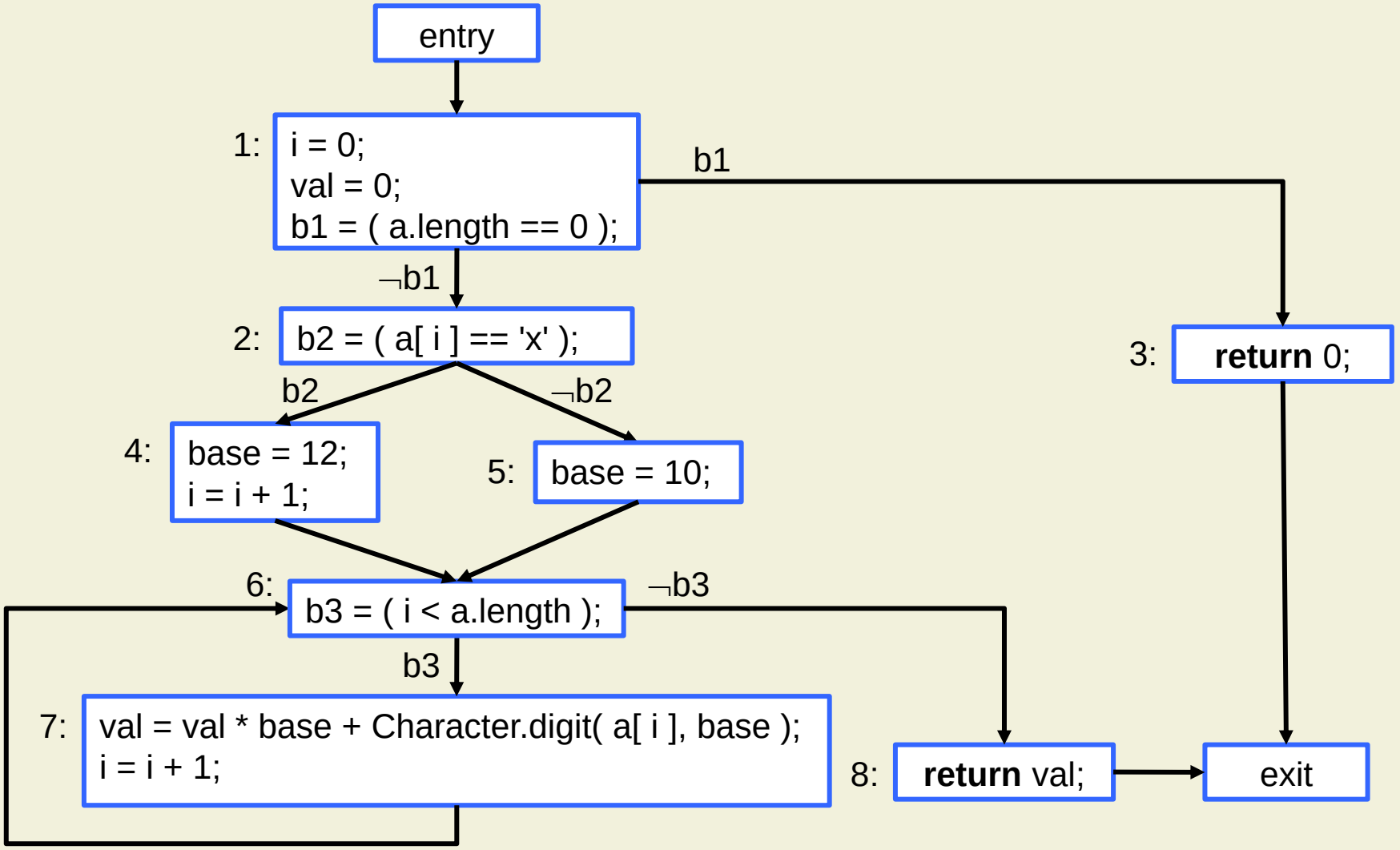
- Convert character sequence to integer
  - Input format: $d_{dec}$* | ‘x’($d_{hex}$*), where d is a (decimal or hexadecimal) digit

```
static int convert( char[ ] a ) {
  int base;  int i = 0;  int val = 0;
  if ( a.length == 0 )  return 0;
  if( a[ i ] == 'x' ) { base = 12; i = i + 1; }
  else { base = 10; }

  while( i < a.length ) {
      val = val * base + Character.digit( a[ i ], base );
      i = i + 1;
  }
  return val;
}
```

We assume here that all inputs are of the required format

# Data Flow Testing Example: CFG



# Data Flow Testing Example: DU-Pairs

| n | Reach( n ) | ReachOut( n ) |
|---|---|---|
| 1 | ∅ | $i_1$, $val_1$ |
| 2 | $i_1$, $val_1$ | $i_1$, $val_1$ |
| 3 | $i_1$, $val_1$ | $i_1$, $val_1$ |
| 4 | $i_1$, $val_1$ | $i_4$, $val_1$, $base_4$ |
| 5 | $i_1$, $val_1$ | $i_1$, $val_1$, $base_5$ |
| 6 | $i_1$, $i_4$, $i_7$, $val_1$, $val_7$, $base_4$, $base_5$ | $i_1$, $i_4$, $i_7$, $val_1$, $val_7$, $base_4$, $base_5$ |
| 7 | $i_1$, $i_4$, $i_7$, $val_1$, $val_7$, $base_4$, $base_5$ | $i_7$, $val_7$, $base_4$, $base_5$ |
| 8 | $i_1$, $i_4$, $i_7$, $val_1$, $val_7$, $base_4$, $base_5$ | $i_1$, $i_4$, $i_7$, $val_1$, $val_7$, $base_4$, $base_5$ |

- We get 14 DU-pairs
- DU-pairs for i:
  (1,2), (1,4), (1,6), (4,6), (7,6), (1,7), (4,7), (7,7)
- DU-pairs for val:
  (1,7), (7,7), (1,8), (7,8)
- DU-pairs for base:
  (4,7), (5,7)

# Data Flow Testing Example: Bug

- Consider the test cases
  - a = { }
  - a = { ‘x’ }
  - a = { ‘1’ }
  - a = { ‘1’, ‘2’ }
- The bug is not detected!

```
static int convert( char[ ] a ) {
  int base;  int i = 0;  int val = 0;
  if ( a.length == 0 )  return 0;
  if( a[ i ] == 'x' ) { base = 12; i = i + 1; }
  else { base = 10; }
  while( i < a.length ) {
    val = val * base + Character.digit( a[ i ], base );
    i = i + 1;
  }
  return val;
}
```

- Branch and loop coverage: 100%
- DU-pairs missed: (4,7) for i, base (coverage 86%)

# Data Flow Testing Example: Observation

- DU-pairs for i and val include (7,7)
- Complete DU-pairs coverage requires more than one loop iteration

```
static int convert( char[ ] a ) {
  int base;  int i = 0;  int val = 0;
  if ( a.length == 0 )  return 0;
  if( a[ i ] == 'x' ) { base = 16; i = i + 1; }
  else { base = 10; }

  while( i < a.length ) {
     val = val * base + Character.digit( a[ i ], base );
     i = i + 1;
  }
  return val;
}
```

# Determining all DU-Pairs: Heap Structures

```
static void repeat( int[ ] from, int[ ] to ) {
  int i = 0;
  if ( from.length == 0 )  return;
  while( i < to.length ) {
    to[ i ] = to[ i ] + from[ i % from.length ];
    i = i + 1;
  }
}
```

- Determining whether a definition and a usage refer to the same heap location, a static analysis would need arithmetic and aliasing information
- Static analysis has to over-approximate

# Measuring DU-Pairs Coverage

- Keep track of currently active definitions
  - defCover: Variable → Block
- Keep track of executed DU-pairs
  - useCover: Variable × $\text{Block}_{\text{def}}$ × $\text{Block}_{\text{use}}$ → $\mathbb{N}$
- Maps can be encoded as arrays, indexed by identifiers for variables and basic blocks

# Measuring DU-Pairs Coverage: Example

```
int foo( boolean a, boolean b ) {
  int x = 1;  defCover[ "x" ] = 0;
  int y = 1;  defCover[ "y" ] = 0;
  if( a ) {
    x = 0; defCover[ "x" ] = 1;
  } else {
    y = 0; defCover[ "y" ] = 2;
  }
  if( b ) {
    useCover[ "x", defCover[ "x" ], 3 ]++;
    return 5 / x;
  } else {
    useCover[ "y", defCover[ "y" ], 4 ]++;
    return 5 / y;
  }
}
```

Current variable definition for x is basic block 0

Current variable definition for x is basic block 1

DU-pair for variable x with current definition and use-block 3 has been executed

# Data Flow Testing: Discussion

- Data flow testing complements control flow testing
  - Choose test cases that maximize branch and DU-pairs coverage
- Like with path coverage, not all DU-pairs are feasible
  - Static analysis over-approximates data flow
  - Complete DU-pairs coverage might not be possible

# Data Flow Testing: Discussion (cont'd)

- DU-pairs coverage is not the only adequacy criterion for data flow testing
  - All definitions, all predicate-usages, all simple-DU-paths, etc.

- DU-pair "anomalies" may point to errors
  - Use before definition (not possible for locals in Java)
  - Double definition without use
  - Termination after definition without use

# 8. Test Case Selection

# Interpreting Coverage

- High coverage does not mean that code is well tested
  - But: low coverage means that code is not well tested
  - Make sure you do not blindly optimize coverage but develop test suites that test the code well

- Coverage tools do not only measure coverage metrics, they also identify which parts of the code have not been tested

# Experimental Evaluation: Approach

- Several studies investigate the benefit of coverage metrics
  - Andrews et al.: “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”, TR SCE-06-02, 2006

- Approach
  - Seed defects in the code
  - Develop test suites that satisfy various coverage criteria
  - Measure how many of the seeded defects are found by the test suits
  - Extrapolate to “real” defects in the code

# Experimental Evaluation: Some Findings

- The test suite size grows exponentially in the coverage
- More demanding coverage criteria lead to larger test suites, but do not detect more bugs
  - Block, decision, data flow coverage
- There is no significant difference in the cost-efficiency of the various coverage metrics
- All adequacy criteria lead to test suites that detect more bugs then random testing, especially for large test suites