# Software Testing

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

Spring 2022

## Lecture 6

# Outline

# Outline

1. Graph coverage III
   - Design artifacts
   - Specifications: testing sequence constraints
   - Specifications: testing state behavior
   - Specifications: use cases

# Graph coverage for design elements

- The most common graph for structural design testing is the call graph.

**Definition**

A call graph of a program is a graph where nodes are methods and edges represent calls.

- Structural testing
  - Node coverage, edge coverage
  - Others ...
- Data-flow testing
  - Caller-callee

# Node and edge coverage

- Node coverage: call every method at least once.
- Edge coverage: execute every call at least once.
- Problem: call graph often very flat

```
class Stack {
   // ...
   public void push (Object o)
   public Object pop ( )
   public boolean isEmpty (Object o)
}
```

- Other graphs?
  - Inheritance graph? Not executable.
  - Could require creation of an object. Weak criterion (still no execution).

# Call coverage

- The following criteria assume an inheritance graph and define coverage criteria that require execution.

**Definition**

A TR has OO call coverage if it contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.

**Definition**

A TR has OO object call coverage if it contains each reachable node in the call graph of every object instantiated for each class in the class hierarchy.

- Possibly many objects.
- Not really used in practice.

# Data-flow testing

At design level, data-flow testing is more interesting.

- Data-flow coupling more complex:
  - Different names for actual and formal parameter
  - Sharing

  Notation:

**Definition**

The method (or unit) that invokes another method (or unit) is called the caller; the invoked method is called the callee. The statement or node that makes that call is called the call site.

  - Parameters at the caller's site are called formal parameters; parameters (variables) at the callee's site are called actual parameters.
- def-use pairs are represented as pairs of triples (method, var, statement).

# Example (call site)

```
A                              // caller
   ...
   Z = B(X)                 // call site, actual parameter X
   ...
end A

B(Y)                       // callee, formal parameter Y
   ...
end B
```

- Considering all def-use pairs between units is too expensive
- Instead: focus on the interface

# Interprocedural DU pairs

**Definition**

A node is a last-def node if it defines a variable $x$ and has a def-clear path from that node through a call site to a use of $x$.

- Both directions: from caller to callee, from callee to caller

**Definition**

A node $n$ is a first-use node for a variable $x$ if it uses $x$ and has a def-clear path and use-clear path from the call site to that node (if $n$ in caller) or from the callee's entry to that node (if $n$ in callee).

- A path from $n_i$ to $n_j$ is use-clear for a variable $x$ if $x \notin \text{use}(n_k)$ for every node $n_k$, $k \neq i, j$ on that path.
- The variable $x$ can obtain its value through parameter passing, return statements, or shared data.

# Example (interprocedural DU pairs)

Caller F

```
x = 14       // last-def, 1
...
y = G(x)     // call site

print(y)     // first-use, 2
```

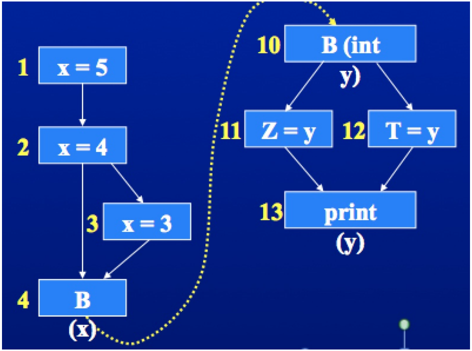
Callee G(a)

```
print(a)     // first-use, 1
...
b = 42       // last-def, 2
...
return b
```

Two interprocedural pairs

- (F, x, x=14) - (G, a, print a)
- (G, b, b=42) - (F, y, print y)

# Example: interprocedural DU pairs

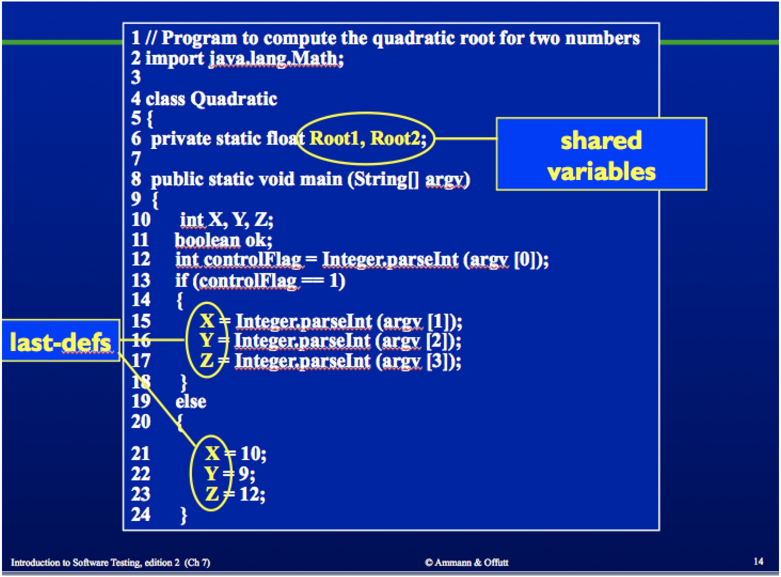



- Last-defs for x: 2,3; for (Z,T): 11,12
- First-uses for y: 11,12
- DU pairs
  - (A,x,2) - (B,y,11), (A,x,2) - (B,y,12), (A,x,3) - (B,y,11), (A,x,3) - (B,y,12)

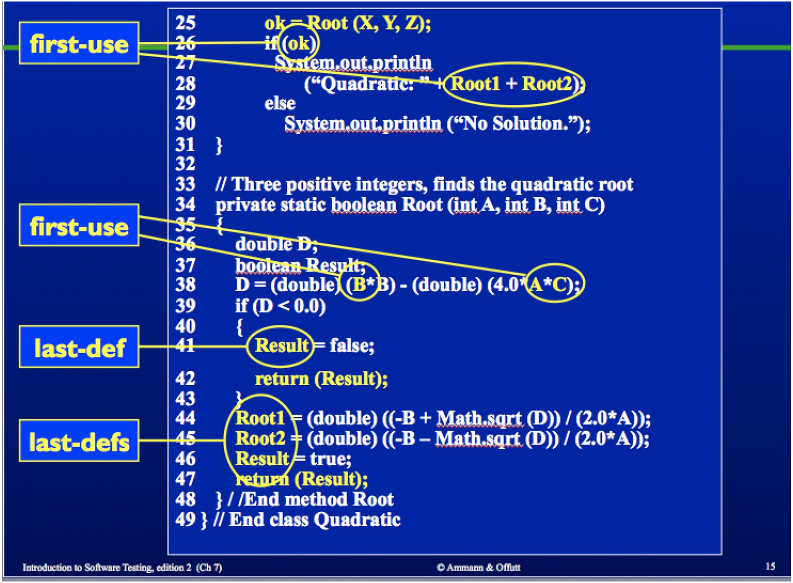
# Another example: interprocedural DU pairs

```
// Program to compute the quadratic root for
two numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
    int X, Y, Z;
    boolean ok;
    int controlFlag = Integer.parseInt(argv[0]);
    if (controlFlag == 1)
    {
        X = Integer.parseInt (argv[1]);
        Y = Integer.parseInt (argv[2]);
        Z = Integer.parseInt (argv[3]);
    }
    else
    {
        X = 10;
        Y = 9;
        Z = 12;
    }
        ok = Root (X, Y, Z);
        if (ok)
          System.out.println
             ("Quadratic: " + Root1 + Root2);
        else
           System.out.println ("No Solution.");
 }

// Three positive integers, finds quadratic root
 private static boolean Root (int A, int B, int C)
 {
    double D;
    boolean Result;
    D = (double) (B*B) -  (double) (4.0*A*C );
    if (D < 0.0)
    {
       Result = false;

       return (Result);
    }
    Root1 = (double) ((-B + Math.sqrt(D))/
(2.0*A));
    Root2 = (double) ((-B – Math.sqrt(D))/(2.0*A));
    Result = true;
    return (Result);
 } // End method Root
} // End class Quadratic
```

# Another example (cont'd)



# Another example: (cont'd)



# Another example: (cont'd), DU pairs

# Discussion

- We considered only variables that are used or defined in the caller.
  - Class and global variables are assumed to be initialized.
  - No transitivity (too expensive)
  - Arrays are considered to be one element.
- Two kinds of def-use pairs
  - intra-procedural
  - inter-procedural

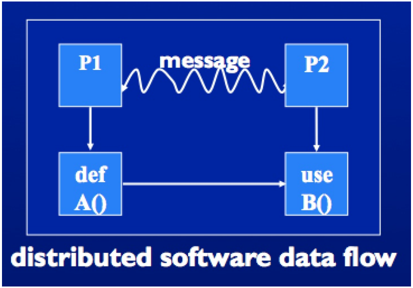
# Refinements

- The notion of DU pairs could be further refined.
- OO DU pairs: require def and use to be executed from the same object

```
begin F
...

   begin A
   ..          // def
   end A

   begin B
   ..          // use
   end B

end F
```

# Refinements (2)

- Web applications, distributed software




where messages could be HTTP, RMI, . . . protocols

- May consider those def/use points as advanced kinds of DU pairs.
- Identifying def-clear paths and test paths not trivial.

# Interprocedural data-flow coverage criteria

- A coupling du path for a variable $x$ is a path from a last-def of $x$ to a first-use of $x$.
- Coverage criteria
  - All-Coupling-Defs coverage: for every last-def, at least one path to a first-use is executed.
  - All-Coupling-Use coverage: for every last-def, at least one path to every first-use is executed.
  - All-Coupling-DU-Path coverage: for every last-def, every path to a first-use is executed.
- As before, All-Coupling-DU-Path coverage may be satisfied by paths with sidetrips.

# Outline

1. Graph coverage III
   - Design artifacts
   - Specifications: testing sequence constraints
   - Specifications: testing state behavior
   - Specifications: use cases

# Graph coverage for specifications

- A specification (formally) defines the expected functional and non-functional requirements.
  - Sometimes called model.
  - Abstraction from the implementation
- We look into two forms of behavioral specification:
  - Sequencing constraints on class methods
  - Descriptions of states and state transitions

# Sequencing constraints

**Definition**

Sequencing constraints are rules that determine in which order methods may be called.

- Constraints are tested by sequences of method calls.
- Encoding of constraints?
  - Explicit, e.g., in pre-conditions
  - Implicit, e.g., in pre-conditions
  - Not at all. Then the tester needs to derive them (documentation, implementation, ask developers)

# Example (sequencing constraints)

```
public int deQueue()
{
   // Pre: At least one element must be on the queue.
   ..
}
public enQueue (int e)
{
   // Post: e is on the end of the queue.
   ...
}
```

- Here, the sequence constraint is implicitly encoded (comment)
  - `enQueue` must be called before `deQueue`.
- Left unspecified: must have at least as many `enQueue`s as `deQueue`s.
  - Could be handled by state behavior techniques (see below).

# Example (sequencing constraints): ADT File

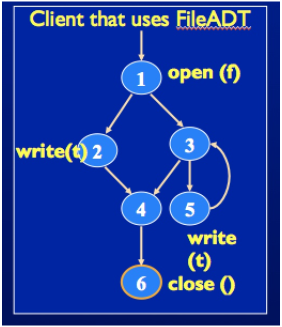
```
class FileADT
{
   open (String fName) // Opens file with name fName
   close () // Closes the file and makes it unavailable
   write (String textLine) // Writes a line of text to the file
}
```

Sequencing constraints

- An `open` must be executed before every `write`.
- An `open` must be executed before every `close`.
- A `write` may not be executed after a `close` unless there is an `open` between.
- A `write` should be executed before every `close`.

# Checking the constraints

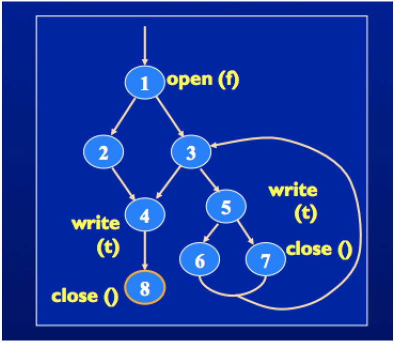
Assume the following client application:




- Is there a path to a *write* that does not go through an *open*?
- Is there a path to a *close* that does not go through an *open*?
- Is there a path from a *close* to a *write*?
- Is there a path from an *open* to a *close* that does not go through a *write*?
- Violation detectable in path [1,3,4,6]

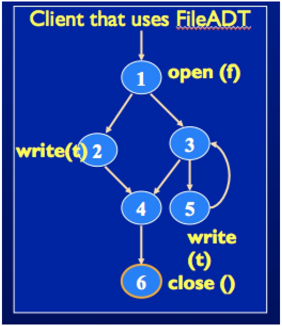
# Another example (checking the constraints)

Assume the following client application:




- Violation: [7,3,4]
- Close before write

# Defining test requirements



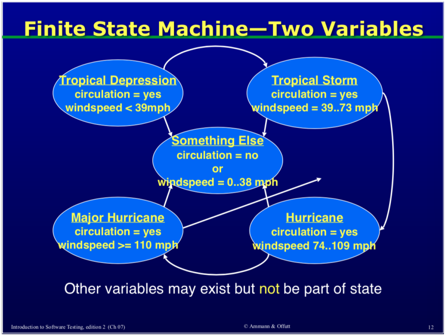

- Violation in path [1,3,4,6]
- Try to execute this path.
- What if the program does not allow taking th path?
    - Recall: the question is undecidable

# Test requirements for FileADT

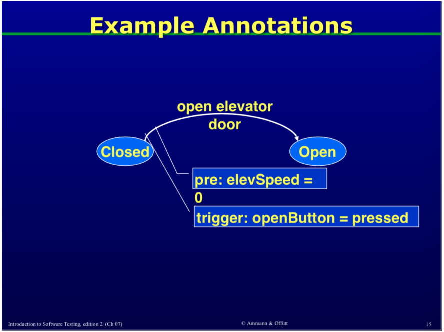
- Formulate test requirements that aim at violating sequence constraints:
  - Cover every path from the start node to every node that contains a *write* such that the path does not go through a node containing an *open*.
  - Cover every path from the start node to every node that contains a *close* such that the path does not go through a node containing an *open*.
  - Cover every path from every node that contains a *close* to every node that contains a *write*.
  - Cover every path from every node that contains an *open* to every node that contains a *close* such that the path does not go through a node containing a *write*.
- In a correct program all test requirements are infeasible.

# Outline

# Example (finite state machine)




- FSM specification of a weather system with 5 states
  - Note: missing transition from state “Something Else”; dangling transition from “Major Hurricane”
  - Variables “circulation” and “windspeed” define the different states and their transitions




- A FSM with two states
- Variables “elevSpeed” and “openButton” associated

# Testing state behavior

- A finite state machine (FSM) can be used to model how variables change their state during program execution.
  - Nodes represent states.
    A state captures, at a given point, the values of program variables.
  - Edges represent transitions from one state to a changed state.
- Applications:
  - Embedded software, protocols (network, web)
  - Compilers (parser)
  - Abstract data types, classes
- Modeling languages exist with different characteristics.
  - Graphical: UML state charts, Petri Net
  - Formal: Automata

# Annotations on FSMs

- FSMs can be annotated with actions.
  - Entry actions to a node, exit action from a node
  - Actions on edges
- FSMs can also be annotated with predicates and conditions
  - Preconditions (guards): boolean conditions that must be true if a transitions can be taken
  - Triggering events: changes to variables that cause transitions to be taken

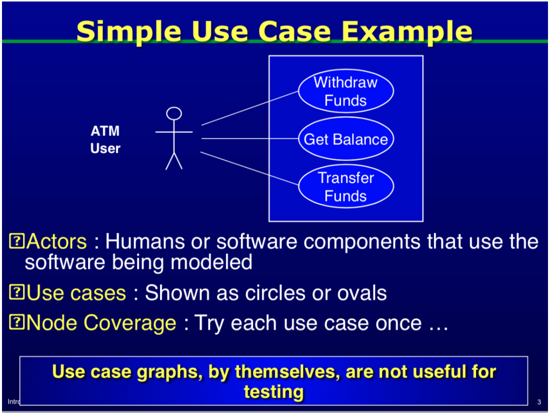
# Interpreting coverage of FSMs

- Structural criteria
  - Node coverage: execute every state
  - Edge coverage: execute every transition
  - Edge-pair coverage: execute every transitions pair
- Data-flow criteria in practice
  - Annotations needed (def/use)
  - Triggers contain def information, but DU path is short (next state)
  - Guards and actions may contain use and def information
  - Nodes may also contain use and def information
- Once an FSM is in place, graph-based testing is straightforward.

# Outline

1. Graph coverage III
   - Design artifacts
   - Specifications: testing sequence constraints
   - Specifications: testing state behavior
   - Specifications: use cases

# Requirements elicitation



# Use cases

- Uses cases are widely used to capture software requirements.
- Graph-based testing
  - Use-case diagrams can be considered graphs, but then the only applicable coverage criterion is node coverage.
  - But use cases also have a richer textual description (preconditions, postconditions, steps alternative flows).
  - For testing purposes, better to derive a graph from the textual description. Sometimes, such graph is already available as "activity diagram."

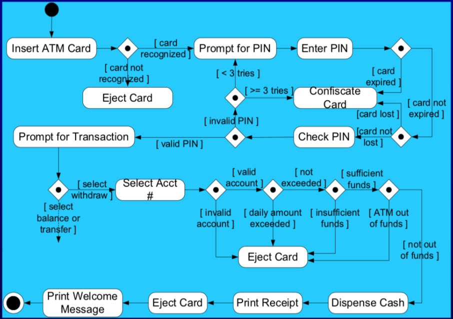
# Textual description

Source: K. Saleh, Software Engineering, Ross 2009.

**Table 3.5 Place Order use case**

| | |
|---|---|
| Identification | UC x |
| Name | Place Order |
| Created by<br>Date created<br>Updated by<br>Date of update | J. Smith, January 5, 2006 |
| Actors involved | Triggering actor: Buyer<br>Secondary (affected) actors: Finance and warehouse |
| Brief description | This use case is started by the buyer to construct electronic basket with books selected from the catalogue |
| Assumptions | Catalogue includes available books |
| Preconditions | Buyer could be a registered or unregistered individual user<br>Buyer could be a registered institutional user<br>Registered buyer has already logged on successfully |
| Postconditions | Electronic basket is created and closed prior to placing the order |
| Priority | High |
| Frequency of use | High (one hundred orders per day during the first year) |
| Flow of events (or steps) | 1. Browse book catalogue<br>2. Select books<br>2.1. Select a book and quantity<br>2.2. Confirm availability<br>2.3. Compute current total cost<br>2.4. Repeat 2.1 through 2.3 until desired books are purchased<br>3. Confirm order<br>4. Provide payment information and get confirmation of payment |

# ATM Withdraw Activity Graph




Introduction to Software Testing, Edition 2 (Ch 07) © Ammann & Offutt 10

# References

- AO, Ch. 7.4, 7.5. (7.6)