Lecture 11: Dynamic Analysis, Part 2

In this lecture, we continue our discussion of dynamic analysis, focusing on testing. We take a brief look at some of the basic notions underlying the theory of testing, and survey the techniques most widely used in practice. At the end, we collect together some practical guidance to help you in your own testing work.

11.1 Testing

Testing is much more effective, and much less painful, if you approach it systematically. Before you start, think about:

- what properties you want to test for;
- what modules you want to test, and what order you'll test them in;
- how you'll generate test cases;
- how you'll check the results;
- when you'll know that you're done.

Deciding what properties to test for will require knowledge of the problem domain, to understand what kinds of failures will be most serious, and knowledge of the program, to understand how hard different kinds of errors will be to catch.

Choosing modules is more straightforward. You should test especially those modules that are critical, complex, or written by your team's sloppiest programmer (or the one who likes clever tricks most). Or perhaps the module that wasn't written latest at night, or just before the release…

The module dependency diagram helps determine the order. If your module depends on a module that isn't yet implemented, you'll need to write a *stub* that stands in for the module during testing. The stub provides enough behaviour for the tests at hand. It might, for example, look up answers in a table rather doing a computation.

Checking results can be hard. Some programs – such as the Foliotracker you'll be building in exercises 5 and 6 – don't even have repeatable behaviour. For others, the results are only the tip of the iceberg, and to check that things are really working, you'll need to check internal structures.

Later on we'll discuss the questions of how to generate test cases and how to know when you're done.

11.2 Regression Tests

It's very important to be able to rerun your tests when you modify your code. For this reason, it's a bad idea to do ad hoc testing that can't be repeated. It may seem like a lot of work, but in the long run, it's much less work to construct a suite of tests that can be reexecuted from a file. Such tests are called *regression tests*.

An approach to testing that goes by the name of *test first programming*, and which is part of the new development doctrine called *extreme programming*, encourages construction of regression tests even before any application code is written. JUnit, the testing framework that you've been using, was designed for this.

Regression testing of a large system is a major enterprise. It can take a week of elapsed time just to run the test scripts. So an area of current research interest is trying to determine which regression test cases can be omitted. If you know which cases test which parts of the code, you may be able to determine that a local change in one part of the code does not require that all the cases be rerun.

11.3 Criteria

To understand how tests are generated and evaluated, it helps to take a step back and think abstractly about the purpose and nature of testing.

Suppose we have a program P that is supposed to meet a specification S. We'll assume for simplicity that P is a function from inputs to outputs, and S is a function that takes an input and an output and returns a boolean. Our aim in testing is to find a test case t such that

S (t, P(t))

is false: that is, P produces a result for the input t that is not permitted by S. We will call t a *failing test case*, although of course it's really a successful one, since our aim is to find errors!

A test suite T is a set of test cases. When is a suite 'good enough'? Rather than attempting to evaluate each suite in a situation-dependent way, we can apply general *criteria*. You can think of a criterion as a function

C: Suite, Program, Spec → Boolean

that takes a test suite, a program, and a specification and returns true or false according to whether, in some systematic sense, the suite is good enough for the given program and specification.

Most criteria don't involve both the program and the specification. A criterion that involves only the program is called a *progam-based* criterion. People also use terms like 'whitebox', 'clearbox', 'glassbox', or 'structural' testing to describe testing that uses program-based criteria.

A criterion that involves only the specification is called a *specification-based* criterion. The terms 'blackbox' testing is used in association with it, to suggest that the tests are judged without being able to see inside the program. You might also hear the term 'functional' testing.

11.4 Subdomains

Practical criteria tend to have a particular structure and properties. They don't, for example, accept a test suite *T* but reject a test suite *T'* that is just like *T* but has some extra test cases in it. They also tend to be insensitive to what combinations of test cases are chosen. These aren't necessarily good properties; they just arise from the simple way in which most criteria are defined.

The input space is divided into regions usually called *subdomains*, each containing a set of inputs. The subdomains together exhaust the input space – that is, every input is in at least one subdomain. A division of the input space into subdomains defines implicitly a criterion. The criterion is that there be at least one test case from each subdomain. Subdomains are not usually disjoint, so a single test case may be in several subdomains.

The intuition behind subdomains is two-fold. First, it's an easy way (at least conceptually) to determine if a test suite is good enough. Second, we hope that by requiring a case from each subdomain, we will drive testing into regions of the input space most likely to find bugs. Intuitively, each subdomain represents a set of similar test cases; we want to maximize the benefit of our testing by picking dissimilar test cases – that is, test cases from different subdomains.

In the best case, a subdomain is *revealing*. This means that every test case in it either causes the program to fail or to succeed. The subdomain thus groups together truly equivalent cases. If all subdomains are revealing, a test suite that satisfies the criterion will be complete, since we are guaranteed that it will find any bug. In practice, it's very hard to get revealing subdomains though, but by careful choice of subdomains it's possible to have at least some subdomains whose error rate – the proportion of inputs that lead to bad outputs – is much higher than the average error rate for the input space as a whole.

11.5 Subdomain Criteria

The standard and most widely used criterion for program-based testing is *statement coverage*: that every statement in the program must be executed at least once. You can see why this is a subdomain criterion: define for each program statement the set of inputs that causes it to be executed, and pick at least one test case for each subdomain. Of course the subdomain is never explicitly constructed; it's a conceptual notion. Instead, one typically runs an instrumented version of the program that logs which statements are executed. You keep adding test cases until all statements are logged as executed.

There are more burdensome criteria than statement coverage. *Decision coverage* requires that every edge in the control flow graph of the program be executed – roughly that every branch in the program be executed both ways. It's not immediately obvious why this is more stringent that statement coverage. Consider applying these criteria to a procedure that returns the minimum of two numbers:

```
static int minimum (int a, int b) {
    if (a ≤ b)
        return a;
    else
        return b;
```

For this code, statement coverage will require inputs with a less than b and vice versa. However, for the code

```
static int minimum (int a, int b) {
    int result = b; if (b ≤ a)
        result = b;
    return result;
```

a single test case with b less than a will produce statement coverage, and the bug will be missed. Decision coverage would require a case in which the if-branch is not executed, thus exposing the bug.

There are many forms of *condition coverage* that require, in various ways, that the boolean expressions tested in a conditional evaluate both to true and false. One particular form of condition coverage, known as MCDC, is required by a DoD standard for safety critical software, such as avionics. This standard, DO-178B, classifies failures into three levels, and demands a different level of coverage for each:

Level C: failure reduces the safety margin
Example: radio data link

Requires: statement coverage

Level B: failure reduces the capability of the aircraft or crew
Example: GPS
Requires: decision coverage

Level A: failure causes loss of aircraft
Example: flight management system
Requires: MCDC coverage

Another common form of program-based subdomain criterion is used in *boundary testing*. This requires that the boundary cases for every conditional be evaluated. For example, if your program tests $x < n$, you would require test cases that produce $x = n$, x=n-1, and x=n+1.

Specification-based criteria are also usually cast in terms of subdomains. Because specifications are usually informal – that is, not written in any precise notation – the criteria tend to be much vaguer. The most common approach is to define subdomains according to the structure of the specification and the values of the underlying data types. For example, the subdomains for a method that inserts an element into a set might include:

- the set is empty
- the set is non-empty and the element is not in the set
- the set is non-empty and the element is in the set

You can also use any conditional structure in the specification to guide the division into subdomains. Moreover, in practice, testers make use of their knowledge of the kinds of errors that often arise in code. For example, if you're testing a procedure that finds an element in an array, you would likely put the element at the start, in the middle and at the end, simply because these are likely to be handled differently in the code.

11.6 Feasibility

Full coverage is rarely possible. In fact, even achieving 100% statement coverage is usually impossible, at the very least because of defensive code which should never be executed. Operations of an abstract data type that have no clients will not be exercised by any system-level test case, although they can be exercised by unit tests.

A criterion is said to be *feasible* if it is possible to satisfy it. In practice, criteria are not usually feasible. In subdomain terms, they contain empty subdomains. The practical

question is to determine whether a particular subdomain is empty or not; if empty, there is no point trying to find a test case to satisfy it.

Generally speaking, the more elaborate the criterion, the harder this determination becomes. For example, *path coverage* requires that every path in the program be executed. Suppose the program looks like this:

if C1 then S1;
if C2 then S2;

Then to determine whether the path $S1;S2$ is feasible, we need to figure out whether the conditions $C1$ and $C2$ can both evaluate to true. For a complex program, this is a non-trivial task, and, in the worst case, is no easier than determining the correctness of the program by reasoning!

Despite these problems, the idea of coverage is a very important one in practice. If there are significant parts of your program that have *never* been executed, you shouldn't have much confidence in their correctness!

11.7 Practical Guidance

It should be clear why neither program-based nor specification-based criteria are alone good enough. If you only look at the program, you'll miss errors of omission. If you only look at the specification, you'll miss errors that arise from implementation concerns, such as when a resource boundary is hit and some special compensating behaviour is needed. In the implementation of the Java *ArrayList*, for example, the array in the representation is replaced when it's full. To test this behaviour, you'll need to insert enough elements into the *ArrayList* to fill the array.

Experience suggests that the best way to develop a good test suite is to use specification-based criteria to guide the development of the suite, and program-based criteria to evaluate it. So you examine the specification, and define input subdomains. Based on these, you write test cases. You execute the test cases, and measure the code coverage. If the coverage is inadequate, you add new test cases.

In an industrial setting, you would use a special coverage tool to measure code coverage. In 6170, we don't expect you to learn to use another tool. Instead, you should just consider your test cases carefully enough that you make an argument that you have achieved reasonable coverage.

Runtime assertions, especially representation invariant checks, will dramatically amplify the power of your testing. You'll find more bugs with fewer cases, and you'll

track them down more easily.