Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 10 — January 25, 2010

Patrick Lam *version 1*

Dataflow Graph Coverage for Source Code

Last time, we saw how to construct graphs which summarized a control-flow graph's structure. Let's enrich our CFGs with definitions and uses to enable the use of our dataflow criteria.

Definitions. Here are some Java statements which correspond to definitions.

- `x = 5`: `x` occurs on the left-hand side of an assignment statement;
- `foo(T x) { ... }` : implicit definition for `x` at the start of a method;
- `bar(x)`: during a call to `bar`, `x` might be defined if `x` is a C++ reference parameter.
- (subsumed by others): `x` is an input to the program.

Examples:

Uses. The book lists a number of cases of uses, but it boils down to "`x` occurs in an expression that the program evaluates." Examples: RHS of an assignment, or as part of a method parameter, or in a conditional.

Complications. As I said before, the situation in real-life is more complicated: we've assumed that `x` is a local variable with scalar type.

- What if `x` is a static field, an array, an object, or an instance field?
- How do we deal with aliasing?

One answer is to be conservative and note that we've said that a definition d reaches a use u if it is possible that the address defined at d refers to the same address used at u. For instance:

```
class C { int f; }
void foo(int q) { use(q.f); }

x = new C(); x.f = 5;
y = new C(); y.f = 2;

foo(x);
foo(y);
```

Our definition says that both definitions reach the use.

Compiler tidbit. In a compiler, we use intermediate representations to simplify expressions, including definitions and uses. For instance, we would simplify:

```
x  = foo(y + 1, z * 2)
```

Basic blocks and defs/uses. Basic blocks can rule out some definitions and uses as irrelevant.

- Defs: consider the last definition of a variable in a basic block. (If we're not sure whether `x` and `y` are aliased, leave both of them.)
- Uses: consider only uses that aren't dominated by a definition of the same variable in the same basic block, e.g. `y = 5; use(x)` is not interesting.

Graph Coverage for Design Elements

We next move beyond single methods to "design elements", which include multiple methods, classes, modules, packages, etc. Usually people refer to such testing as "integration testing".

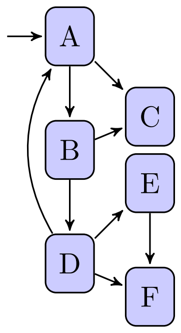
Structural Graph Coverage.

We want to create graphs that represent relationships—*couplings*—between design elements.

Call Graphs. Perhaps the most common interprocedural graph is the *call graph.*

- design elements, or nodes, are methods (or larger program subsystems)
- couplings, or edges, are method calls.

Consider the following example.

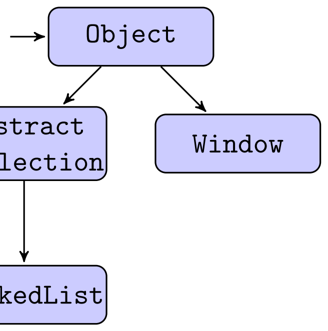


- For method coverage, must call each method.
- For edge coverage, must visit each edge; in this particular case, must get to both A and C from both of their respective callees.

Like any other type of testing, call graph-based testing may require test harnesses. Imagine, for instance, a library that implements a stack; it exposes `push` and `pop` methods, but needs a test driver to exercise these methods.

Object-oriented Structures. Static methods are straightforward to construct callgraphs for and to test, using what we've seen. We can approximate the call graph for instance methods by considering all potential targets of a virtual call.

The other obvious graph in an object-oriented program is the inheritance hierarchy graph.



The inheritance hierarchy is not as useful as we'd hope, since one doesn't really call a class, but instead instantiate an object and call its methods. Here are some possibilities:

- *OO Call Coverage:* instantiate one object per class, get the call graph, then cover that call graph.
- *All-object-call:* for every instantiated object in a program, cover all calls for that object.