Software Testing Approaches

Prof. Alex Bardas

Software Testing Approaches

- **White-box testing**
  - Choose test data with knowledge of the implementation
  - Test if internal operations are performed according to specifications
  - Test if internal components have been adequately exercised
- **Black-box testing**
  - Analyze a running program by probing with various inputs
  - Test if each function is fully operational
  - Search for errors in each function

White-box Testing

- Derive test cases to
  - Guarantee that all independent paths within a module will be executed at least once
  - Exercise all logical decisions on their true and false branches
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity
- When should we stop adding new test cases to our test set?
  - **Code coverage** measures the degree to which the source code of a program has been tested

Test Coverage

- What is a good test?
  - Has a high probability of finding an error
  - No (or minimal) redundancy

Objective: *to uncover errors*
Criteria: *in a complete manner*
Constraint: *with a minimum of effort and time*

- After we have done some testing, how do we know the testing is enough?

Test Coverage

- The most straightforward measure: **input coverage**
  - # of inputs tested / # of possible inputs → exhaustive testing
  - **Unfortunately, # of possible inputs is typically infinite, not feasible**

Test Coverage

- The power of a test suite is NOT determined by the # of test cases

```
int max(int x, int y)
{
    if (x > y)
      return x;
    else
      return x;
}
```

- **For 32-bit integer input**
  - **# of possible test cases =$2^{32} \times 2^{32} = 2^{64}$**
- **Which is better?**
  - Test set A: {(x=3,y=2), (x=2,y=3)}
  - Test set B:{(x=3,y=2), (x=4,y=3), (x=5,y=1)}

Test Coverage

- **Test coverage**
  - Measures the degree to which the specification or code of a software program has been exercised by tests

  - Code Coverage
  - Specification coverage
  - Model coverage
  - Error/fault coverage

Code coverage measures the degree to which the source code of a program has been tested

Code Coverage

- Definition:
  - Divide the code in to elements
  - Calculate the proportion of elements that are executed by the test suite
- Criteria
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Data flow coverage

Code Coverage

- **Statement Coverage**
  - Select a test suite and measure how many elementary statements in the program are executed by a test case

```
1. areTheyPositive(int x, int y) {
2.     if (x >= 0)
3.         print("x is positive");
4.  else print("x is negative");
5.    if (y >= 0)
6.         print("y is positive");
7.    else print("y is negative");
8. }
```

Two test sets T_1 and T_2

T_1 = {(x=12,y=5), (x= –1,y=35), (x=115,y=–13),(x=–91,y= –2)}

→ 8 out of 8 statements are covered

T_2 = {(x=12,y= – 5), (x= –1,y=35)}

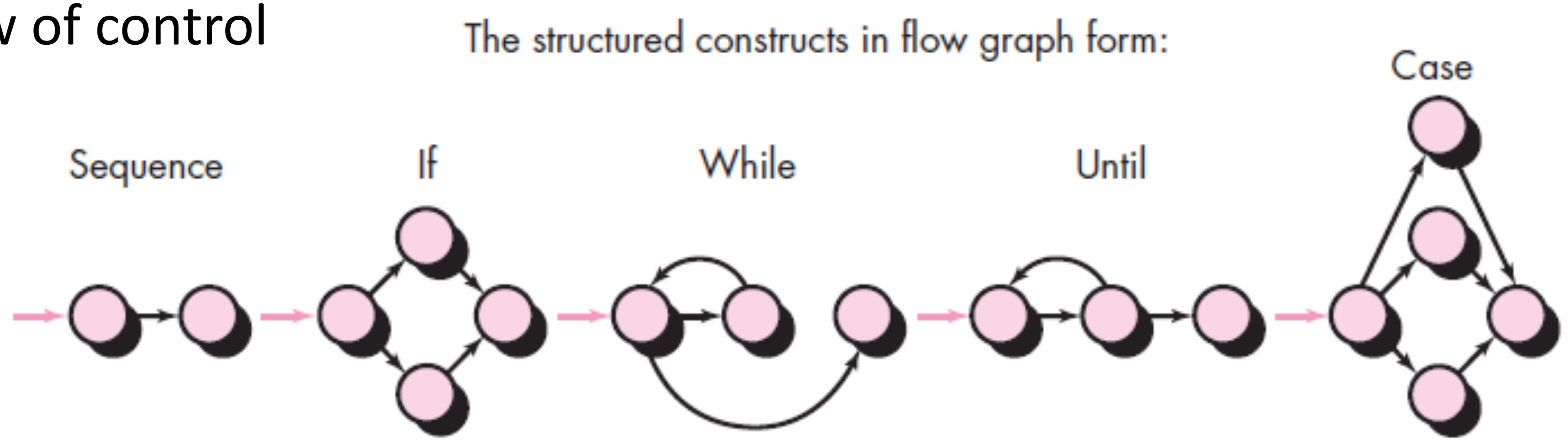
→ 8 out of 8 statements are covered, with a smaller test set

Code Coverage

- **Statement Coverage**
  - Select a test suite and measure how many elementary statements in the program are executed by a test case
- Statement Coverage in Practice
  - Most used in industry
  - Typically, we target 80-90% coverage
    - Microsoft reports 80-90% statement coverage
    - Safety-critical software must achieve 100% statement coverage
    - For large systems, 100% is usually very hard

Code Coverage

- **Branch Coverage** (edge coverage)
  - Every branch of the control flow is traversed at least once by some test case
- **Control Flow Graph**
  - Construct a control graph for a program
  - Node: procedural statements
  - Edge: flow of control



Code Coverage

- **Branch Coverage**
  - A branch is considered executed when ALL outcomes are executed

```
main() {
    int x, y, z, w;
    read(x);
    read(y);
    if (x != 0)
        z = x + 10;
    else
        z = 1;
    if (y>0)
        w = y / z;
    else
        w = 0;
}
```

$$\frac{\#\text{ of executed branches}}{\#\text{ of branches in the program}}$$

Code Coverage

- **Branch Coverage**
  - A branch is considered executed when ALL outcomes are executed

```
1.  main() {
2.      int x, y, z, w;
3.      read(x);
4.      read(y);
5.      if (x != 0)
6.          z = x + 10;
7.      else
8.          z = 1;
9.      if (y>0)
10.         w = y / z;
11.
12. }
```

Test cases:

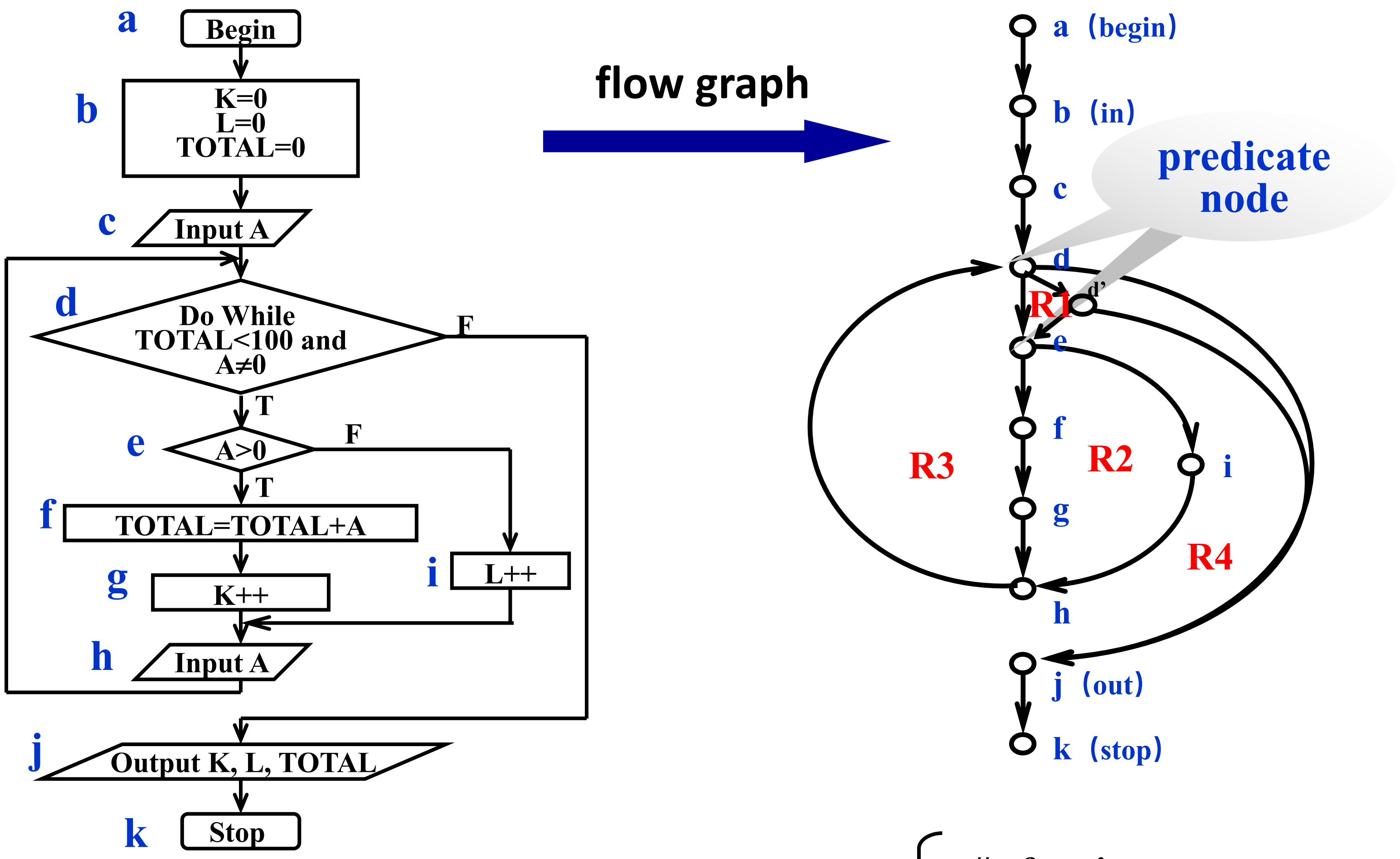
- (x = 1, y = 22)
- (x = 0, y = -10)
- (x = 0; y = 2)

Still doesn’t reveal the fault in statement 10

- (x = -10, y = 1)

Code Coverage

- **Path Coverage**
  - Try to cover all distinct paths through a program
  - The strongest code coverage criterion
  - However, usually not feasible
    - Exponential paths in acyclic programs
    - Infinite paths in some programs with loops
- Cyclomatic complexity?
  - # of linearly independent paths

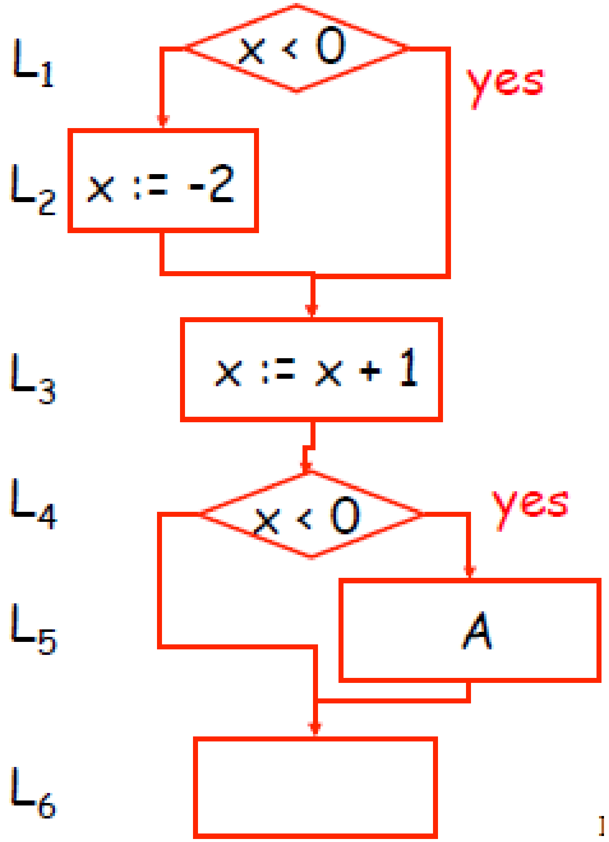


Cyclomatic Complexity =

$$\left\{\begin{array}{l} \text{\# of regions} \\ E - N + 2 = 14\text{-}12\text{+}2 = 4 \\ P + 1 = 3\text{+}1 = 4 \end{array}\right.$$

Code Coverage

- Path Coverage
  - N conditions → 2^N paths
  - Many are not feasible
    - $L_1L_2L_3L_4L_6$
- Need to determine independent paths
  - Count path #



branch coverage ≤ cyclomatic complexity ≤ # of paths

Code Coverage

- Path Coverage

```
1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 1;
9.     if (y>0)
10.        w = y / z;
10.    else
11.        w = 0;
12.}
```

Test cases:

- (x=1,y=22)
- (x= 0,y=10)
- (x=1, y=-22)
- (x=0, y=-10)

Still doesn't reveal the fault in statement 10

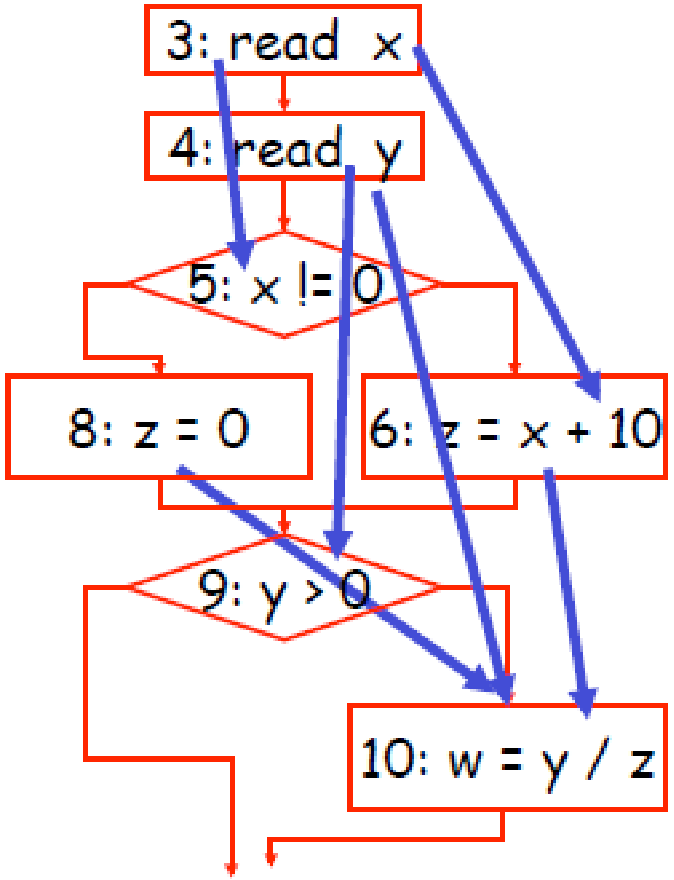
- x=-10, y=1
- this is an error that structural coverage cannot reveal

Data Flow Coverage

- Cover all **def-use pairs** in a software
  - def: write to a variable
  - use: read of a variable
  - use $\boldsymbol{u}$ and def $\boldsymbol{d}$ are paired
  - when d is the direct precursor of u in certain execution

$$\frac{\text{\# of executed def-use pairs}}{\text{\# of def-use pairs in the program}}$$

Not easy to locate all use-def pairs



Deriving Test Cases

- How to derive white-box test cases?
  - Use the design or code as a foundation and draw a control flow graph
  - Determine the cyclomatic complexity of the graph
  - Determine a basis set of linearly independent paths
  - Prepare test cases that will force execution of each path in the basis set

Code Coverage: In Practice

- Code coverage is the most widely used technique for test evaluation
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Data flow coverage
- Far from perfect
  - A lot of corner cases can never be found
  - 100% code coverage is rarely achieved
    - Some commercial software is released with around 60% code coverage
    - Many open source software: even lower than 50% code coverage

Black-box Testing

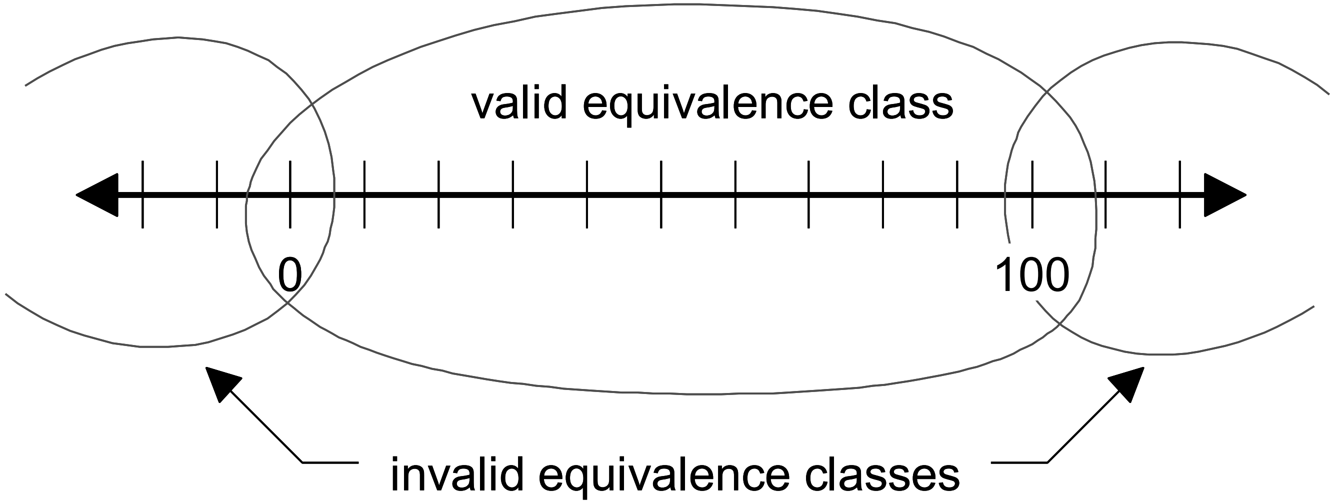
- A.k.a., **functional testing** or **behavioral testing**
  - To demonstrate software operations are based on specified functions without regard for its internal logic
  - Tests are designed to answer:
    - How is ***functional*** validity tested?
    - How is system ***behavior and performance*** tested?
    - What ***classes of input*** will make good test cases?
    - Is the system particularly ***sensitive*** to certain input values?
    - How are the ***boundaries*** of a data class isolated?
    - What data rates and data volume can the system ***tolerate***?
    - What effect will specific ***combinations*** of data have on system operation?

Equivalence Testing

- **Equivalence testing**
  - Divides the space of all possible inputs into ***equivalence groups***
    - Requirements: *coverage, disjointedness, representation*
  - Assume the software behaves similarly for all inputs from an equivalence group
  - Therefore, we can reduce the # of test cases
- Two steps:
  1. Partition the values of input parameters into equivalence groups
  2. Choose the test input values

Equivalence Testing

- **Equivalence testing**
  - How to partition?



- If the input parameter specifies a **range** of values
  - Partition into one valid and two invalid equivalence classes

Equivalence Testing

- **Equivalence testing**
  - If the input parameter specifies a **single value**
    - Partition into one valid and two invalid equivalence classes
    - `{x<1.4},{x = 1.4},{x>1.4}`
  - If the input parameter specifies a **set** of values
    - Partition into one valid and one invalid equivalence class
    - `{1, 3, 5}, {other integers}`
  - If the input parameter specifies a **Boolean value**
    - Partition into one valid and one invalid equivalence class
    - `{true}, {false}`
  - Multiple parameters may involve a combination (cross product)
    - <room, key>

Boundary Testing

- **Boundary testing**
  - A special case of equivalence testing
  - Focuses on the boundary values of input parameters
    - Special cases at the boundary of equivalent classes are often overlooked
- Often, test
  - Elements from the *edges* of the equivalence class
  - Elements from the *outliers*
    - zero, min/max values, empty set, empty string, null
  - Confusion between > and ≥

Object-Oriented Software

- Initially, we hoped it would be easier to test OO software than procedural software
  - Soon it became clear that this is not true
  - Some of the older testing techniques are still useful
  - New testing techniques are designed specifically for OO software
- In OO, the smallest testable unit is a **class**
    - A method is similar to a procedure, but it is part of a class
    - Method is tightly coupled with other methods and fields in the class

Class Testing

- Traditional black-box and white-box techniques still apply!
  - e.g. testing with boundary values
- Inside each method
  - Obtain at least 100% branch coverage
  - Cover all def-use (DU) pairs inside a method (intra-method)
- DU pairs that cross method boundaries (inter-method)
  - e.g.:
    - inside method m1, field f is assigned a value
    - inside method m2, this value is read

Class Testing

- **Example: Inter-method DU Pairs**

```
class A {
        private int index;
        public void m1() {
            index = …;
            …
           m2();
        }
        private void m2() { … x = index; … }
        public void m3() { … z = index; … }
}
```

test 1:
- **call m1, which writes index**
- **then calls m2, which reads the value of index**

test 2:
- **call m1, and then call m3**

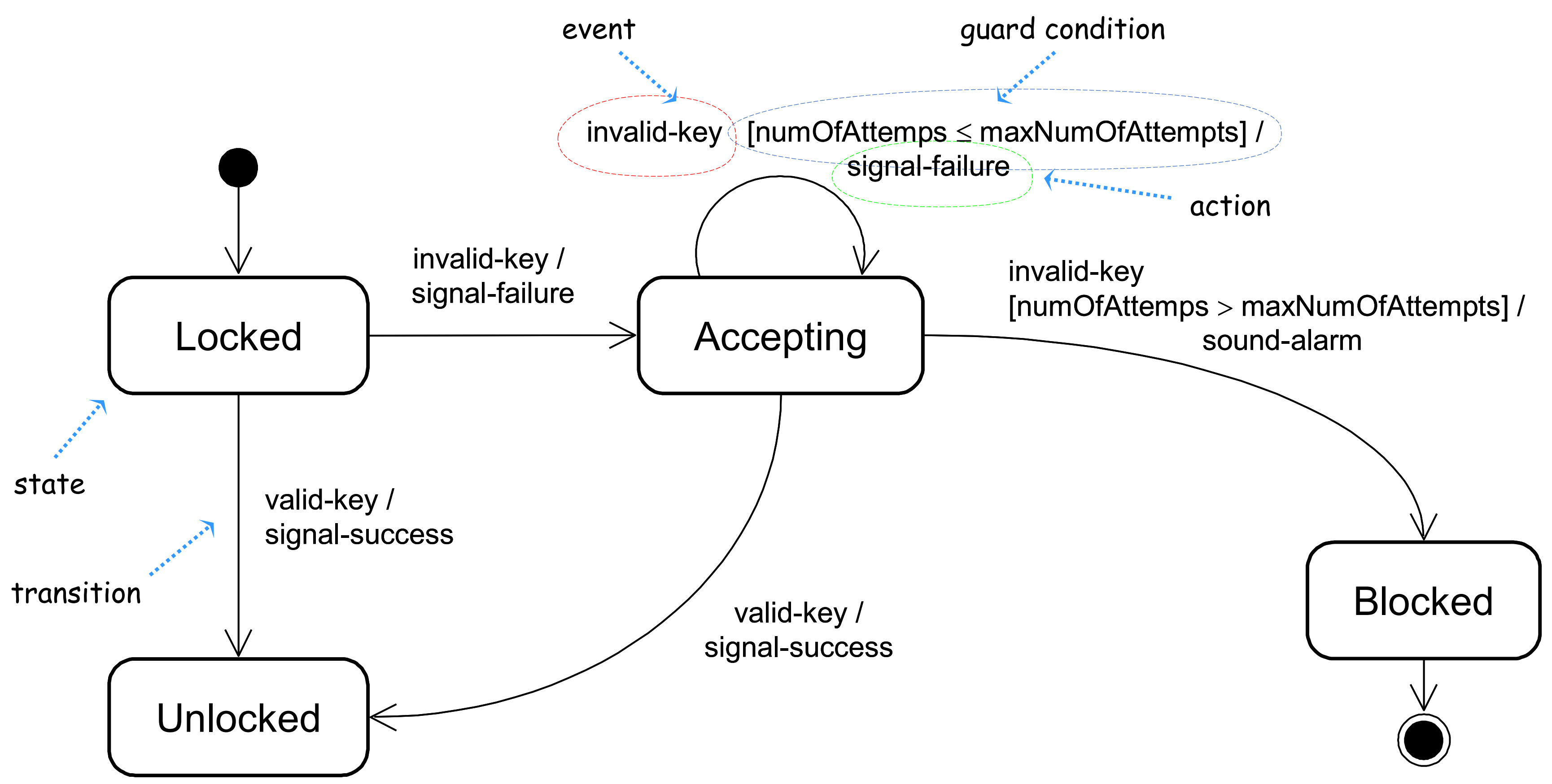
State-based Testing

- **State-based testing**
  - The behavior of software depends on the object state
  - The state of an object is defined as a constraint on the values of the object’s attributes
  - Defines a set of abstract states that a software unit can take
  - Tests the unit’s behavior by comparing its actual states to the expected states
  - Highly depend on the use of the state diagram

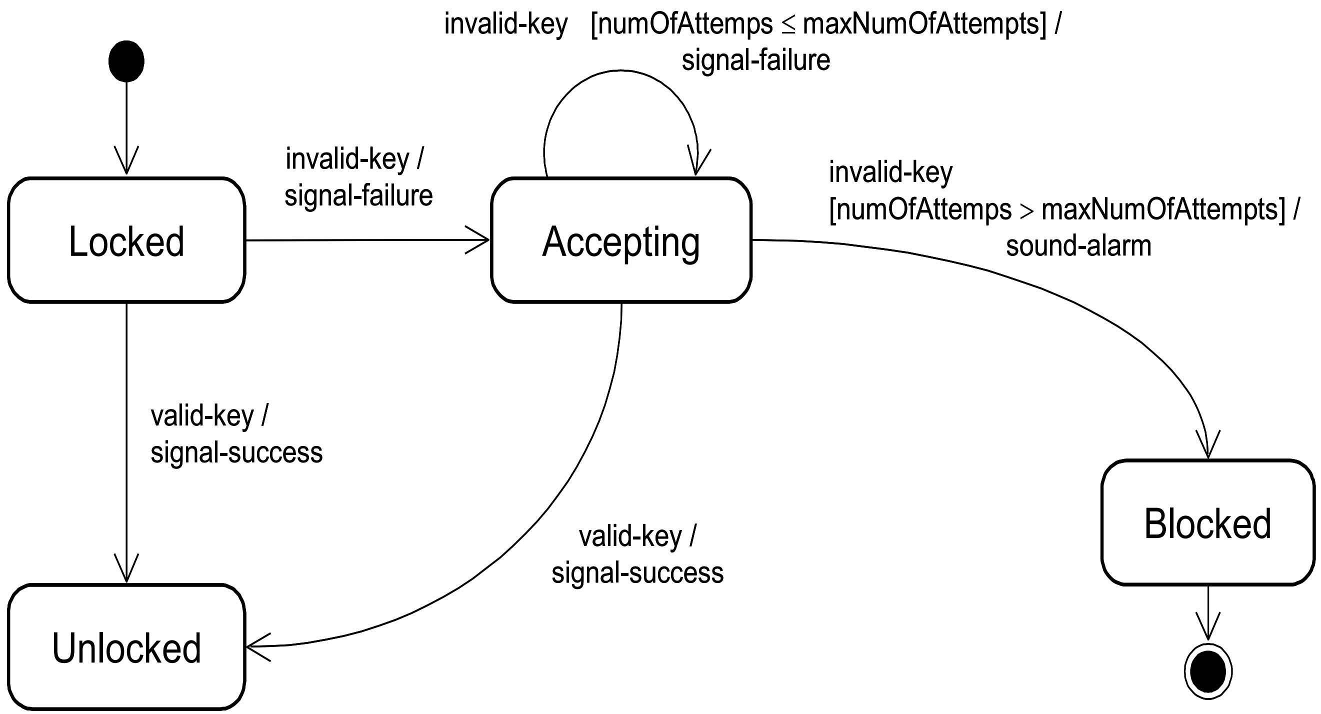
State-based Testing

- **State-based testing**
  - Derive the state diagram for the tested unit
    - Define states, and possible transitions, triggering events (methods)
    - Choose test values for each state
  - Initialize the unit and run the test
    - Similarly, implement the test driver
    - Finish executing, compare the actual state with expected state

State-based Testing



State-based Testing



To ensure state coverage:

- ✓ Cover all identified states at least once
- ✓ Cover all valid transitions at least once
- ✓ Trigger all invalid transitions at least once

Five valid transitions:

{ Locked→Unlocked, Locked→Accepting, Accepting→Accepting, Accepting→Unlocked, Accepting→Blocked }

References

- Prof. Fengjun Li's EECS 448 Fall 2015 slides
- This slide set has been extracted and updated from the slides designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill 2014) by Roger Pressman