EE360T/382V Software Testing

khurshid@ece.utexas.edu

February 14, 2020

Overview

Now – Graph coverage for designs and specs

Last time – Graph coverage for source code

Next time – Logic coverage

EE360T/382V Software Testing

khurshid@ece.utexas.edu

Chapter 2*: Graph Coverage

*Introduction to Software Testing by Ammann and Offutt

2.4 Graph coverage for design elements

Focus: couplings – measure dependency relations between units based on interconnections

- Faults in one unit may effect the coupled unit

Call graph – nodes represent methods and edges represent method invocations

- Most commonly used graph for structural design coverage
- Can create several (possibly disconnected) call graphs based on different parts of the module

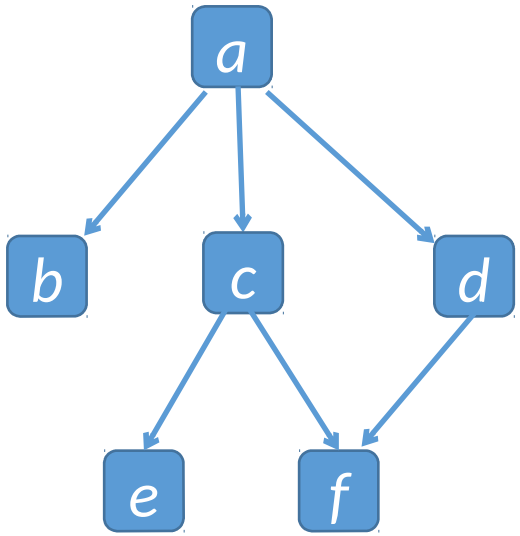
Call graph coverage

Method (node) coverage – each method be invoked at least once

Call (edge) coverage – each method invocation in code be executed at least once

Example:

- Edge coverage requires *f* to be executed at least twice



Inheritance and polymorphism

Focus: inheritance hierarchy

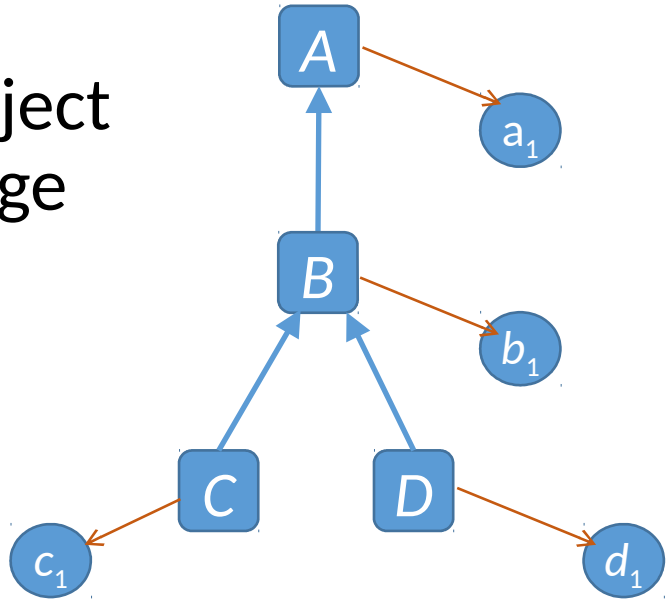
Coverage requires object creation

- By itself, it is weak

OO call coverage – create an object of each class and cover each edge of the call graph

- Can extend it to all objects created by the program

Criteria based on inheritance not used widely



Data flow for design elements

Data flow connections are often complex

- Defs and uses are in *different* units
  - Focus on *last* defs and *first* uses

Parameter coupling – parameters are passed in calls

Shared data coupling – two methods access the same data object as a global variable

External device coupling – two methods access the same external medium, e.g., file system

Parameter coupling terminology

Caller – method that invokes another method

Callee – method that is invoked

Call site – statement that makes the call

Actual parameter – value passed at call site

Formal parameter – variable in method declaration

Interface between two methods – mapping of actual to formal parameters

Coupling variable – variable whose value is used in another method

Coupling def-use criteria

D2.40 Last-def – set of nodes that define a variable for which there is a def-clear path from the node through the call site to a use in another method

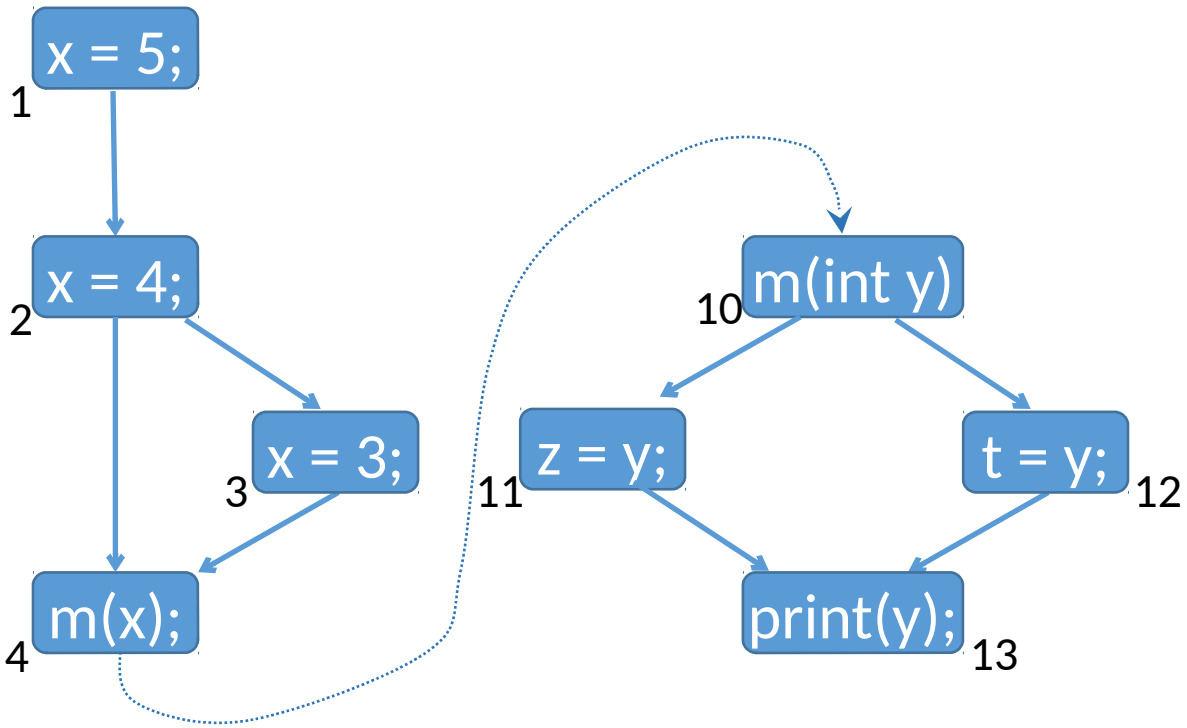
A path from location n_i to n_j is use-clear w.r.t. variable v if for every node n_k on the path where $k \neq i$ and $k \neq j$, v is not in $use(n_k)$

D2.41 First-use – set of nodes that have uses of a variable for which there is a def-clear and use-clear path from the entry point (if the use is in the callee) or the call site (if the use is in the caller) to the nodes

A coupling du-path is from a last-def to a first-use

- All def-use criteria from before can be adapted

Example



Last-defs = { 2, 3 }

First-uses = { 11, 12 }

Shared data coupling examples

Object-oriented (OO) direct coupling data flow

- *m*() calls *a*() and *b*()
  - *a*() defines
  - *b*() uses

OO indirect coupling data flow

- *m*() call *a*() and *b*()
  - *a*() calls *e*()
  - *b*() calls *f*()
    - *e*() defines
    - *f*() uses

2.5 Graph coverage for specs

A sequencing constraint imposes restrictions on the order in which methods may be invoked

- May be expressed implicitly or explicitly (or be missing)

Example of an implicit constraint:

```
public int dequeue() {
    // pre: queue is non-empty
... }
public void enqueue(int x) {
    // post: x is at the end of the queue
... }
```

- The only way *pre* of *dequeue* is satisfied is if *enqueue* is called at least once before *dequeue*

Checking using constraints

Sequencing constraints allow two forms of checking:

- Static checking where all paths in a client are checked for conformance to constraints
- Dynamic checking (testing) where test requirements consist of constraint violation
  - If a test meets a requirement, a bug is found

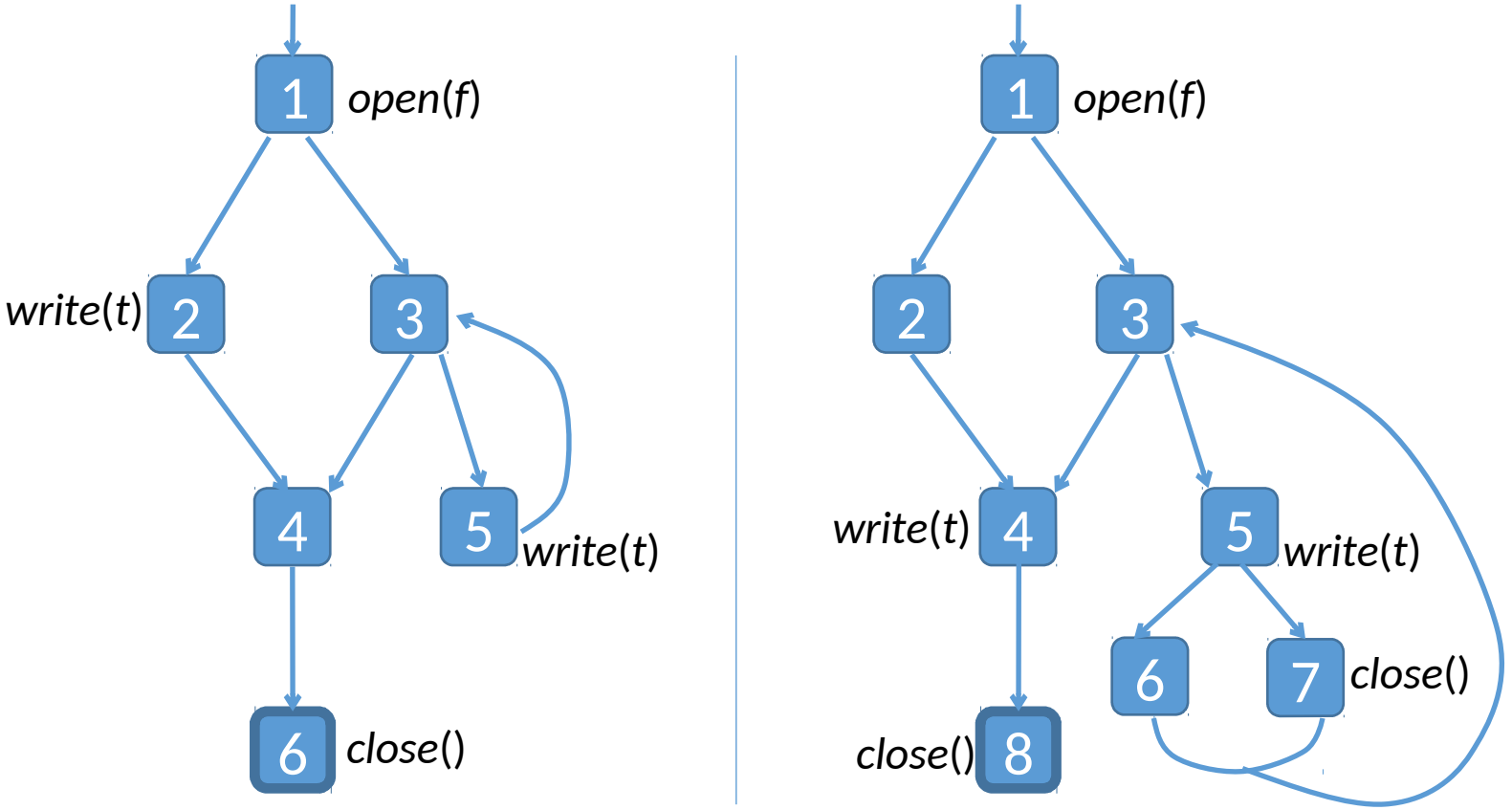
Example: file-ADT

- *open*(*String fName*) // opens file *fName*
- *close*() // closes the file
- *write*(*String text*) // writes *text* to file

Example file-ADT constraints

1. An *open*(*f*) must be executed before every *write*(*t*)
2. An *open*(*f*) must be executed before every *close*()
3. A *write*(*t*) must not be executed after a *close*() unless an *open*(*f*) appears in between
4. A *write*(*t*) must be executed before every *close*()
5. A *close*() must not be executed after a *close*() unless an *open*(*f*) appears in between
6. An *open*(*f*) must not be executed after an *open*(*f*) unless a *close*() appears in between

Example CFGs with file-ADT labels



Example test requirements

Try to get a test execution to violate a constraint, e.g.,

1. Cover every path from start to every *write*(*t*) node such that the path does not have an *open*(*f*) node
2. Cover every path from start to every *close*() node such that the path does not have an *open*(*f*) node
3. Cover every path from every *close*() node to every *write*(*t*) node such that the path does not have an *open*(*f*) node
4. …

For a correct client, all such requirements will be infeasible

?/!