Data-flow Based Testing

(A&O Ch. 2.2.2, 2.4.2, 7.1)

(2nd Ed. Ch 7.2.3, 7.4.2. Unfortunately, there is no equivalent of 7.1 in the 2nd Ed.)

Course Software Testing & Verification

2024/25

Wishnu Prasetya & Gabriele Keller

Plan

- Data flow driven testing
- Integration testing
- Testing inter-class interactions in an OO setup

Note: an important criticism to control-flow based testing (covered in the lecture on graph-based testing) is that it ignores how data actually flows through the control flows, and we cannot distinguish between which data flows might be more critical to test. Data-flow based testing tries to address this shortcoming. Later, it also gives us a useful instrument to handle integration level testing.

OO poses another kind of challenges for testing, in particular with respect to error prone patterns due to inheritance and dynamic binding. We will also discuss how to address such situations; we will build our approach based on the previously discussed integration testing approach.

Basic idea

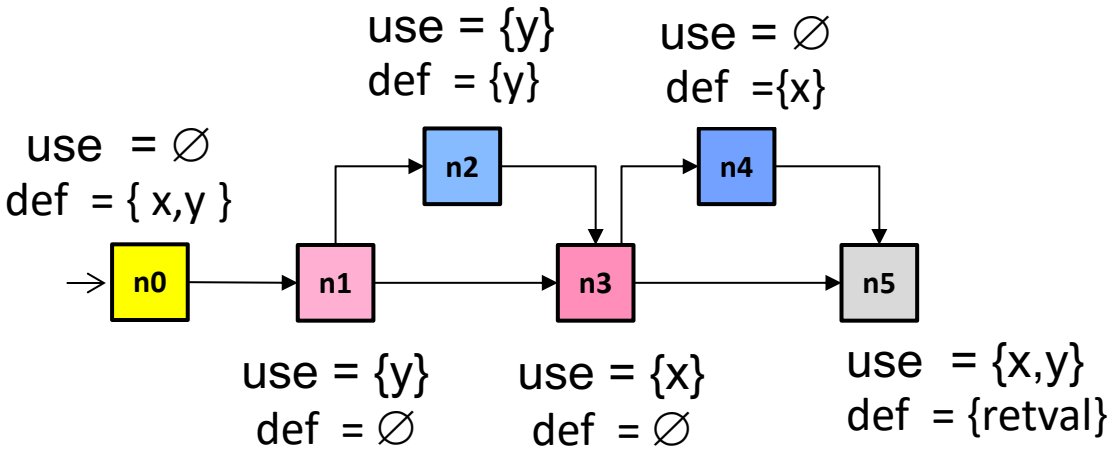
```
P(x,y)  {  if (y<0)  y = -y      else y = 1 ;
           if (x<0)  x = -x + y else x = 0  ;
           return  x }
```

- Imagine the test set (leaving out the oracles): { P(-1,-1) , P1(0,0) } . This two tests would give us full edge coverage.
- However, this ignores data flow between each branch of the first "if" to each branch of the second "if". Each data flow triggers different behavior, and arguably should have been tested.
- Arguably, dataflow is what actually determines the behavior of a program, rather than control flow. From this perspective, dataflow based testing is more complete than control flow based.

Data-flow based approach

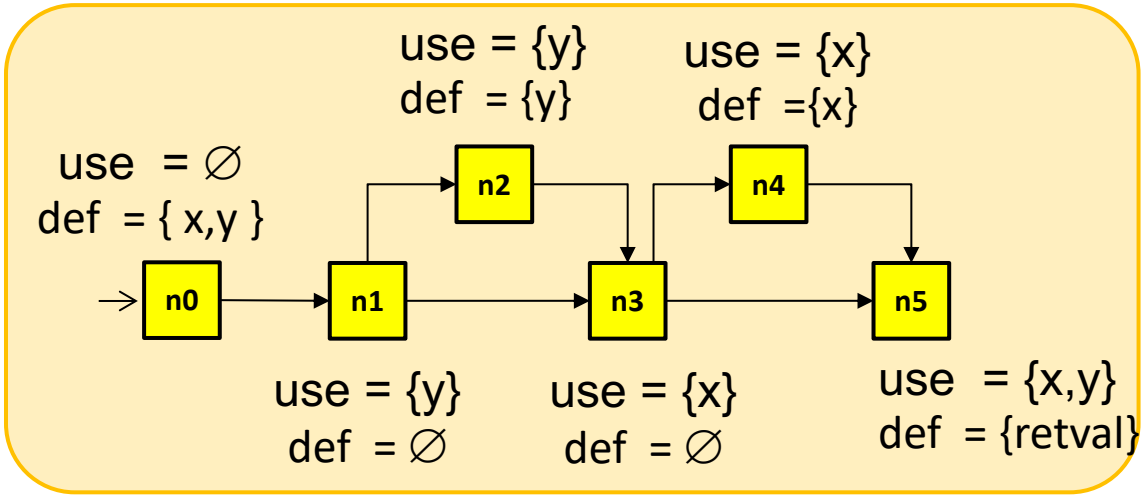
(sec. 2.2.2/ 2nd Ed: 7.2.3)

```
P(x,y) { if (y<0) y = -y
         if (x<0) x = 0
         return x+y     }
```



- Data flow graph : CFG, where the nodes and edges are decorated with information about the variables used and defined by each node/edge.
- In my examples I avoid decorating on the edges.
- Some subtlety on the granularity of the nodes/edges → later.

Terminologies

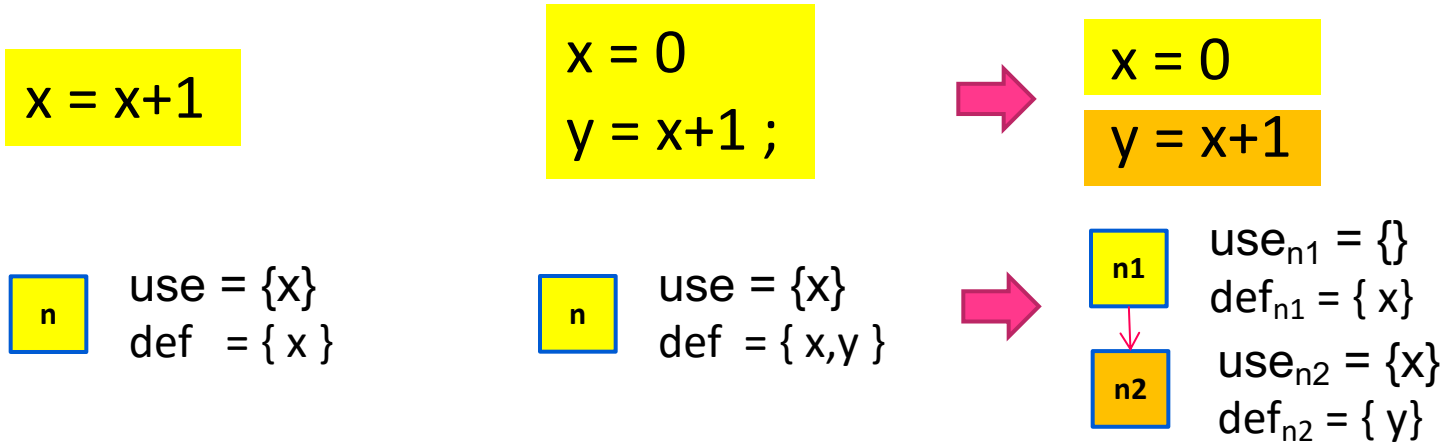


du-paths wrt x:
013, 0123
0134, 01234,
0135, 01235
45

Not du-path wrt x:
01345 (not def-clear)

- Def/use graph: see above. In general, def/use labels are allowed on the edges as well.
- A path p = [i,...,k] is *def-clear* wrt v iff $v \notin def(j)$ and $v \notin def(e)$ for all nodes j and edges e in p ***between*** i and k.
- A *du-path* wrt v = a **simple** and **def-clear** path from i to k with $v \in def(i)$ and $v \in use(k)$.

Block's granularity



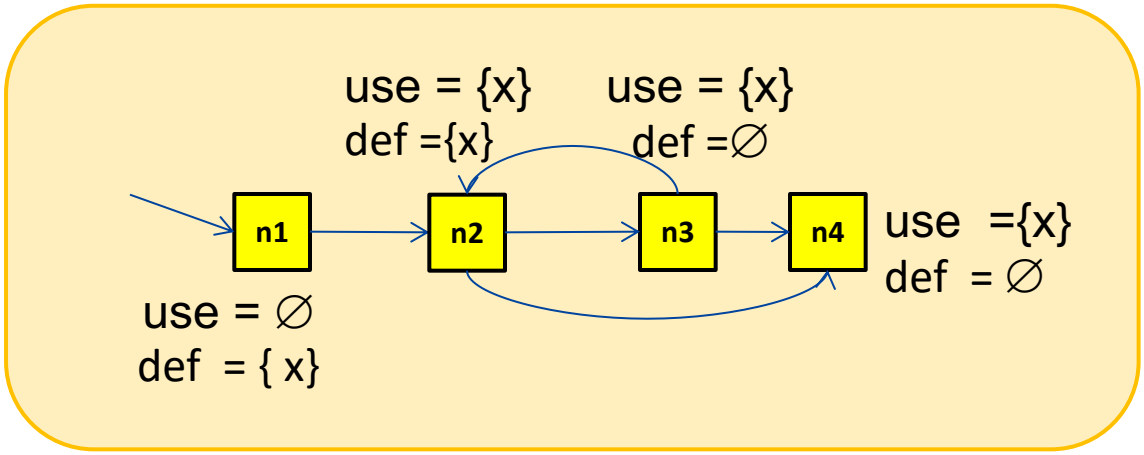
- When a node both *defines* and *uses* the same variable *x*, we will assume that it uses *x* first, *then* assigns to it, as in

  x = ... x ...

- If this assumption is broken (middle example) → **split** the node. Else this will conflict the intention of your definition of du-path.
- Check 2.3.2 (2nd Ed. 7.3.2) on how to map a source code to a def/use graph.

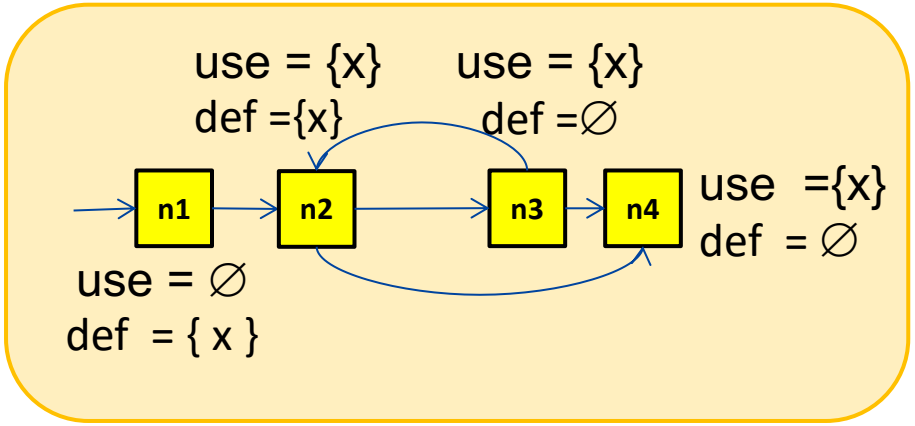
Data-flow based coverage

(Rapps-Weyuker 1985, Frankl-Weyuker 1988)



- *Def-path set* *du*(*i*,*v*): the set of all du-paths wrt *v*, that starts at *i*.
- *Def-pair set* *du*(*i*,*j*,*v*): the set of all du-paths wrt v, that starts at *i*, and ends at *j*.
- Same concept of touring.
- (C 2.9/2nd Ed. C7.15, All Defs Covrg, **ADC**) For each def-path set S = du(i,v), TR includes at least one member of S.

Data-flow based coverage



ADC TR:
12,
234

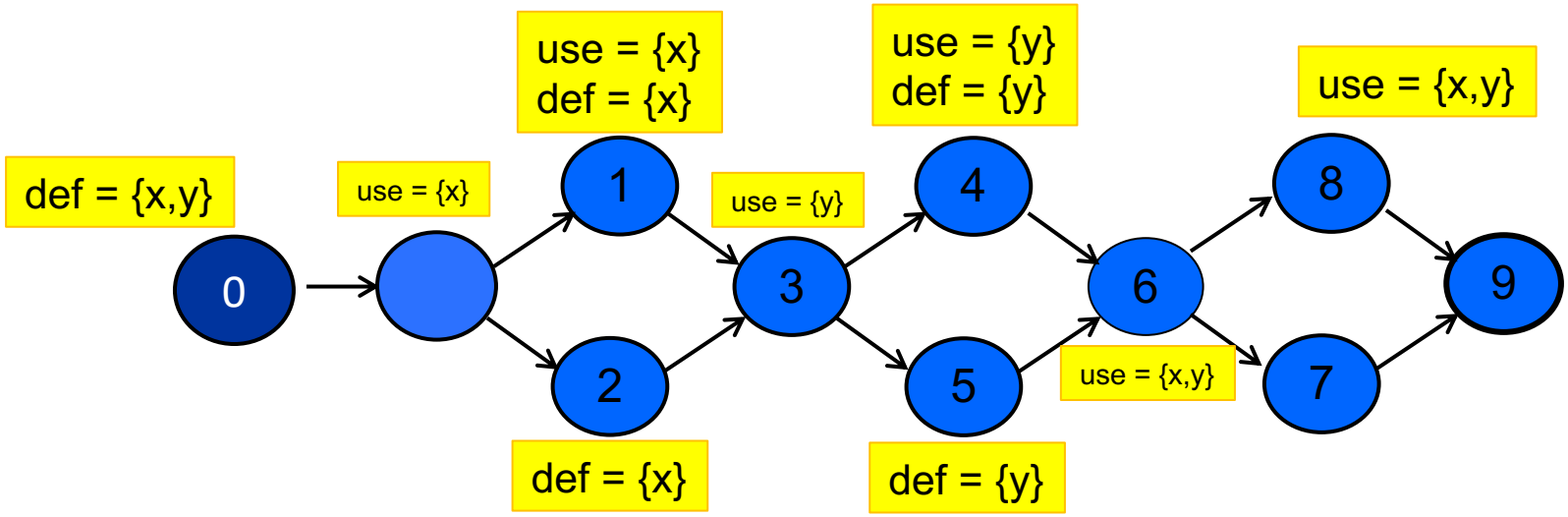
AUC TR:
12
23 , 232,
234

ADUC TR:
12
23 , 232,
234, 24

- (C 2.10/2nd Ed. C7.16, All Uses Covrg, **AUC**) For each def-pair set S = du(i,j,v), TR includes at least one member of S.
- (C2.11/2nd Ed. C7.17, All du-paths Covrg, **ADUPC**) For each def-pair set S = du(i,j,v), TR includes *all* members of S.
- Note: the above example only has one variable, namely x; consequently all TRs above only concern x.

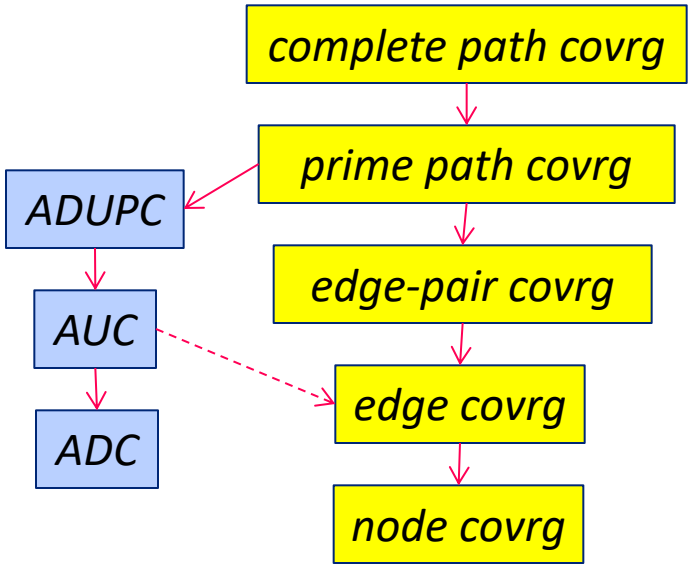
Example AUC vs PPC

```
if x>0 then x = 2x else x=0
if y>0 then y = 2y else y=0
if x*y> 0 then return x+y else return 0
```



- 8 prime paths; we need 8 tests to cover them all
- There are at least 6 def-use-pair sets: du(0,1,x), du(1,8,x), du(2,8,x), du(0,4,y), du(4,8,y), du(5,8,y). You can cover them all (AUC) with 2 tests.
  Note-1: covering the above def-use pairs will also cover the remaining du-pairs.
  Note-2: on the down side, you may miss covering node-7.

Overview of subsumption relations



- PPC $\supseteq$ ADUPC, because every simple path is a subpath of some prime path.
- AUC $\supseteq$ EC .... under the following assumptions:
  - there is at least one def
  - every def reaches at least one use.
  - every use (of *v*) is preceded by a def.
  - every edge uses a variable (note: not the case in "if e then S" without else)

Summary

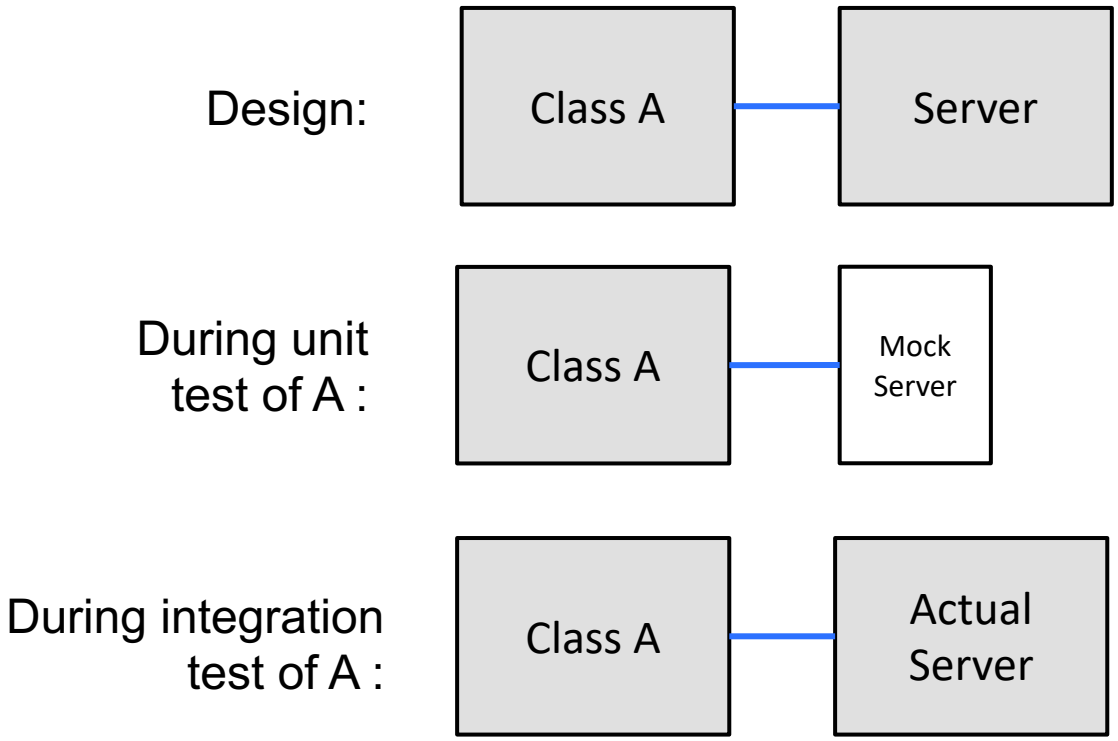
- We learned another set of white-box test-coverage criteria, namely data-flow based coverage.

- An alternative to prime-path-based testing, if the latter becomes too expensive, while being more aware of the semantic of the program under test.

Integration Test

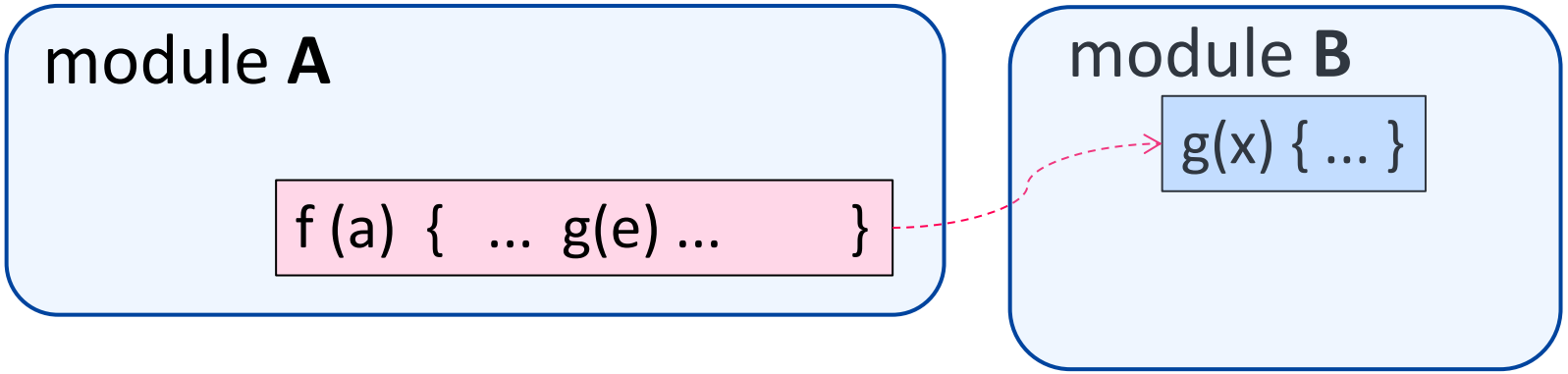
- Integration test: to test if a composition of program units works correctly.
- “Hidden” in Section 2.4 (2nd Ed. 7.4) about using design elements (e.g. your UML models) as your coverage criterion. 2.4.1 (2nd Ed. 7.4 p146) says: *“testing software based on design elements is usually associated with integration testing”.*
- 2.4.2 (2nd Ed. 7.4.2) is actually more about integration testing rather than design-element-based testing.

Integration Test, example



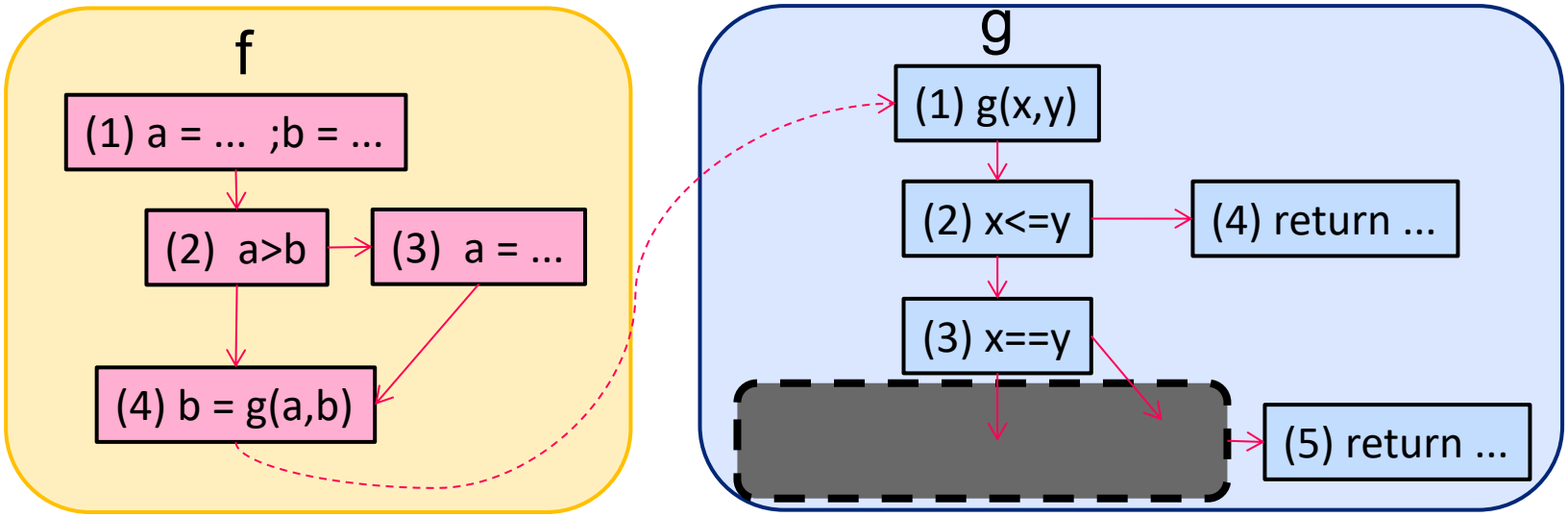
We can test the integration between A and the Server by re-running the unit test of A, but replacing the mock server with a real server. Question: what should we use to determine the adequacy of this test?

Data flow approach to Integration Test (2.4.2/2nd Ed. 7.4.2)



- Imagine a method *f* from module *A* uses a method/API *g* from module B. How to test such a composition?
- Idea 1: cover all edges → does not capture "integration".
- Idea 2: cover all prime paths in the combined CFG of both → blows up.
- Idea 3: focus on the flow of the "coupling data" around the call point.

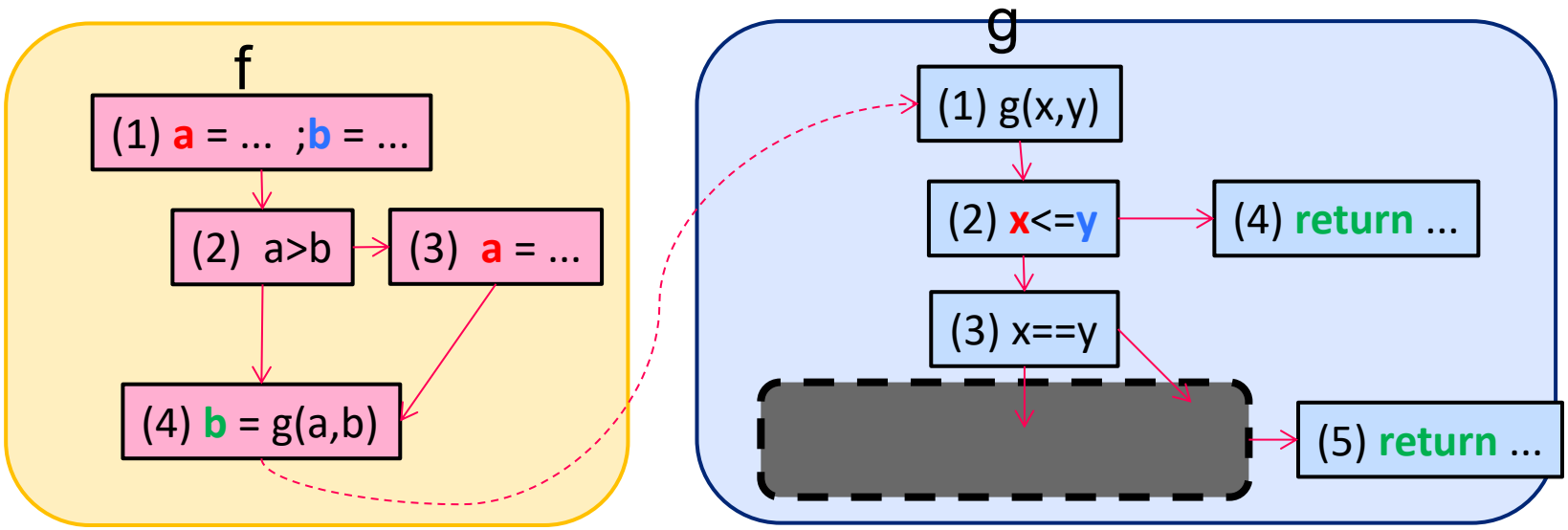
Terminology



- *caller, callee, actual parameters, formal parameters*
- *parameter coupling, shared variable coupling, external device coupling*
- More general concept: *coupling variable* → a 'variable' defined in one 'unit', and used in the other.

Coupling du-path

(Jin-Offutt, 1998)



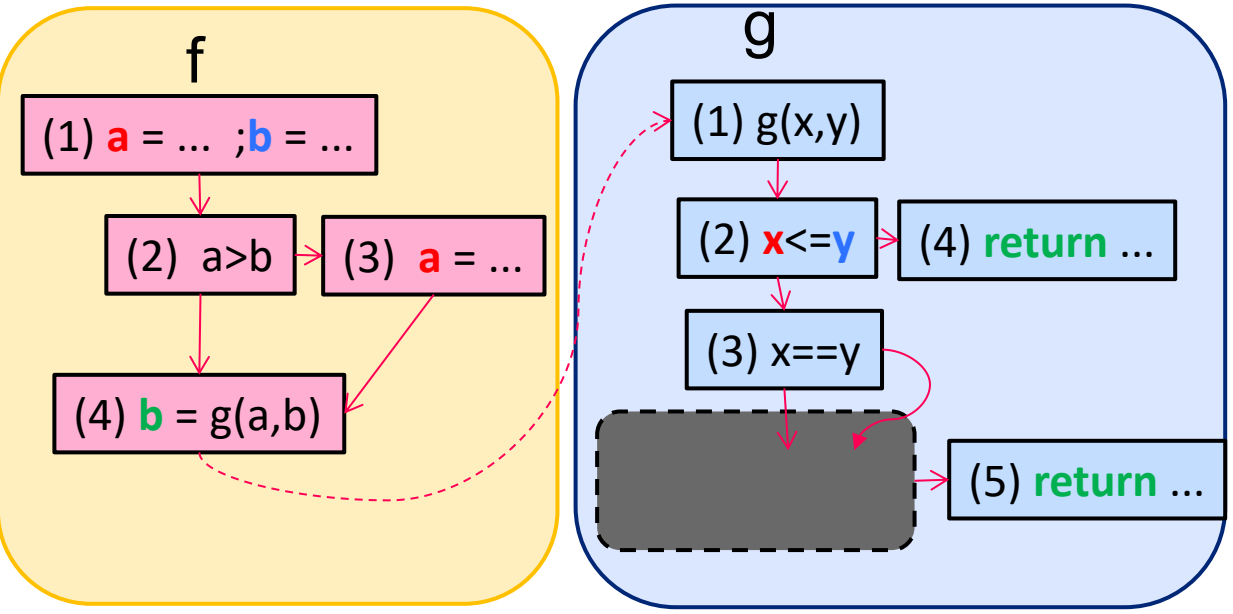
- (Def 2.40/2nd Ed. 7.39) *Last-def of v* : set of nodes defining *v*, from which there is a def-clear path (wrt *v*) through the call-site, to a use in the other unit.
- Can be at the caller or callee! Example: b@f1, a@f3, g.return@g4, g.return@g5.
- (Def 2.41/2nd Ed. 7.40) *First-use (of v)* E.g.: b@g2, g.return@f4
- *Coupling du-path of v* is a path from *v*'s last-def to its first-use.

Note on the first use at the callee

- When you just pass a variable as a in g(a,b), it makes sense to define g's first use to be the first time g uses the parameter (location g.2 in the previous example).

- But when you pass a complex expression like g(a+b,0), things become more obscure. We will define g's first uses of a and b to be the first node of g that uses, in this example, its first formal parameter. So, in this example it is g.2.

Integration Test-Requirement



Coupling vars:

a, b, g.return

Coupling du-paths :

a: [f1,f2,f4,g1,g2]
[f3,f4,g1,g2]

b: [f1,f2,f4,g1,g2]
[f1,f2,f3,f4,g1,g2]

g.return: [g4,f4]
[g5,f4]

- (ACDC) *All Coupling Defs Cov*. : for every last-def *d* of each coupling var *v*, TR includes one coupling du-path starting from *d*.
- (ACUC) *All Coupling Uses Cov*. : for every last-def *d* and first-use *u* of *v, TR* includes one couple du-path from *d* to *u*.

The next slides are based on A&O Ch 7.1. Unfortunately, this chapter does not appear in the 2nd Ed.

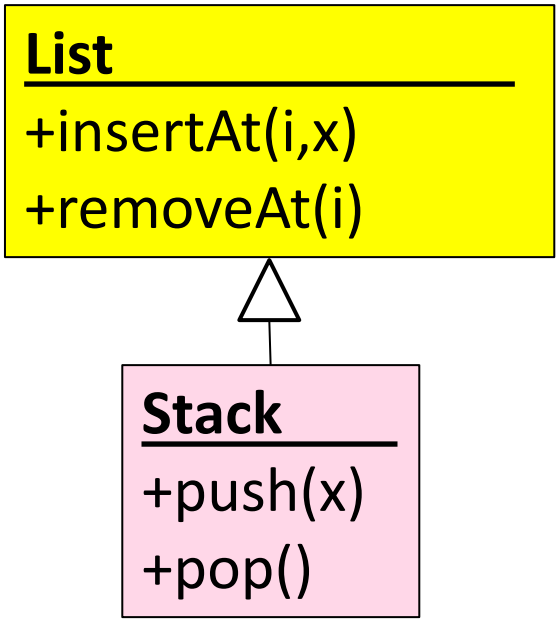
OO is powerful, but also comes new sources of concerns

- A function behaves “cleanly”. In contrast, a procedure may have side effects.
- An OO program is more complicated:
  - access control (priv, pub, default, protected, ...)
  - side effects on instance variables and static variables
  - inheritance:
    - dynamic binding
    - interaction through inheritance

Inheritance-related error prone programming patterns

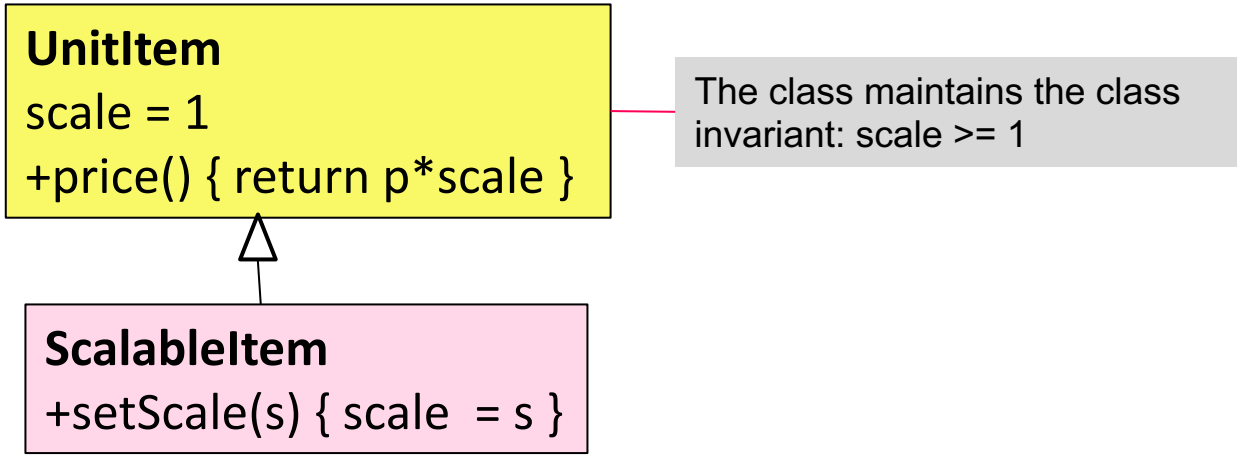
- A&O call them inheritance-related “anomalies”: use of error prone programming patterns. AO list 9 anomalies related to inheritance and dynamic binding:
  - Inconsistent Type Use (ITU)
  - State definition anomaly (SDA) , ... 7 more
- A&O also implicitly discuss two more: deep yoyo and data-flow anomaly.

Inconsistent Type Use (ITU) anomaly



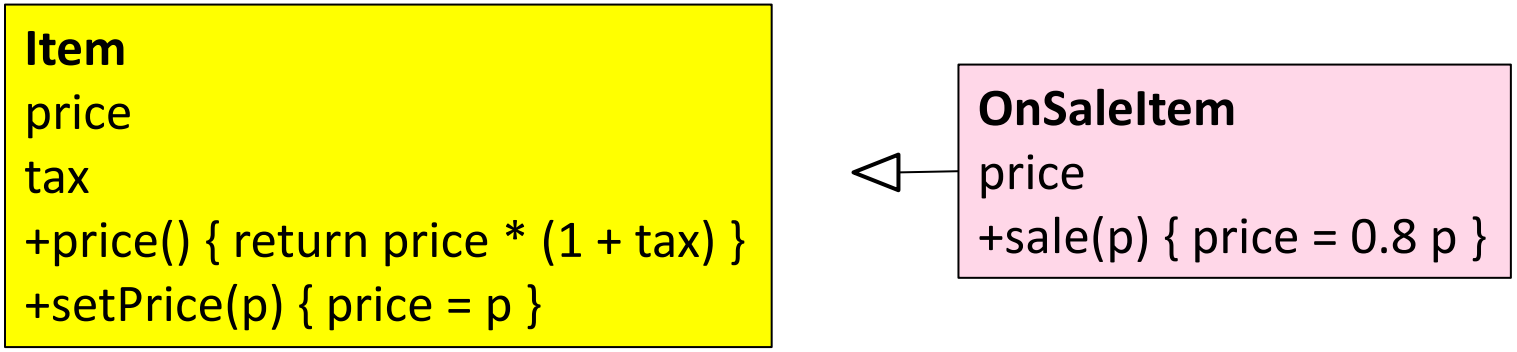
This is a quite common subclassing pattern. However note that in OO a Stack s can also be used as a List, which leads to an error prone situation, since a stack should not allow elements to be inserted at or removed from an arbitrary position. But a user method may not be aware that s is actually a stack, and call insertAt and removeAt! The class Stack should have overridden the methods, and e.g. make them throw an illegal operation exception.

State Definition Anomaly (SDA)



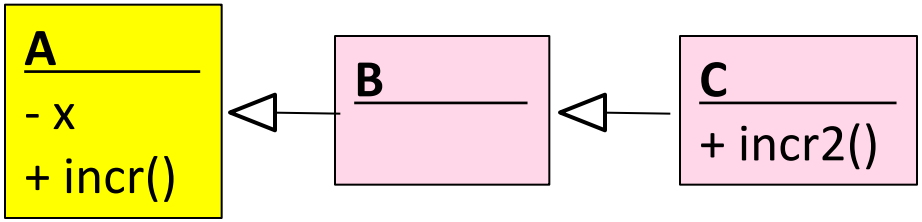
ScalableItem **directly** manipulates the part of its state which it inherits from UnitItem (the assignment "scale=s"), rather than doing so through UnitItem's method. This error prone as it may unwittingly break UnitItem's class invariant.

State definition inconsistency due to state variable hiding (SDIH)



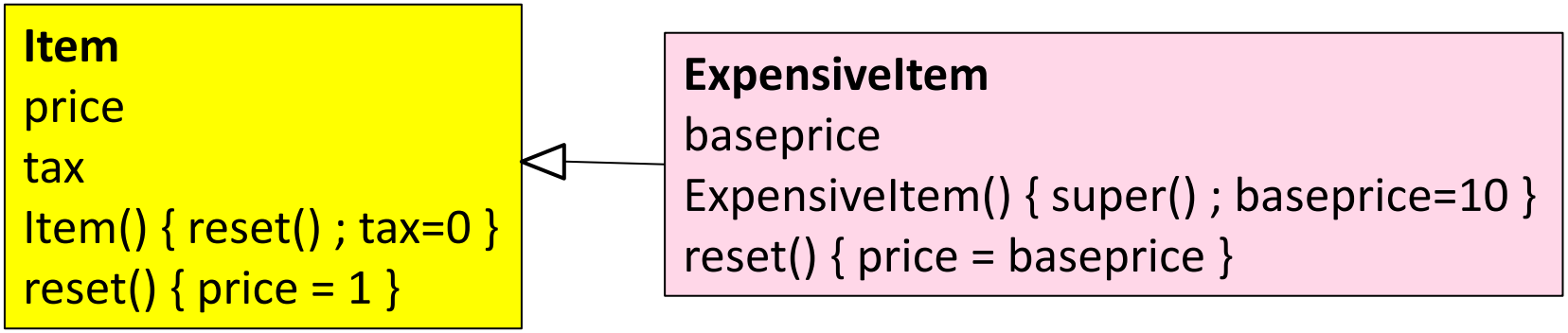
The class OnSaleItem declares it own “price” field, which then shadows the original “price” field inherited from the class Item. Notice that OnSaleItem does not override setPrice, which uses “price” in its calculation. This leads to an error prone situation. When we have an instance o of OnSaleItem, and we call o.price(), in Java the method price() would use the old price rather than the newly declared price.

State Visibility Anomaly (SVA)



C's needs a method *incr2* that would increase *x* by 2. However it can't get to *x*, because it is private to *A*. It can however calls incr() twice. So far it is ok. Now imagine someone else changes *B* by overriding *incr*. This suddenly changes the behavior of *C*; as it now calls *B*'s *incr* instead.

Anomalous Construction Behavior 1 (ACB1)



In the above design, Item's constructor calls reset(). ExpensiveItem overrides reset(). So far so good. But then, ExpensiveItem's constructor calls Item's constructor. This leads to an error prone situation. The latter would then call reset(). In Java, it will call the new reset() rather than the old one (which may or may not be the intended behavior).

Check the remaining anomalies yourself

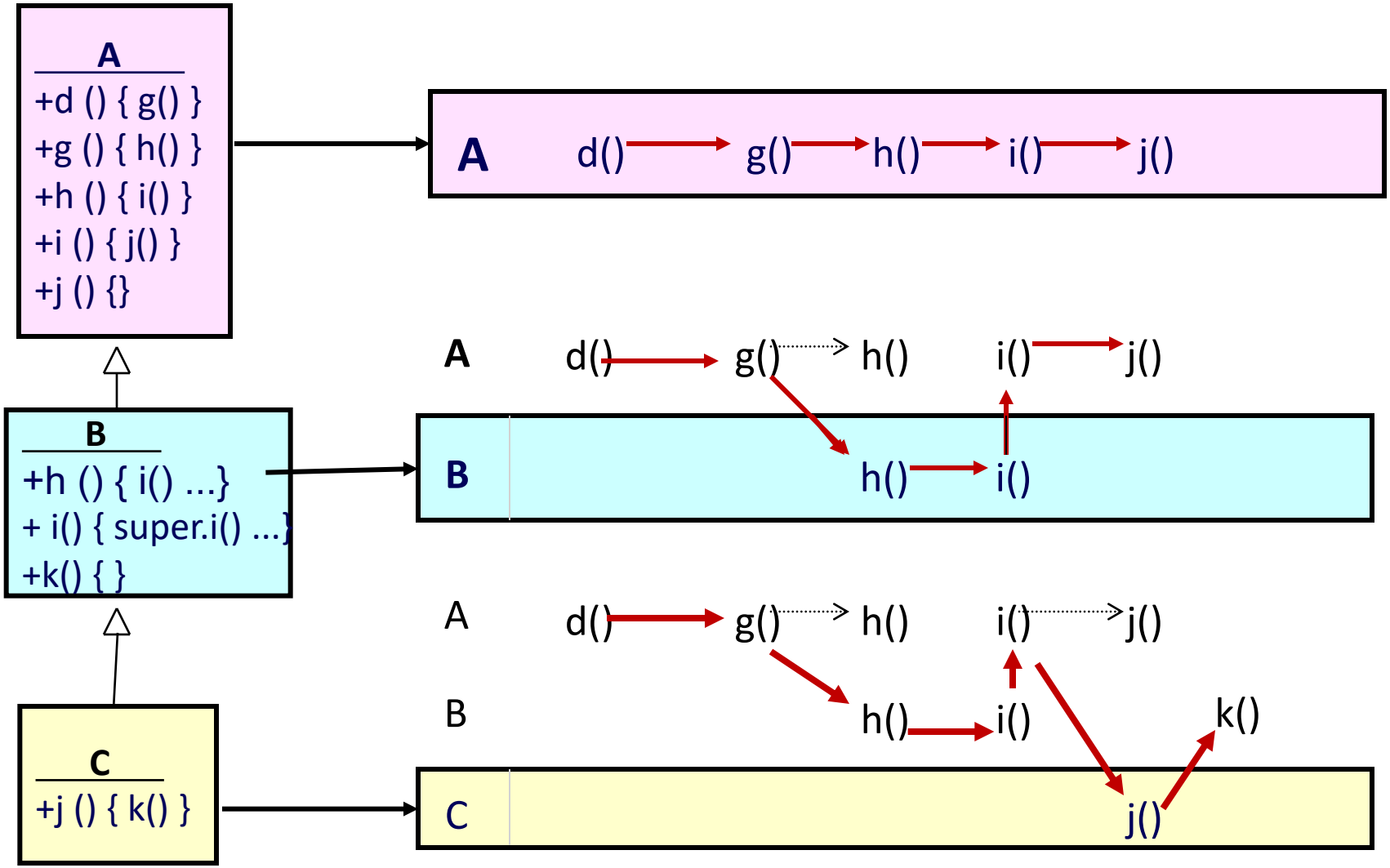
- State Defined Incorrectly (SDI)
- Indirect Inconsistent State Definition (IISD)
- Anomalous Construction Behavior 2 (ACB2)
- Incomplete Construction (IC)

Error prone situations due to inheritance

- In previous examples we have seen how inheritance can change the behavior of a class, and can create error prone situations.
- These are not necessarily errors (could be the intended behavior), but there are definitely needs to verify them.
- Also notice that methods from subclasses and superclasses can implicitly call each other, called “yoyo” effect.

The call graph of d() in A,B,C

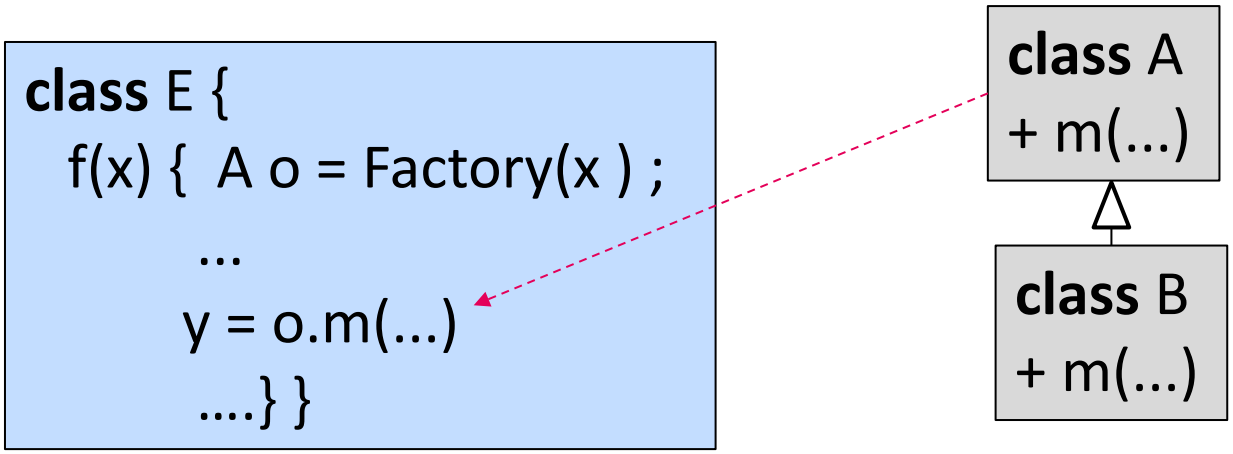
yoyo graph, Alexander-Offut 2000



Testing OO programs

- Of course: every method/class should be unit-tested.
- However, previously mentioned inheritance-related error prone situations happen at the integration level when a module uses a class C, but it might get a subclass of C at the runtime.
- We can test this through the previously discussed integration test approach, and enhance it to cover for inheritance as well.

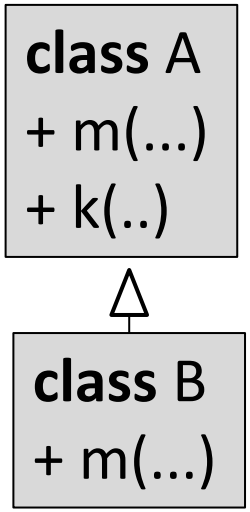
Inter-class testing 1



Testing the usage of class A by class E : treat this as an integration testing problem → we already have a solution (data-flow based integration test), but **additionally** now we need to quantify over the subclasses of A, due to dynamic binding (the exact *m* called in *f* depends on the type of actual type of *o*).

Inter-class testing 2

```
class E2 {
  f(x) {   A o = Factory(x ) ;
           …
           y = o.m(...)
           ...
           z = o.k(...)   …}
```



A class E2 that call methods of A **multiple times** within the same method (f). This may create additional dynamic: the behavior of o.m() may affect o.k() through side effect on o's fields. Some of this behavior could be critical. We will extend the previous approach of integration testing to also cover this kind of cross-method interactions through objects' state.

Terminology

```
f(x) {  (1)  A o = Factory(x ) ;
        (2)  y = o.m(..) ;
        ...
        (3)  z = o.k(..)  }
```

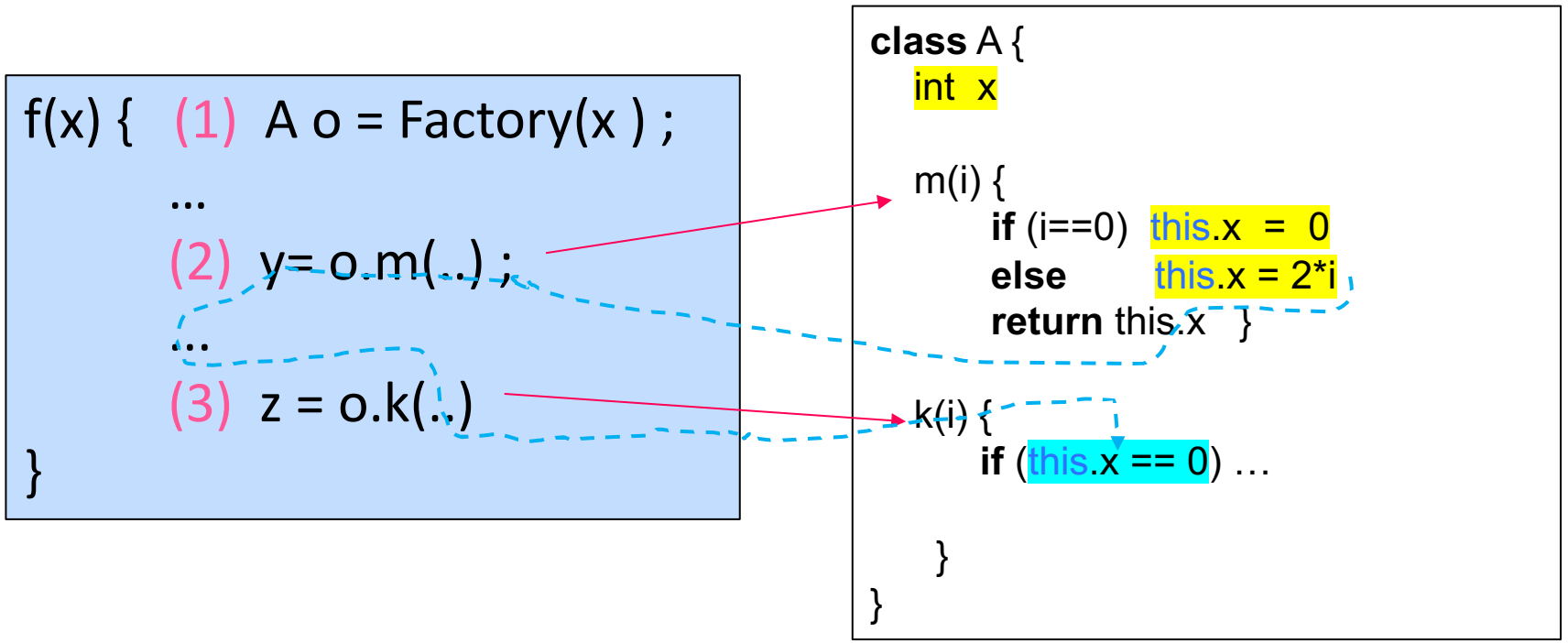
- *f* itself is here called the *coupling method*.
- The field o.x is "indirectly" defined (i-def) if the field is updated in some o.method() called by f.
  Analogously, we define i-use.
- Focus on coupling **within** *f* due to fields o.*x* indirectly defined by o.*m*(), and used by o.*k*(); o.m() is then called *antecedent* of this coupling and o.k() the *consequent*.

Terminology

```
f(x) {  (1)  A o = Factory(x ) ;
        (2)  y= o.m(..) ;
        ...
        (3)  z = o.k(..)                }
```

- *Coupling sequence c*: a pair of antecedent m, and consequence *n*, such that there is at least 1x *coupling path*: a **def-clear** path from *m* to *k* with respect to some field *o.x* defined in *m* and used in *k.*
- *o* is called the *context var* of the coupling sequence *c*
- *o.x* is called a *coupling var* (of *c*)
- The set of all coupling vars of *c* → coupling *set.*

Coupling Sequence, example



- Dashed-blue line: a coupling path σ over o.x from a yellow in m() to the blue in n(), assuming the path is def-clear on o.x in between.
- 2 → 3 in f(x) forms a coupling sequence if there is such a coupling path.

Example with multiple coupling sequences

```
f2(x) {  (1)  A o = Factory(x ) ;
         (2)  o.m(..)
         ....
         (3)  o.k(..)
         ...
         (4)  o.n(..)  ...                }
```

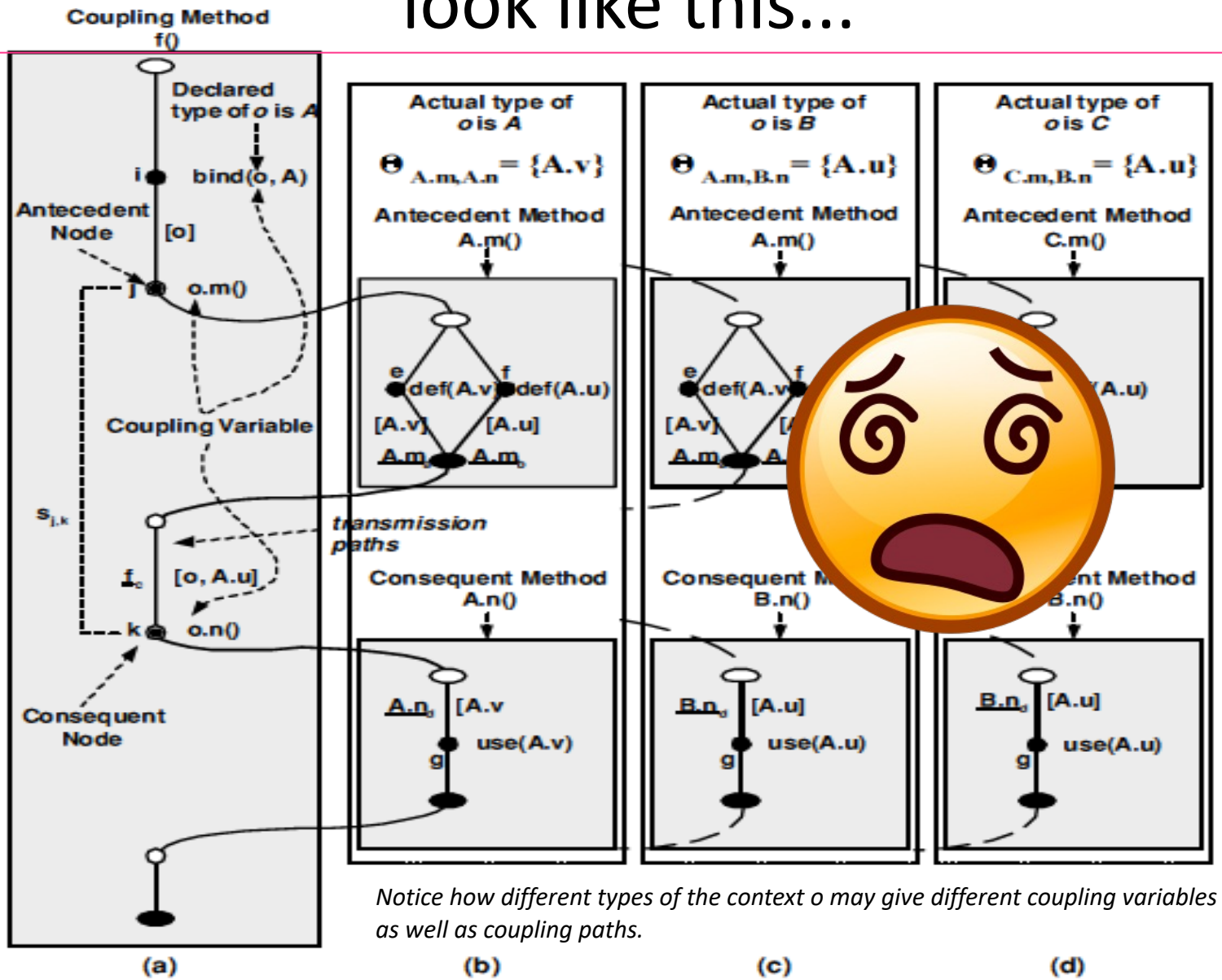
idef = { o.x , o.y }

iuse = { o.x, o.y } idef = { o.x }

iuse = { o.x , o.y } idef = { o.x }

Coupling sequence	Coupling vars	note
2 → 3	o.x , o.y	Assuming there are def. clear paths between the idef of o.x , o.y in 2 and their iuse in 3
3 → 4	o.x	Assuming there is a def. clear path between the idef of o.x in 3 and its iuse in 4
2 → 4	o.y	Assuming there is no def. clear path between the idef of o.x in 2 and its iuse in 4

With polymorphism the situation will now look like this...

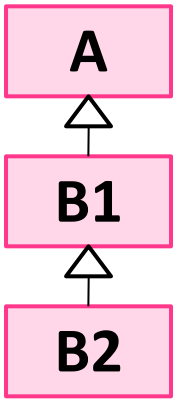


Notice how different types of the context o may give different coupling variables as well as coupling paths.

Binding triple

Each coupling sequence *c* induces one or more binding "triples"; each specifying the coupling variables for **various types** of the context-var *o*. For example, these could be the binding triples of the coupling sequence *cs1* = o.m()@2→ o.k()@3 ;

```
f2(x) {  (1)  A o = Factory(x ) ;
         (2)  o.m(..)
         ....
         (3)  o.k(..)
         ...
         (4)  o.n(..)  ...          }
```

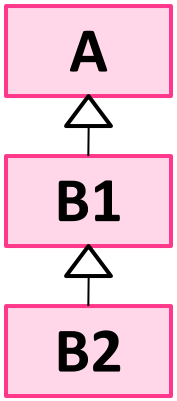


coupling seq	type of o	coupling vars
cs1	A	{ o.v }
	B1	{ o.u }
	B2	{ o.u , o.v }

binding triples of cs1

Inter-class coverage test criteria

Imagine the following coupling sequences within some method f :



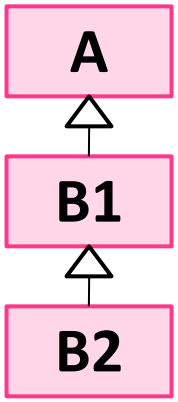
coupling seq. in f	type(o)	coupling vars	coupling paths
cs1	A	{ o.v }	o.v : { σ_1 }
	B1	{ o.u }	o.u : {τ_1, τ_2 }
	B2	{ o.u , o.v }	o.u : { π_1 } , o.v :{π_2 , π_3}
cs2	B2	{ o.u , o.v }	o.u : { ρ_1 } , o.v :{ρ_2}

binding triples of c = o.m()→ o.n() , extended with infos over coupling paths.

(Def 7.53) **All-Coupling-Sequences** (ACS): **for every coupling sequence** *cs* in the coupling method f , TR includes at least one coupling path of *cs.* → ignore polymorphism.

In the above example, the TR would consist of two paths, one for *cs1* and one for *cs2*.

Inter-class coverage test criteria

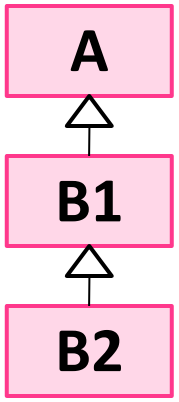


coupling seq. in f	type(o)	coupling vars	coupling paths
cs1	A	{ o.v }	o.v : { σ_1 }
	B1	{ o.u }	o.u : {τ_1, τ_2 }
	B2	{ o.u , o.v }	o.u : { π_1 } , o.v :{π_2, π_3 }
cs2	B2	{ o.u , o.v }	o.u : { ρ_1 } , o.v :{ρ_2}

(Def 7.54) **All-Poly-Classes** (APC): for **every coupling sequence** cs, and **every binding triple** *t* of *cs*, TR includes at least one coupling path of *t.* → ignore that *cs* may have multiple coupling vars.

In the above example, the TR would consist of 3 paths for cs1, and 1 for cs2.

Inter-class coverage test criteria



coupling seq. in f	type(o)	coupling vars	coupling paths
cs1	A	{ o.v }	o.v : { σ_1 }
	B1	{ o.u }	o.u : {τ_1, τ_2 }
	B2	{ o.u , o.v }	o.u : { π_1 } , o.v :{π_2 , π_3}
cs2	B2	{ o.u , o.v }	o.u : { ρ_1 } , o.v :{ρ_2}

- (Def 7.55) **All-Coupling-Defs-Uses** (ACDU): for **every coupling sequence** cs and **every coupling var *v*** of *cs*, TR includes at least one of *v's* coupling path.
- (Def 7.56) **All-Poly-Coupling-Defs-Uses** (APCDU): for **every coupling sequence** cs and **every coupling var** *v* of **every binding triple** of *cs,* TR includes at least one of *v's* coupling path.