10 criteria for selecting a code coverage tool

In order to develop safe and reliable software, testing is an indispensable part of quality assurance. Without sufficient and documented tests, it is impossible to determine whether software is secure and functionally correct. The measurement of code coverage (test coverage) is particularly important in this context. This is because it can be used to determine how comprehensively a piece of software has already been tested. Code coverage indicates the ratio of tested code to total code. In simplified terms, for example, code coverage is 75% if three out of four possible options are run during the test.

Particularly in safety-critical software development, industry standards prescribe precise requirements for code coverage, so that products cannot be certified here without proof of sufficient test coverage. But also in other development projects, companies increasingly attach great importance to software quality and measure code coverage.

Various code coverage analyzers are available on the market for measuring code coverage. They differ significantly in terms of handling and quality. For this reason, we show ten basic criteria for selecting a code coverage tool:

1. Independence from compiler

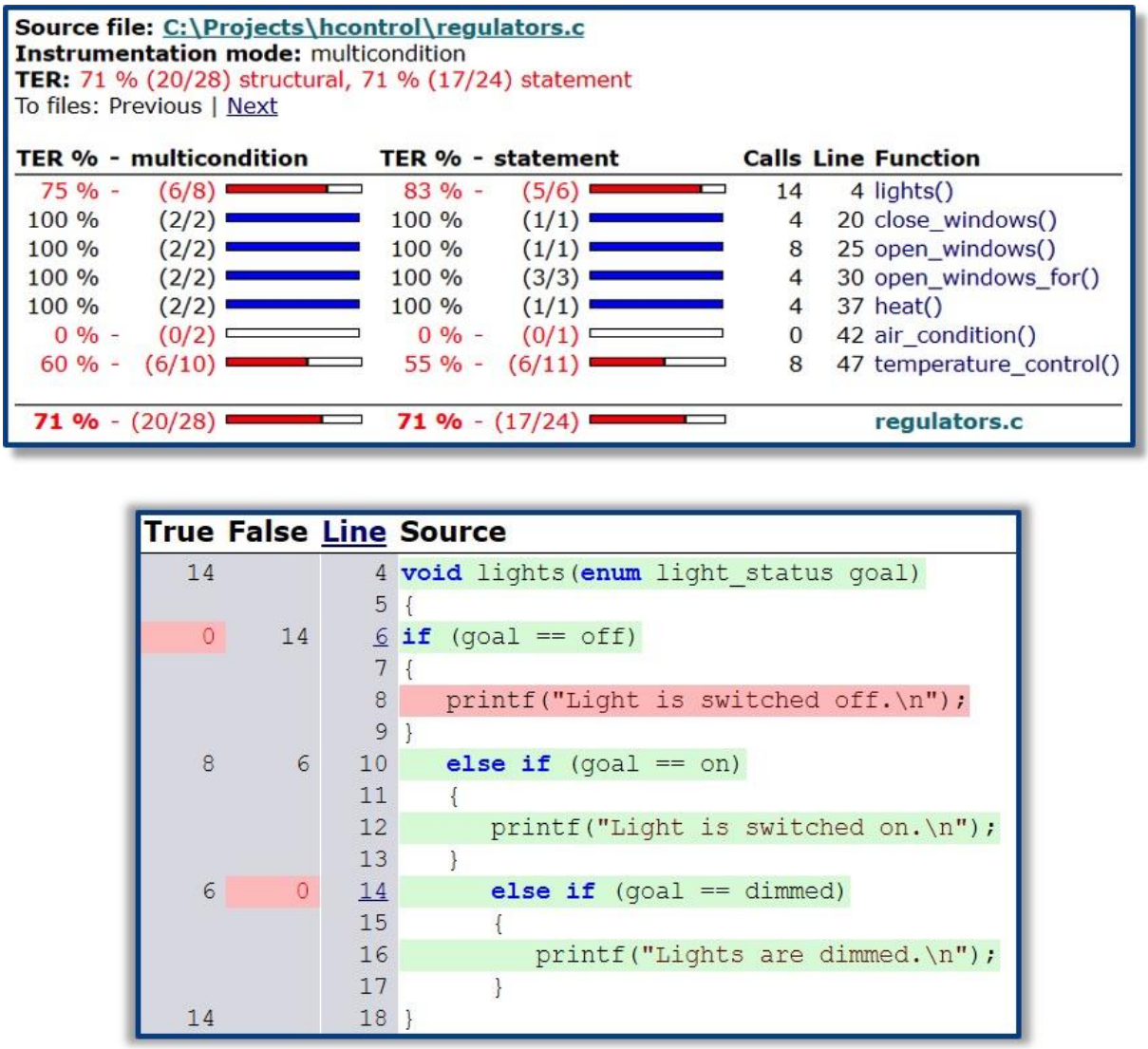
Of course, a code coverage tool must work with the compiler used in the project. However, it makes a lot of sense to rely on a tool that can be used independently of the compiler right from the start. Such tools can be used then in all projects and also in the current project in the case of a change of the compiler. A coverage tool that can be used compiler-independently can be used in a much more diverse way and is therefore a worthwhile investment.

2. Ease of use

The best software is reluctantly (and thus rarely) used if it is unnecessarily complicated or not well thought out. Simple handling, on the other hand, can significantly increase the user's acceptance of the use of a test coverage tool. Ideally, the tool runs in the background and does not generate any additional work for the user during testing.

3. Comprehensibility of the coverage reports

When evaluating the coverage reports, it should be clear at a glance which parts of the code have already been tested and where coverage is still lacking. With good coverage tools, the tester can easily identify at source code level which test cases are still outstanding. By executing these missing tests, the code coverage can then be increased in a targeted manner. At the same time, this avoids unnecessary work that would result from redundant tests.



Source file: C:\Projects\hcontrol\regulators.c
Instrumentation mode: multicondition
TER: 71 % (20/28) structural, 71 % (17/24) statement
To files: Previous | Next

TER % - multicondition	TER % - statement	Calls	Line	Function
75 % - (6/8)	83 % - (5/6)	14	4	lights()
100 % (2/2)	100 % (1/1)	4	20	close_windows()
100 % (2/2)	100 % (1/1)	8	25	open_windows()
100 % (2/2)	100 % (3/3)	4	30	open_windows_for()
100 % (2/2)	100 % (1/1)	4	37	heat()
0 % - (0/2)	0 % - (0/1)	0	42	air_condition()
60 % - (6/10)	55 % - (6/11)	8	47	temperature_control()
71 % - (20/28)	**71 %** - (17/24)			**regulators.c**

True	False	Line	Source
14		4	`void lights(enum light_status goal)`
		5	`{`
0	14	6	`if (goal == off)`
		7	`{`
		8	`printf("Light is switched off.\n");`
		9	`}`
8	6	10	`else if (goal == on)`
		11	`{`
		12	`printf("Light is switched on.\n");`
		13	`}`
6	0	14	`else if (goal == dimmed)`
		15	`{`
		16	`printf("Lights are dimmed.\n");`
		17	`}`
14		18	`}`

Figure: In addition to an overview of the code coverage of the individual code parts (upper), an effective code coverage tool such as Testwell CTC++ also displays detailed information (lower) that shows exactly to what extent the source code is covered by tests, even for the highest coverage levels. (Source: Verifysoft Technology)

4. Support of higher coverage levels for safety-critical development

For the testing of safety-critical software, the standards (e.g. ISO 26262 in the automotive sector, DO-178C in aviation and EN-50128 in rail transport) stipulate high coverage levels up to MC/DC coverage. It is therefore imperative to ensure that the coverage tool supports all required coverage levels. In order to be able to use a solution in the long term, not only current, but also future requirements should be taken into account. Important to know: many coverage tools offer only decision or branch coverage and are therefore insufficient for safety-critical software development.

5. Flexible Integration

Even within a company, development environments and tool chains are often very heterogeneous. A coverage tool should easily cope with all these different environments. Integration into the respective build process and into the execution of tests must be possible seamlessly and without great effort. Provided that the tool can also be used via the command line, advantages are offered in the creation of automated builds.

6. Low Instrumentation Overhead

Most coverage tools measure code coverage by instrumenting the source code. The source code is enriched by the coverage tool with “counters”, which count where and how often the relevant code parts were executed during testing. However, this increases the size of the original code. When testing on embedded targets that have limited memory, care should therefore be taken to keep this so-called instrumentation overhead as low as possible. The differences in memory requirements between the individual code coverage tools are sometimes considerable. The code coverage analyzer Testwell CTC++ from Verifysoft Technology, for example, is very resource-efficient in this respect. With Testwell CTC++ it is even possible to further reduce the required memory space again by using the bit coverage option (bit coverage then only measures whether a code part has been tested, but not how often it was tested).

7. Support of different programming languages

Companies often work with different programming languages or plan to introduce additional languages in the future. It therefore makes sense to choose a tool that supports all or as many of these languages as possible right from the start.

8. Support for “creative” programming

Some coverage tools get problems when analyzing language constructs that deviate from common standards or have a high nesting depth. However, a good tool for measuring test coverage should also be able to cope with a “creative” programming style.

9. Suitability for safety-critical software development

When developing safety-critical software, the relevant standards require that the entire tool chain has to be qualified. The aim here is to prove that both the coverage analyzer and the other tools used within the entire toolchain work reliably. Manufacturers of professional code coverage tools support software projects with qualification kits and advice on tool qualification. In this context, attention should also be paid to whether the selected coverage tool is already being used successfully in safety-critical projects.

10. Evaluation licenses, technical support and customer references

The suitability of a coverage tool for one’s own projects should be checked during a tool evaluation. During this period, you will already get an impression of the performance of the technical support. Is the support also available by telephone or only by e-mail? How competent are the support staff? What about the response times? How good and practical is the user manual? Does the manufacturer also offer training? Last but not least, it is also advisable to take a look at the manufacturer’s customer references. These can provide further information about the quality of the coverage analyzer and the vendor’s performance.

Conclusion

Code coverage is mandatory for safety-critical software development for good reason. But it is also a good method for anyone who wants to improve their software quality in general to measure and increase the coverage and the value of tests. When selecting a code coverage analyzer, care must be taken to ensure that the tool meets the requirements set. In addition, factors such as ease of use and professional support play an important role. Used correctly, a good test coverage tool helps to significantly improve quality, increase the motivation of developers and testers, and perform tests in a cost-saving manner.

Code Coverage at a Glance

Function Coverage
Function Coverage measures whether all functions of the program were called. The Function Coverage is the “weakest” of the usual test coverage levels.

Statement Coverage
Statement coverage measures how high the percentage of tested statements is compared to all statements.

Decision Coverage / Branch Coverage
At this coverage level, each decision must be tested at least once as true and once as false. For normal if statements, this corresponds to branch coverage, where each branch must have been executed.

Condition Coverage
Condition coverage considers compund decisions in detail. For decisions that consist of multiple atomic conditions composed via Boolean operators, each of these conditions must be tested individually as “true” and as “false”.

Multicondition Coverage and Modified Condition/Decision Coverage (MC/DC)
For multicondition coverage, all possible true-false combinations must be checked for composite decisions. In the case of multiple conditions within a decision, this requires a mostly impracticably high number of test cases. In practice and in standards, Modified Condition/Decision Coverage (MC/DC) is therefore relevant, where the number of test cases is reduced, and the informative value of the test coverage remains sufficiently high.

Autor:

Klaus Lambertz *is Chief Executive Officer / Managing Director, Co-founder and shareholder at Verifysoft Technology GmbH. Prior to co-founding Verifysoft Technology GmbH in 2003, Klaus Lambertz had sales and management positions with different software testing solution providers in France and Germany (Parasoft, Testlight). From 1993-1999 he was responsible for exports to the German and Central European Markets for two of France´s most important printing plants (Partenaires-Livres, Maury Imprimeur). After working as a bank clerk in Cologne from 1981-1986, he graduated in studies of Economics, Marketing and Foreign Trade in Cologne (Germany) and Paris (France).*