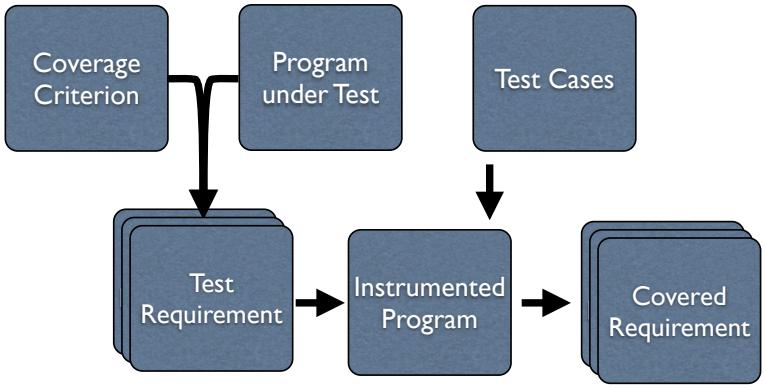
Measuring Code Coverage



1

A coverage criterion can be seen as a finite set of test requirements that a test suite should fulfill. There is usually more than one way to cover a test requirement, so a coverage criterion is not a unique description of a test suite. To make use of coverage in practice we need to measure it. This is done by instrumenting the source code with an instrument for every single test requirement, described by the coverage criterion. When test cases are run on the instrumented program the instrumentation keeps track of what has been executed, and so at the end of the execution

Coverage Value

$$\text{Coverage value} = \frac{\#\text{ Covered test requirements}}{\#\text{ Total test requirements}}$$

I've got 100% statement coverage on my program. How many bugs are left?

2

Coverage can be used in three ways: 1) Adequacy: Have I got enough tests? 2) Guidance: Where should I test more? 3) Automation: Generate test that satisfies a test requirement
Coverage is usually quantified as the percentage of test requirements satisfied. But what does that mean?

Coverage is dangerous

- Developers write test only to satisfy coverage
- 100% coverage can detect no faults:
  Coverage measures what is *executed*, not what is *checked*
- Coverage metrics tell you what code is not tested, but cannot accurately tell you what code is tested:
  - Low coverage means code is not well tested
  - But high coverage does not mean code is well tested

3

The use of coverage has some dangerous aspects, that might even reduce the quality of testing. If success is only quantified in a coverage metric, developers will get very efficient and writing test cases that satisfy the coverage goals, but not at finding bugs. Also, it is possible to cover the entire program without detecting a single bug - testing is more than just input generation (see Mutation testing lecture).

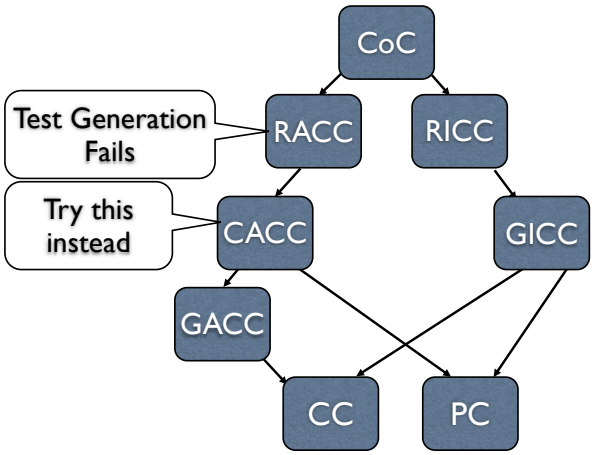
Coverage is useful

- It always tells you where you haven't tested
- Testing everything a bit is better than not testing most of the program - unless you know where the faults are
- Coverage != coverage
  Stricter criterion → more tests
- More tests = more chances of hitting bugs

4

Despite it's downsides coverage has some useful sides: It is very efficient at telling you which parts of a program you haven't tested at all. Intuitively, testing everything a little bit should be better than testing some aspect thoroughly and neglecting the rest - unless you already know where the bugs are (which you don't in general).

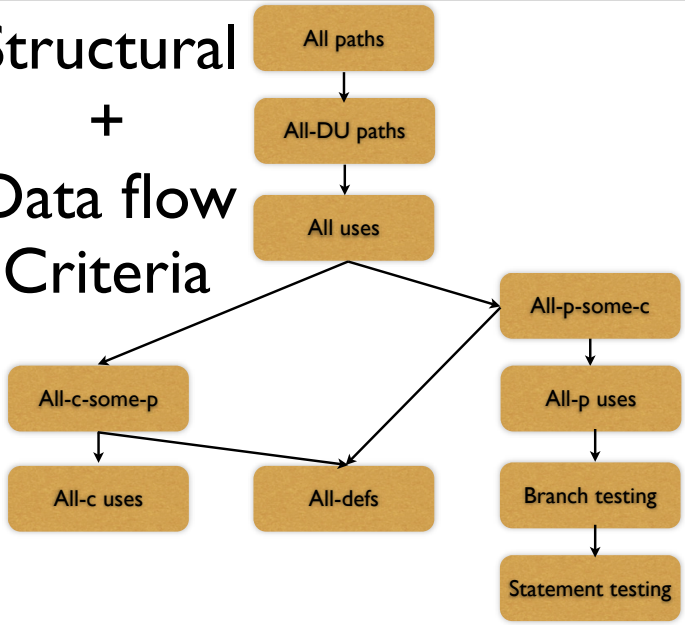
Best Effort Strategy



5

An arrow from one criterion to another means that the former subsumes the latter.
This means that if we test for CoC, we will automatically satisfy all other coverage criteria as well.
Determining infeasibility is undecidable, so what can we do in practice if we can't find a test case for a particular test requirement? A simple solution is to use a best effort approach: If, after some time, we cannot find a test case for a test requirement, we simply turn to the next simpler version of the same predicate in the subsumption hierarchy.

Structural + Data flow Criteria



6

All-definitions: One DU path for each definition
All-c-uses: One DU path for each definition-c-use pair
All-p-uses: One DU path for each definition-p-use pair
All-c-uses-some-p-uses: One DU path for each definition-p-use pair
If there is no p-use, then one c-use
All-p-uses-some-c-uses
All-DU-pairs: One DU path for each def-use pair: = All-p-uses +all-c-uses, also known as All-Uses
All-DU-paths: All DU paths