Test Coverage and Risk

Hans Schaefer
Consultant
5281 Valestrandsfossen, Norway
hans.schaefer@ieee.org
http://home.c2i.net/schaefer/testing.html

How to check that a test was good enough?

A test is never „good enough“, it is always a sample.

The criteria in this chapter can be used as a compromise between the Ideal and a simple measurement.

How precise measurement?

As precise as needed.

Measuring costs time and money

- **the measurement itself**
- **tools**
- **interpretation**

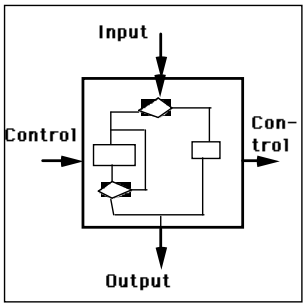
More precise measurement only for research projects.
Typically, you measure what you can measure automatically.

Test Coverage in Programs

Theory:

The more details you test, the more faults you find.

Thus, a test with higher coverage is (theoretically) better.



Why measure coverage

- a measure of how much testing was done
- useful as exit criterion
- easy to measure with automated tools
- some relation to quality
- you know where holes in your test are

Some more comments, from Cem Kaner, author of the book “Testing Computer Software”:

On the issue of developer testing, I agree with everyone who says that developers should do careful, planned glass box testing. (I call it "glass box" because you're looking inside the box when you do this stuff. White boxes are as opaque as black ones.)

The developer is in the best position to understand the risks associated with many of the conscious design trade-offs, and is in the best position to understand the various optimizations provided for special circumstances.

The developer, if she has time, is in the best position to conduct the series of tests needed to achieve complete line coverage, or branch coverage, or predicate coverage, because she will understand the intent behind the lines and the issues that can be presented to nontrivially test those lines, or branches, or whatever.

If the developer doesn't have enough time to do this testing, somebody else will have to do it, but less efficiently. The developer's testing will help her assure that the code does what she think it's supposed to do.

This is important -- too many bugs coming out of development are obvious bugs that would have been fixed if the programmer had done much testing.

But the developer is only going to confront **some** of her own blind spots if she does her own testing. Therefore (at least, I think this is a "therefore") we want someone else besides the developer to do a lot of testing too.

This isn't an issue of programmer incompetence (though we sometimes do have to deal with that). And it is not an issue of programmer bad faith (again, if there is bad faith, we must include independent testing as one of a set of compensating and corrective measures). And it is not that the programmer doesn't have enough time to do her own testing (though in my world, programmers are rarely given the time they need to do the testing they should do).

Even when the programmer is well rested, has the time, the inclination, and the skill to do the testing, she will still have blind spots and someone else can find things that she will miss.

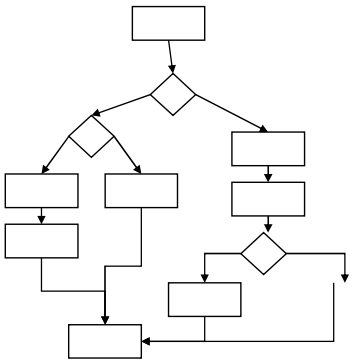
In the ideal world (which is, I am told, common practice in some companies), I think that we want extensive testing by the authoring programmer, plus extensive testing/inspection by some other programmer.

Whether these folk are doing module testing exclusively, or module testing plus some functional testing, or system-level integration testing, I hope that their work is driven by a deep understanding of the code, and that their test cases are designed with reference to the code. If so, they are doing "glass box testing."

Definition of (white box) coverage criteria

Statement Coverage
Branch Coverage
Condition Coverage
Multi-Condition Coverage
.....
Path Coverage

Data Coverage
Interface Coverage



Decision or branch coverage (DC)
every decision tested in each possible outcome
Condition/Decision coverage (C/DC)
as above plus, every condition in each decision tested in each possible outcome
Modified Condition/Decision (MC/DC)
as above plus, every condition shown to independently affect a decision outcome (by varying that condition only) (minimal multicondition, or meaningful impact)
Multiple-condition coverage (M-CC)
all possible combinations of conditions within each decision taken

Path Coverage

Percentage of paths through the program which are covered. Impossible to reach 100% if loops are in the program.

Data Coverage

Coverage of set-use pairs. I.e.: initialize - read, Write - read.

Interface Coverage

Every interface is covered. See under Integration test.

Example program

```
Begin
Read date
If (YY = leapyear and MM = February)
   then
      DD := 29
End if
Print date
End
```

Test cases

28 Feb 2004 -> 100% statement coverage, 50% Branch
31 Mar 2003-> 100% statement and Branch and condition

Multicondition coverage requires four test cases, all combinations:
28 Feb 2003
28 Feb 2004
31 March 2003
31 March 2004

Some coverage tools may be extremely slow, because they write the whole execution history to disk.

Exercise

Consider the example program.

Introduce a fault: “and” replaced by “or”.

Consider the example test cases on the page before:

Does any coverage criterion guarantee that THIS fault is found?

The minimum coverage criteria

Depends on RISK:

For highly critical programs (airborne systems, RTCA-DO-178B):

- **"Modified decision condition coverage"(MCDC).**

For less critical ones:

- **Branch coverage, then statement coverage.**

General experience:

- **Near 100% statement and near 85% Branch coverage (by Review or test or both)**

Coverage easy to achieve for logical and algorithmic functions, not easy for communication and I/O intensive functions (Reuben Gallant, EuroSTAR 99)

Keith Miller, A modest proposal for software testing, IEEE Software, 2/2001

Coverage Criteria for Integration Test

Test every interface statement (Call, invoke)

All real time coordination (Start, stop, wait, resume, break,...)

Global Data flow: All In-out-Pairs through buffers, data bases, files, global variables etc.

All database and file access statements

Coverage Criteria for System Test

70 to 85% Branch coverage.

This is an experience value from general industry practice.

How to measure coverage

Use a fully automated Test Coverage Analysis Tool.

Tools exist for your languages.

They consist of three parts:

- Preprocessor: Puts count statements into the code.
- Runtime routine: Count-statements count execution and write statistics to a file or table.
- Postprocessor: Sums up statistics and reports results.

Problem

The probes (count statements) have influences on execution time of the code. Thus, to test such issues, the code must first be tested with coverage measurement, then again without.

Caution!

Coverage can lead away from the real goal of testing:

- **Finding as many defects as possible in as short a time as possible.**
- **Finding the highest risk defects.**

White box test coverage does not find faults of omission!

But: A coverage goal can help to control testing, as programmers will take a more serious look towards testing.

Exercise

Below you find a list of data about some modules having been tested. Module complexity is given by the number of code lines and code branches. Test coverage is given by the percentage of branches covered during testing. Discuss if some modules should be tested more.

Module no.	# program lines	# branches	covered % of branches
1	200	15	93
2	50	10	100
3	50	5	40
4	120	15	80
5	600	5	80
6	100	48	98
7	100	10	90
8	80	30	30
9	50	5	100
10	50	5	20
11	80	7	0
12	60	10	80
13	400	125	74

Coverage and the British Standard for Software Component Testing

The Standard measures coverage two ways:

Coverage of test case generation methods applied
Coverage of code (control and data flow)

British Standard BS 7925-2 “Software Component Testing” and
BS 7925-1 “Vocabulary of Terms in Software Testing”

Test coverage for changes

Execute all changed statements.

Execute all code taking data from the changed area.

Execute all code which could be affected through different time characteristics.

If this is too complicated: Do a **full regression test!**

Summary

Define and follow up coverage.
First a black box test.
Then interpretation of measurements.
Then more test to achieve coverage.
Then inspection of untested parts.

No coverage criterion guarantees error free code.

Literature

Glenford Myers, "The Art of Software Testing“, 1979.

Boris Beizer, "Software Testing Techniques”, 1990.

Thomas McCabe: Structured Testing, IEEE Tutorial, IEEE Catalog No. EHO 200-6, 1983.

Hewlett Packard Journal, June 1987, pg.. 13 ff.

Harry M. Sneed, "Data Coverage Measurement in Program Testing", Proceedings of the Workshop on Software Testing, 15. - 17. July 1986, IEEE Catalog no. 86TH0144-6, pg. 34 ff.

Mary J. Harrold and Mary Lou Soffa, “Selecting and Using Data for Integration Testing, IEEE Software, March 1991.

Joseph R. Horgan, Saul London, Michael R. Lyu, “Achieving Software Quality with Testing Coverage Measures”, IEEE Computer, Sept. 1994.

Cem Kaner, James, Bach , Bret Pettichord, LESSONS LEARNED IN SOFTWARE TESTING, Wiley, 2001.

Andreas Spillner und Tilo Linz, Basiswissen Softwaretest, dpunkt Verlag 2002, 2003 (in German).