Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 11 — January 27, 2010

Patrick Lam *version 1*

Data Flow Graph Coverage for Design Elements

The structural coverage criteria for design elements were not very satisfying: basically we only had call graphs. Let's instead talk about data-bound relationships between design elements.

First, some terms.

- *caller:* unit that invokes the callee;
- *actual parameter:* value passed to the callee by the caller; and
- *formal parameter:* placeholder for the incoming variable.

Illustration.

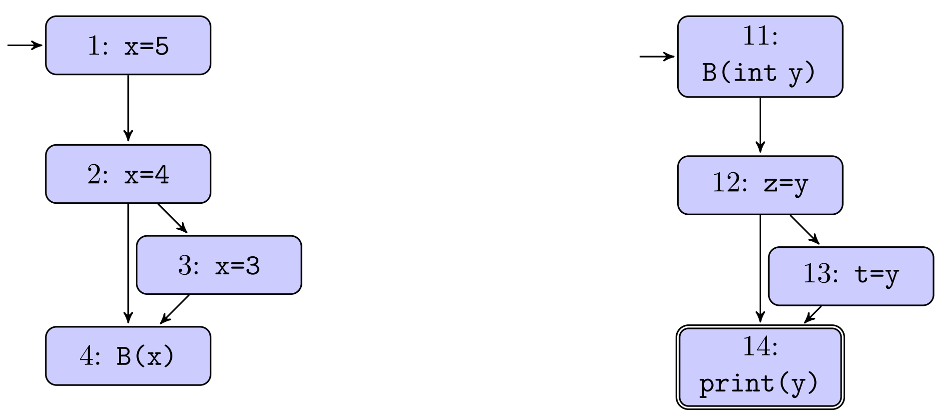
caller:

```
...
foo(actual1, actual2);
...
```

callee:

```
void foo(int formal1, int formal2) {
  ...
}
```

We want to define du-pairs between callers and callees. Here's an example.



We'll say that the *last-defs* are 2, 3; the *first-uses* are 12, 13. We formally define these notions:

Definition 1 (Last-def). *The set of nodes N that define a variable* x *for which there is a def-clear path from a node $n \in N$ through a call to a use in the other unit.*

Definition 2 (First-use). *The set of nodes N that have uses of* `y` *and for which there exists a path that is def-clear and use-clear from the entry point (if the use is in the callee) or the callsite (if the use is in the caller) to the nodes N.*

We need the following side definition, analogous to that of def-clear:

Definition 3 *A path $p = [n_1, \ldots, n_j]$ is* use-clear *with respect to v if for every $n_k \in p$, where $k \neq 1$ and $k \neq j$, then v is not in use(n_k).*

In other words, the last-def is the definition that goes through the call or return, and the first-use picks up that definition.

Here is another example:

```
x = 14;   // last-def
y = g(x);
print(y); // first-use
```

```
int g(a) {
  print(a); // first-use
  b = 24;   // last-def
  return b;
}
```

Tests can, of course, go beyond just testing the first-use. Our first-use and last-def definitions, however, make testing slightly more tractable. We could, of course, carry out full inter-procedural data-flow, i.e. covering all du-pairs, but this would be more expensive.

Coupling

We've just seen that we might want to test the interaction between two different design elements. Here are three different ways that different program parts might interact:

- *parameter coupling*: relationships between caller and callees; passing data as parameters;
- *shared data coupling*: one unit writes to some in-memory data, another unit reads this data; and
- *external data coupling*: one unit writes data e.g. to disk, another read reads the data.

In all of these cases, *coupling variables* mediate the transfer of data between units. We can generalize the notion of a *du-path* to a *coupling du-path* with respect to a particular coupling variable, giving us the coverage criteria "All-Coupling-Defs", "All-Coupling-Uses", and "All-Coupling-du-Paths", all with last-def/first uses.

Note that parameter coupling could also apply to distributed software; there is just a more complicated implementation of passing parameters.

Notes on Coupling Data-flow:

- We are only testing variables that have a definition on one side and a use on the other wise; we do not impose test requirements in the following case:

  ```
  foo(20, 17);

  foo(int x, int y) {
    return x * 5;  // no use of y, so no test requirement
  }
  ```

- Static and instance fields have implicit definitions:

  ```
  class T { int x; int foo() { return x; } }
  ```

  We consider `x` to be defined in `T`'s constructor, and static fields to be defined in `main()`. Of course, the usual definitions are still definitions.

- We don't consider transitive du-pairs:

  $$\rightarrow \boxed{A} \rightarrow \boxed{B} \rightarrow \boxed{C}$$

  A last-def in A doesn't reach a first-use in C for our purposes.

- We will generally treat an entire array as one variable.

Java Example.

```
class M {
  int x;
  void foo() { x = 5; }
  int bar() { return x; }
}
```

```
void notCouplingMethod() {
  M m = new M(), n = new M();
  m.foo(); print (n.bar());
}
```

Because `m` and `n` are different objects, there is no du-pair here. We have a (last-)def of `m.x` and a (first-)use of `n.x`.