SYSC 4101 / 5105

Integration Testing
Part I—What is it?

Introduction

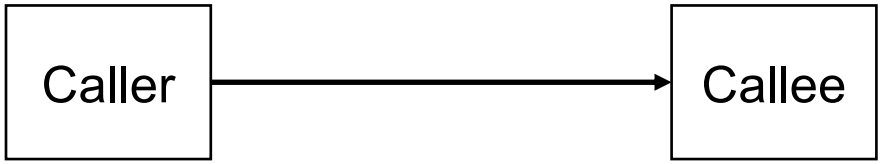
- Elements of a computer program cooperate
  - One function calls another
  - One class' methods call another class' methods
  - One subsystem relies (delegates requests to) on another subsystem
- Cooperating elements can (often) be tested in isolation
  - Refer to coming discussion on test scaffolding
- Elements that have passed their (isolated) tests may not work together
  - Mismatching interfaces, mismatching interaction protocols …
- Testing elements cooperation is necessary
  - This is integration testing

Coupling-based Criteria

- Refer to the testing of interfaces between units / modules to assure they have consistent assumptions and communicate correctly.
  - Coupling between two units measures the dependency relations between two units by reflecting the interconnections between units; faults in one unit may affect the coupled unit (Yourdon and Constantine, 1979)
- Assumption:
  - The units integration tested have passed their unit tests
- General Principle
  - Each module to be integrated should pass an isolated (unit) test
  - Integration testing must be performed at a higher level of abstraction – looking at program as atomic building blocks and focusing on their interconnections
  - Goal: *Guide* testers during integration testing, help define a criterion to determine when to *stop* integration testing

Basic Definitions

- The *interface* between two units is the mapping of actual to formal parameters
- Technique ensures that variables defined in *caller* units are appropriately used in *callee* unit
- Look at variables definitions *before* calls (i.e., before the call sites) and returns to other units, and uses of variables just *after* calls and returns from the called unit
- Solution based on three types of coupling paths:
  - **Parameter coupling**
  - Shared data coupling (i.e., non local variables shared by several modules)
  - External device coupling (i.e., references of several modules to the same external device)



Basic Definitions (cont.)

- *Call sites*:
  - statements in caller (A) where callee (B) is invoked
- *Coupling-def*:
  - A coupling-def is a node that contains a definition in a unit that can reach a use in another unit on at least one execution path.
- *Coupling-use*:
  - A coupling-use is a node that contains a use in a unit that can be reached by a definition in another unit on at least one execution path.

Basic Definitions (cont.)

- *Last-Defs*:
  - Last definition before call
    - The set of nodes (in the caller's CFG) that define x for which there is a def-clear path from the node to the call site.
  - Last definition before return
    - The set of nodes (in the callee's CFG) that define x for which there is a def-clear path from the node to the return site.
- *First-Uses*:
  - First use in callee
    - The set of nodes (in the callee's CFG) that have uses of y and for which there exists a def-clear and use-clear path between the entry point and the nodes.
  - First use in caller
    - The set of nodes (in the caller's CFG) that have uses of y and for which there exists a def-clear and use-clear path between the callsite and the nodes.

Basic Definitions (cont.)

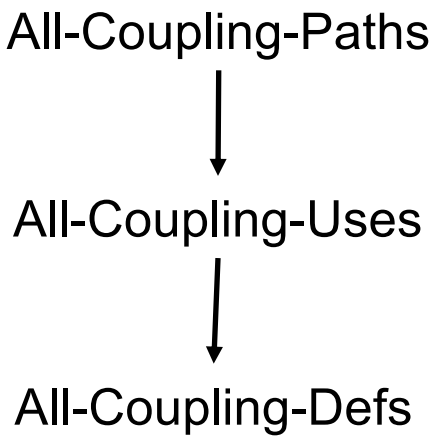
- Use-clear path
  - A path is use-clear with respect to a variable if it does not contain a use of that variable (except the last node/edge of the path)
- Why do we require use-clear paths?
  - Because we assume integrated units have passed their unit tests
  - i.e., further uses (past the first use) have been exercised

Parameter Coupling

- Parameter coupling path:
  - last-def-before-call $\rightarrow$ first-use-in-callee
  - last-def-before-return $\rightarrow$ first-use-after-call
- List of criteria (apply to 3 types of coupling paths):
  - all-coupling-defs (adaptation of all-defs)
    - requires that for each coupling definition at least one coupling path to **at least one** reachable coupling use is executed.
  - all-coupling-uses (adaptation of all-uses)
    - requires that for each coupling definition at least one coupling path to **each** reachable coupling use is executed.
  - all-coupling-paths (adaptation of all du-paths)
    - requires that **all** loop-free coupling paths be executed.

Parameter Coupling

- Subsumption hierarchy:



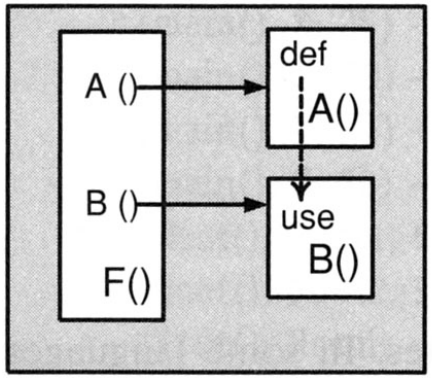
Example

```
procedure QUADRATIC is

…
begin
  GET (Control_Flag);
  if (Control_Flag = 1) then
    GET(X); --- last-def-before-call (X)
    GET(Y); --- last-def-before-call (Y)
    GET(Z); --- last-def-before-call (Z)
  else
    X := 0; --- last-def-before-call (X)
    Y := 0; --- last-def-before-call (Y)
    Z := 0; --- last-def-before-call (Z)
  end if;
  OK := TRUE; --- last-def-before-call (OK)
  ROOT(X,Y,Z,R1,R2,OK); --- call-site
  if OK then --- first-use-after-call (OK)
    PUT(R1); --- first-use-after-call (R1)
    PUT(R2); --- first-use-after-call (R2)
  else
    PUT("No solution");
  end if;
end QUADRATIC;
```

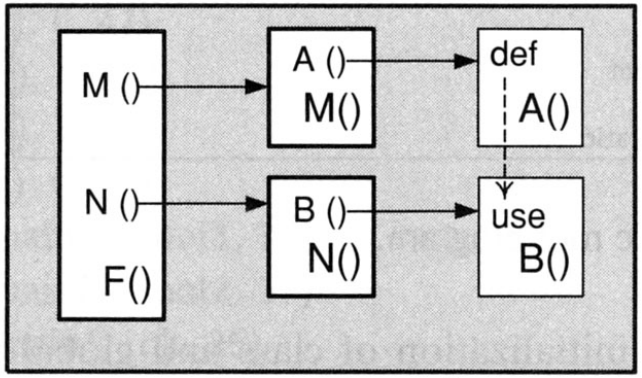
```
procedure ROOT(A,B,C: in FLOAT;
               ROOT1,ROOT2: out FLOAT;
               Result: in out BOOLEAN) is

…
D: FLOAT
…
begin
  D := B**2-4.0*A*C;
       --- first-use-in-callee (A,B,C)
  if (Result and D < 0.0) then
       --- first-use-in-callee (Result)
    Result := FALSE
       --- last-def-before-return (Result)
    return;
  end if;
  ROOT1 := (-B+sqrt(D))/(2.0*A);
       --- last-def-before-return (ROOT1)
  ROOT2 := (-B-sqrt(D))/(2.0*A);
       --- last-def-before-return (ROOT2)
  Result := TRUE;
       --- last-def-before-return (Result)
end ROOT;
```

Discussion

- Does not account for transitive du-pairs
  - If A calls B, and B calls C, last-defs in A do not reach first-uses in C.
- Same comment as in previous section for uses/defs of arrays and objects
- Coupling in OO software does not necessarily involve direct calls.



object-oriented
direct coupling
data flow



object-oriented
*in*direct coupling
data flow

Ammann & Offutt