CSSE4630 Lecture Notes: Dynamic Program Analysis

Mark Utting

Weeks 11-12, version 1.0

1 Overview

In this part of the course, we look at *dynamic program analysis*, which is about analysing programs by executing them. This covers a broad range of topics, including:

- testing (blackbox and whitebox)
- runtime monitoring / verification
- model-checking
- memory usage analysis
- performance analysis
- and more...

Recall that software testing can be divided into two main approaches:

- **Black-box testing**: where we view the software as an opaque black box, so the code is not visible. We focus on testing the input-output behaviour of the software, without observing any internal behaviour of the software.

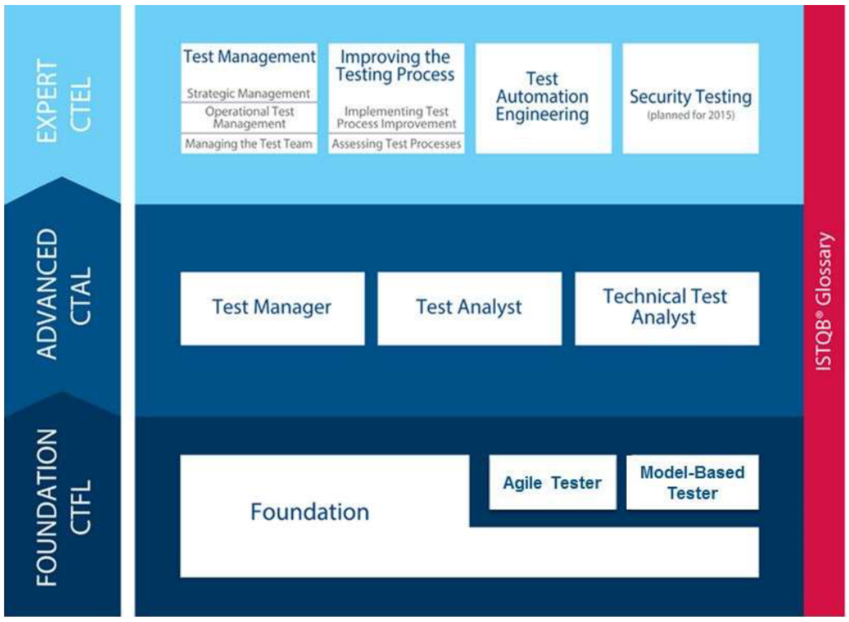
- **White-box testing**: where we can see the internal structure of the software, such as its source code, so we can observe the program as it executes.

In this course we focus only on white-box approaches, since we are interested in analysing programs (usually with the source code available, or at least the bytecode).

More specifically, we will cover several white-box topics, which can be grouped into the following areas:

1. measuring coverage (metrics for measuring how much of the program has been executed);
2. increasing coverage (ways of increasing those metrics);
3. checking properties (via observing, or testing, or model-checking)

We will refer to some testing concepts from the ISTQB (International Software Testing Qualifications Board) syllabii and glossary. These are all available from `https://www.istqb.org/downloads.html`.



2 Measuring Coverage

There are many different metrics for determining how much of a program's behaviour has been covered during testing. We will focus on the following white-box '*structural coverage*' metrics:

- Statement coverage (SC)
- Decision coverage (DC)
- Modified Condition/Decision Coverage (MC/DC)
- Multiple Condition coverage (MCC)
- Path coverage, or Basis Path coverage [optional]
- Mutation testing coverage

The first five of these are described in the ISTQB **Advanced Level Syllabus Technical Test Analyst** (2019). You should already be familiar with at least the first two — they are also covered in the **Certified Tester Foundation Level Syllabus**.

Here is a brief definition of each kind of coverage metric above, plus a few less-common intermediate metrics for completeness (adapted from [UL07]). For each kind of coverage we describe the requirements to get 100% coverage — but in practice we measure the percentage of this goal that is achieved (0 . . . 100%).

Note the terminology: each **if** or **while** statement contains a *decision* that determines the branching of the control flow, and that decision contains one or more primitive *conditions*, which may be combined by disjunction, conjunction and negation operators.

Statement Coverage (SC): the test suite must execute every reachable statement.

Decision Coverage (DC): (also called **branch coverage**) the test suite must ensure that each reachable *decision* is made true by some tests, and false by other tests. *Decisions* are the branch criteria which modify the flow of control in selection and iteration statements etc. For a test suite to satisfy decision coverage, we also require it to satisfy statement coverage (for example, this ensures that straight-line methods with no decisions are executed).

Condition Coverage (CC): A test set achieves CC when each condition in the program is tested with a true result, and also with a false result. For a decision containing N conditions, two tests can be sufficient to achieve CC (one test with all conditions true, one with them all false), but dependencies between the conditions typically require several more tests.

Decision/Condition Coverage (D/CC): A test set achieves D/CC when it achieves both decision coverage (DC) and CC.

Full Predicate Coverage (FPC): A test set achieves FPC when each condition in the program is forced to true and to false, in a scenario where that condition is *directly correlated* with the outcome of the decision. A condition c is directly correlated with its decision d when either $d \iff c$ holds, or $d \iff \textit{not}\ c$ holds. For a decision containing N conditions, a maximum of $2N$ tests are required to achieve FPC.

Modified Condition/Decision Coverage (MC/DC): This strengthens the *directly correlated* requirement of FPC by requiring the condition c to *independently affect* the outcome of the decision d. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. For a decision containing N conditions, a maximum of $2N$ tests are required to achieve MC/DC (the same upper bound as FPC), so the number of tests is proportional to the complexity of the decision (the number of conditions within it).

Multiple Condition Coverage (MCC): A test set achieves MCC if it exercises all possible combinations of condition outcomes in each decision. This requires up to 2^N tests for a decision with N conditions, so is practical only for simple decisions (for example, that contain less than 5 conditions).

Path Coverage (PC): the test suite must execute every satisfiable path through the control-flow graph. For programs with loops, this may require an infinite number of paths, which is not practical to test. So in practice we may prefer the next criteria, which is weaker, but more achievable.

Basis Path Coverage: this is based on covering the control flow graph (CFG) of the program. Firstly the most important (non-exception) path through the CFG is tested. Then each decision along that path is changed, starting from the first one, while trying to leave other decisions unchanged as much as possible. This process continues until all decisions in the CFG have been exercised. For more details, see the ISTQB Advanced Level Syllabus Technical Test Analyst syllabus.

Figure 1 shows the relationships between most of these coverage criteria. Statement coverage is the weakest coverage metric, while MCC is the strongest of the metrics in the figure. Path coverage is stronger than Basis Path Coverage, which is stronger than Decision Coverage, but it does not necessarily test all the conditions within each decision, so does not guarantee condition coverage.

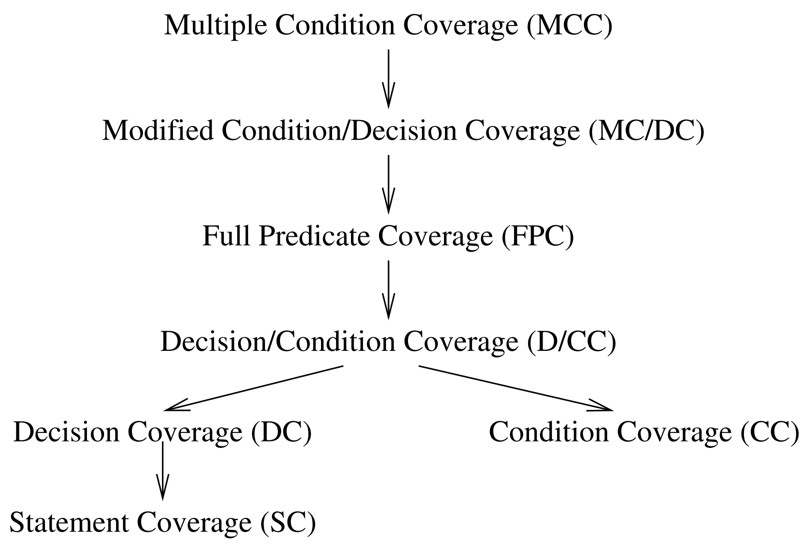

Multiple Condition Coverage (MCC)
Modified Condition/Decision Coverage (MC/DC)
Full Predicate Coverage (FPC)
Decision/Condition Coverage (D/CC)
Decision Coverage (DC)
Condition Coverage (CC)
Statement Coverage (SC)

Figure 1: The hierarchy of control-flow coverage criteria for multiple conditions. $A \longrightarrow B$ means that criteria A is stronger than (subsumes) criteria B.

Exercise: Here is an example Scala method from tip/analysis/IntervalLattice.scala.

```
/**
  * Abstract '==' on intervals;
  */
def eqq(a: Element, b: Element): Element =
  (a, b) match {
    case (FullInterval, _) => FullInterval
    case (_, FullInterval) => FullInterval
    case ((IntNum(l1), IntNum(h1)), (IntNum(l2), IntNum(h2))) =>
      if (l1 == h1 && h1 == l2 && l2 == h2)
        (IntNum(1), IntNum(1))
      else
        (IntNum(0), IntNum(1))
    case _ =>
      (IntNum(0), IntNum(1))
  }
```

Imagine that we test it with two test inputs:

- `eqq(FullInterval, FullInterval)`
- `eqq( (0,2), (0,3) )`

What coverage have we achieved, for each of the first four metrics?

- Statement coverage (SC)?
- Decision coverage (DC)?
- Modified Condition/Decision Coverage (MC/DC)?
- Multiple Condition coverage (MCC)?

2.1 Mutation Testing

We will use the PIT tool from `https://pitest.org` to measure mutation testing coverage in Java programs.

They say: *"PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling."*

They define mutation testing in just 51 words:

> Mutation testing is conceptually quite simple.
>
> Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived.
>
> The quality of your tests can be gauged from the percentage of mutations killed.

In other words:

$$mutation\_score = killed/total\_mutations$$

Note that you can install PIT via your build tool (Ant, Gradle, etc), or by installing a specific IntelliJ IDEA plugin. See the assignment for detailed setup instructions.

3 Increasing Coverage

In this section we will briefly go over several test design strategies for generating tests for a given **system under test** (SUT), to try to increase the coverage of its behaviour and/or of its code. We will start with the simplest and dumbest test generation methods (monkey testing) and work our way up to smart test-generation techniques that try to maximise coverage of the SUT code (white-box techniques), or its expected behaviour (black-box techniques), etc.

3.1 Monkey Testing and Fuzz Testing

Random testing, or 'monkey testing', is where we just throw random values at the system under test to see if we can make it crash. Random generation can be surprisingly effective in this goal!

The term 'Fuzz testing' (or 'fuzzing') is used to cover a wide variety of techniques, ranging from random testing to sophisticated concolic testing. But the common idea is again that we generate inputs to try and make the SUT crash.

There are several kinds of fuzz testing, including [FBJ$^+$16, Section 5.3.4]:

- random fuzzing = the same as monkey testing.
- mutation fuzzing = we take existing input values and 'fuzz' them (e.g. mutate them to be slightly different), so that we try many variants around each known input value.
- generation-based fuzzing = use a model of the input (e.g. a grammar) or of the expected vulnerabilities, and generate input test data from that model. This is similar to model-based testing (MBT), which we will study shortly. The difference is that the model here is just a model of the input of the SUT, whereas with MBT the model usually covers the input-output behaviour of the SUT.

- gray-box fuzzing = observing the code of the SUT as different inputs are tried, and adjusting the inputs correspondingly. These days this is called *concolic testing* and we will study it shortly.

3.2 Model-Based Testing (MBT)

MBT is a black-box test generation technique that aims to systematically test the behaviour of the SUT. The input-output behaviour.

The idea of MBT is that we design an abstract model, in some suitable notation or language, which expresses part of the desired behaviour of the SUT. Then we generate tests from that model, systematically covering all aspects of the model in some way [UL07]. If our model accurately captures some aspects of the SUT behaviour, then the generated tests should systematically cover that expected behaviour, and detect any areas where the SUT real behaviour differs from the expected model.

For this presentation, we will just use simple finite state machine (FSM) models.

Example:

Some common Test Generation Algorithms from FSMs include (from weaker to stronger):

- state coverage = we generate tests that visit every state at least once;
- transition coverage = we generate tests that take each transition at least once (e.g. use the Chinese postman algorithm);
- transition-pair coverage = make sure we test each pair of adjacent transitions;

A useful algorithm for generating the shortest tour of all the transitions is the Chinese Postman algorithm, invented by Meigu Guan (Mei-Ko Kwan) in 1962 [Gua62]. See `https://dzone.com/articles/solving-the-chinese-postman-problem` for the algorithm and some nice animations, or `http://www.harold.thimbleby.net/cpp/index.html` for another description and a Java implementation.

Exercise: Use the Chinese Postman algorithm to generate an all-transitions tour of your model from the example above.

3.3 Property-Based Testing (PBT)

The idea of PBT is that we write down a property that we expect the SUT to satisfy, then we use a 'QuickTest' tool (the original QuickTest tool was for Haskell, but now similar tools are available for most languages) to generate lots of inputs to try and find a counter-example to that property — that is, some inputs that make the property false.

If you want a general introduction to PBT and jqwik, here is an excellent introductory blog article by Johannes Link in the September 2019 Java magazine from Oracle:

- `https://blogs.oracle.com/javamagazine/know-for-sure-with-property-based-testing`

Here is a brief description of properties, from the jqwik user manual[1]:

> Properties are the core concept of property-based testing.
>
> You create a Property by annotating a public, protected or package-scoped method with @Property. In contrast to examples a property method is supposed to have one or more parameters, all of which must be annotated with @ForAll.
>
> At test runtime the exact parameter values of the property method will be filled in by jqwik.
>
> Just like an example test a property method has to:
>
> - either return a boolean value that signifies success (true) or failure (false) of this property.
> - or return nothing (void). In that case you will probably use assertions to check the property's invariant.
>
> If not specified differently, jqwik will run 1000 tries, i.e. a 1000 different sets of parameter values and execute the property method with each of those parameter sets. The first failed execution will stop value generation and be reported as failure - usually followed by an attempt to shrink the falsified parameter set.

And a short example (which finds a failure case!):

```
import net.jqwik.api.*;
import org.assertj.core.api.*;

class PropertyBasedTests {

    @Property
    boolean absoluteValueOfAllNumbersIsPositive(@ForAll int anInteger) {
        return Math.abs(anInteger) >= 0;
    }
}
```

Here is a longer presentation on property-based testing, by John Hughes and Johannes Link, with extensive examples of detecting buggy implementations of binary search trees: `https://github.com/jlink/how-to-specify-it`.

Exercise: Use Property-Based Testing to test an all-transitions tour of your model from the example above.

[1]See `https://jqwik.net/docs/current/user-guide.html`, accessed 18-Oct-2020.

3.4 Concolic Testing

TO BE ADDED

References

[FBJ+16] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one - security testing: A survey. volume 101 of *Advances in Computers*, pages 1 – 51. Elsevier, 2016.

[Gua62] M. Guan. Graphic programming using odd and even points. *Chinese Mathematics*, 1:273–277, 1962.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier, 2007.