

Testwell CTC++: Code Coverage Analysis for safety-critical Embedded Systems

by Professor Dr. Daniel Fischer (University of Applied Sciences Offenburg, Germany)

Software for embedded systems is often used in safety-critical systems. In this area malfunctions could lead to accidents or damages of high magnitude and even to loss of lives. Therefore, security standards such as the DO178-C (aviation), the ISO 26262 (automotive) or the EN 50128 (railway) demands harsh proof of code coverage. In dependency to the criticality, a suitable level of code coverage has to be applied.

Function Coverage is calculated by counting the number of all called functions and divide it by the number of all functions existing within the embedded software. The benefit of these tests is rather small because the control flow inside the functions is ignored completely.

Statement Coverage counts the statements that have been executed by tests. On this level already, dead code can be spotted or statements that have no test case yet can be found.

Branch Coverage is calculated on the basis of all primitive branches without pointing them out explicitly.

MC/DC (Modified Condition Decision Coverage) is calculated by considering all atomic conditions of a compound condition. For each atomic condition it has to be proven by a pair of test cases, that the final decision is affected by this atomic condition while the other atomic conditions are unchanged. This coverage level is mandatory for security critical software in aeronautic and automotive industries.

CTC++ Coverage Report - Functions Summary

[Directory Summary](#) | [Files Summary](#) | [Functions Summary](#) | [Execution Profile](#)

To directories: [First](#) | [Previous](#) | [Next](#) | [Last](#) | [Index](#) | [No Index](#)

Directory: .



TER: 86 % (24/ 28)

File: [./calc.c](#)

Instrumentation mode: function-decision-multicondition

TER: 82 % (14/ 17)

To files: [Previous](#) | [Next](#)

TER % - covered/ all	Calls	Line	Function
82 % - 14/ 17 	9	4	is_prime()
82 % - 14/ 17 			calc.c

File: [./io.c](#)

Instrumentation mode: function-decision-multicondition

TER: 80 % (4/ 5)

To files: [Previous](#) | [Next](#)

Testwell CTC++ shows code coverage for all functions

For code coverage analysis, the source code of the embedded software is instrumented and preferably executed on the host as well as on the target platform. For some embedded systems several limitations has to be considered such like less available RAM and ROM memory. This fact does not allow an extensive instrumentation of the code.

To perform coverage analysis, obviously there has to be a greater memory usage as for the non-instrumented source code. Instrumentation leads to greater consumption of RAM and ROM. This can be especially for small embedded systems a real challenge. A partial instrumentation with repeated test cycles could be one solution. To reduce RAM usage, there is also the possibility of decreasing the size of a single counter from 32 bit to 16 bit or even down to 8 bit. If it is only important that the code is covered and not how often, then it is possible to replace the 32-Bit counters by single bits. This is called Bit Coverage and can downsize the need for additional RAM by a factor of nearly 32. This technology is supported by Testwell CTC++ Test Coverage Analyser of Verifysoft Technology.

Further information: http://www.verifysoft.com/en_ctcpp.html

CTC++ Coverage Report - Execution Profile #1/3

[Directory Summary](#) | [Files Summary](#) | [Functions Summary](#) | [Execution Profile](#)

To files: [First](#) | [Previous](#) | [Next](#) | [Last](#) | [Index](#) | [No Index](#)

File: ./calc.c

Instrumentation mode: function-decision-multicondition

TER: 82 % (14/ 17)

Start/ End/

True False - [Line](#) Source

```

1  /* File calc.c ----- */
2  #include "calc.h"
3  /* Tell if the argument is a prime (ret 1) or not (ret 0) */
Top
9  0  4  int is_prime(unsigned val)
5  {
6      unsigned divisor;
7
2  7  8      if (val == 1 || val == 2 || val == 3)
1  8  8  T || _ || _
0  -  8  F || T || _
1  8  8  F || F || T
      7  8  F || F || F
2  9  9      return 1;
5  2  10     if (val % 2 == 0)
5  11  11         return 0;
58  2  12     for (divisor = 3; divisor < val / 2; divisor += 2)
13     {
0  58 - 14         if (val % divisor == 0)
0  -  15             return 0;
16     }
2  17  17     return 1;
18 }

```

*****TER 82% (14/17) of SOURCE FILE calc.c**

[Directory Summary](#) | [Files Summary](#) | [Functions Summary](#) | [Execution Profile](#)

To files: [First](#) | [Previous](#) | [Next](#) | [Last](#) | [Top](#) | [Index](#) | [No Index](#)

The read part of the code is not yet covered. A test where the first condition is false and the second is true need to be added in order to achieve full code coverage.