How to implement easily

Effective Test Coverage Assessment

Gebhard Greiter, 2012

One of the most difficult aspects of testing is answering the question how complete a test-suite we use actually is. The question we should be able to answer is:

Which part of my code is not yet tested?

To get an answer we need practical metrics and corresponding code instrumentation.

Classical metrics are:

- **Statement Coverage**
- **Branch Coverage**
- **Path Coverage**

But: **Path** coverage is not useful at all because most programs contain looping statements (so that the number of possible paths may not even be finite).

The usual way to measure **Branch** as well as **Statement** coverage is to measure **Block Coverage**:

A **Block** in this sense is always a pair (**S**,**A**) such that

- **S** is a sequence of atomic statements
  - — starting with a statement to which we can jump,
  - — ending in a jump statement
  - — and not containing any other jump statement.
- **A** is the address of a statement to which the jump statement could lead us.

By an atomic statement we mean a statement atomic in the source code language (usually one line of code if you program in C, C++, Java, or C#). Note: Conditional statements - ***if*** and ***switch*** - are not atomic and are always seen as at least two pairs (**S**,**A**).

In contrast to **Path** coverage, **Block** coverage *will* make sense and is the most precise information on code coverage you can hope to get.

For a practitioner however all these classical metrics have four serious problems:

- **Problem 1:** To know the percentage of Block coverage generated by your test is certainly interesting but will tell you nothing about how to create more test cases in order to get better code coverage.
- **Problem 2:** Even more: You will not know whether the coverage you achieved so far is quite good or is — compared with what testers typically achieve — very poor and not enough (no typical numbers have been published so far).
- **Problem 3:** Code instrumentation to measure Block Coverage requires a quite sophisticated tool (a tool usually not available).
- **Problem 4:** If you are to test software as a Black Box (e.g. for acceptance test), you might not have access to the source code at all.

So you see:

More practical metrics are needed.

And they really exist:

- **Case 1:** You are allowed to see and to recompile the source code of the application under test.

  In this case I usually use what I call **Return Door Coverage**: It is quite easy to write a simple utility allowing us to instrument code to the effect that whenever execution returns from the block implementing a method call, a corresponding counter (representing the exit point) is augmented.

  Because the set of all the watch points you get is much smaller than the set of all the pairs (S,A) in the sense of Block Coverage, it may well make sense to say that an application is not yet tested good enough as long as Exit Coverage is lower than e.g. 70%.

  Even better: The protocol you get will tell you precisely how often each method was called (and also how often it did exit any specific exit door).

- **Case 2:** You do not have access to the source code of the application under test.

  In this case all the counters you use can only be counters in your test drivers. These counters then are to represent aspects under test.

  The first step in designing your test suite is then to define all aspects you want to test; assign identifiers to them so that every method of your test suite can contain logging statements writing to stdout or a log file aspect identifiers (the identifiers of aspects, e.g. functions, tested by such a method call).

  The test coverage you need to aim at is then 100% **Aspect Coverage**.

So you see: ***Both of these metrics — often applied in parallel — produce really valuable information on how thoroughly your code was testet.*** I recommend them highly (and use them all the time).