Qt Quality Assurance

WHITEPAPER

Code Coverage for Safety-Critical Programs

Metrics & Considerations

This white paper summarizes the different implications and considerations surrounding code coverage for safety-critical programs, along with code coverage requirements mandated by four major standards governing safety-critical software in various industry contexts.

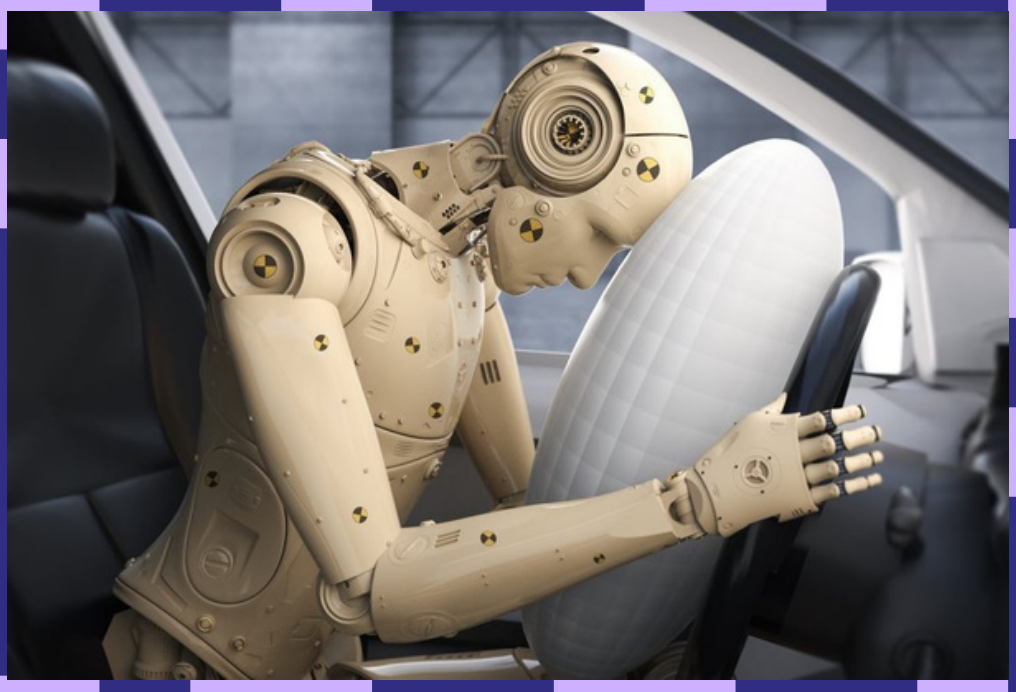


Table of contents

1. Safety-Critical Systems as Software

A system is "safety-critical" if a failure in its operation could result in human fatality (or severe injury) or significant damage to property or the environment. Such systems are becoming increasingly computer-based. (Take, for example, the RATP's ongoing development of fully autonomous subway lines in the Paris railway network.) In this digital transformation, standards in the field of safety engineering have emerged, which set requirements on the software development of software-based safety-critical systems.

Under the IEC 61508 standard, which governs functional safety of electrical/electronic/programmable electronic safety-related systems, the probability of a dangerous failure is such that, less than one failure — one human life lost — is probable every 114,115 years of continuous system operation at the top Safety Integrity Level (SIL).

Apart from their macroscopic view of human, property, and environmental safety, these standards' primary goal is to ensure the software quality and fitness at the source code level. In other words, the low (or zero) defect rate requirement extends to software operation. A method to achieve this is through quality assurance testing.

2. Code Coverage Requirements for Safety-Critical Programs

Each safety standard mandates a specific list of software testing requirements, one of them being code coverage of the program. Three questions arise.

1. What is code coverage?
2. How is code coverage quantified and measured?
3. Why is code coverage a requirement in safety-critical systems?

Code coverage is an analysis method that measures the percentages of source code functions, statements, and conditions executed by one or more tests. This code coverage data is measured and analyzed using a tool that instruments the program code, a pre-compiler step that inserts instructions into the code to trace the executions. When we run a suite of tests run against the instrumented binary, coverage data becomes available.

Code coverage analysis plays a crucial role in ensuring the fitness of safety-critical programs, and their more extensive systems. This kind of analysis informs development teams which (potentially critical) areas of the program have been left untouched by a priori testing.

Using a code coverage analysis tool enables the developer to identify dead code and eliminate it, detect bugs and resolve them, refactor existing code to improve efficiency, eliminate redundant tests, and more.

In regulated industries, code coverage plays a vital role in reducing the chance of critical defects occurring in production. By helping to ensure software quality, code coverage helps to ensure software confidence.

To attain certification, you must achieve standard-specific levels of code coverage. Each standard is unique in its requirements. We'll turn now to coverage metric requirements within the context of four major standards. These are:

- ISO 26262 Road Vehicles – Functional Safety
- IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems
- DO-178 C Software Considerations in Airborne Systems
- DO 50128 Software for Railway Control and Protection Systems

In the following sections, we provide a sample `C++` program, together with sample invocations, to describe the different coverage metrics against which test coverage can be measured. The metrics are function coverage, line coverage, statement (block) coverage, decision (or branch) coverage, Modified Condition/Decision Coverage (MC/DC), and Multiple Condition Coverage (MCC). Our sample program detects whether the test case input is a real number. It demonstrates all coverage metrics, including how some are weaker than others for specific tests.

3. Coverage Metrics

3.1 Function Coverage

3.1.1 Definition

The function coverage of a program counts how many functions were called (and how often). Here, the count includes member functions (or methods) in object-oriented programming languages like `C++`.

Consider our sample program, with an example function call:

```
1  #include <string>
2
3  bool isANumber( const std::string &s )
4  {
5      bool first = true;
6      for ( int i = 0; i < s.length(); i++ )
7      {
8          const char c = s[i];
9          if ( first )
10         {
11             first = false ;
12             if ( c == '-' )
13                 continue;
14         }
15         if (
16                 ! (( c >= '0' && c <= '9' ) || c == '.' )
17            )
18             return false;
19     }
20     return true;
21 }
```

```
1  int main()
2  {
3          isANumber("123")
4  }
```

Note that this metric reports only that a function was called; it does not report the execution of the body of the function.

3.1.2 Relevance for Safety Standards

ISO 26262*	IEC 61508	DO-178C	EN 50128
A = +	1 = ++	-	-
B = +	2 = ++	-	-
C = ++	3 = ++	-	-
D = ++	4 = ++	-	-

Table 1
Key: + denotes recommended; ++ denotes highly recommended. A-D, 1-4 are the respective standard's Safety Integrity Levels (SILs).

*architectural level

Function coverage is generally useful as an initial assessment of a project's coverage, but more robust metrics are required for in-depth analyses. Indeed, within the safety standards discussed in this paper, function coverage is required always with stricter metrics.

3.2 Line Coverage

3.2.1 Definition

Line coverage is the number of executed source code lines divided by the total number of source code lines. Depending on the code coverage toolchain in use, only lines that contain executable statements may be considered, not those with pure declarations. Other tools count pure declarations as executable code, for example: `int x = 0`

3.2.2 Formatting Dependency

This metric is unstable because it depends strongly on a program's code formatting. Consider our original program above, with a sample invocation:

```
1
2 int main()
3 {
4         isANumber("123")
5 }
```

This results in 80.00% line coverage. If you would reformat line 16 like this:

```
1  #include <string>
2
3  bool isANumber( const std::string &s )
4  {
5      bool first = true;
6      for ( int i = 0; i < s.length(); i++ )
7      {
8          const char c = s[i];
9          if ( first )
10         {
11             first = false ;
12             if ( c == '-' )
13                 continue;
14         }
15         if (
16                 ! (( c >= '0' && c <= '9' )
17                     || c == '.' )
18             )
19             return false;
20     }
21     return true;
22 }
```

The same invocation would result in 81.82% line coverage — for a decision which is only partially executed.

Intermediate line coverage is meaningless in quality assessments, and therefore only complete, 100% coverage is considered. One can imagine writing an entire function on one line or breaking the statements into multiple lines to change the coverage percentages systematically, without any increase in the *quality* of the tests.

3.2.3 Disguised Control Flow

Consider our original code listing before we demonstrated the formatting dependency. Let's modify the code to move the subsequent `return` statement in line 16, as follows:

```
1  #include <string>
2
3  bool isANumber( const std::string &s )
4  {
5      bool first = true;
6      for ( int i = 0; i < s.length(); i++ )
7      {
8          const char c = s[i];
9          if ( first )
10         {
11             first = false ;
12             if ( c == '-' )
13                 continue;
14         }
15         if (
16                 ! (( c >= '0' && c <= '9' ) || c == '.' ) )
       return false;
17     }
18     return true;
19 }
```

With the following invocation:

```
1
2  int main()
3  {
4          isANumber("123")
5  }
```

Our coverage increases to 88.89%. Note that line 16 is displayed as covered. But our test has not touched the `return false` statement; for decisions, or branching of code, line coverage does not detect the missing test.

3.2.4 Relevance for Safety Standards

None, due to the limitations listed above. Statement coverage, addressed in section 3.3 remediates the formatting dependency described above. Decision coverage, described in section 3.4, addresses the issue of branching seen in the example.

3.3 Statement Coverage

3.3.1 Definition

Statement coverage tracks the executed program statements. Statement coverage is calculated by dividing the number of executed statements by the total number of statements. Depending on the code coverage toolchain in use, the metric may be reported using simple or compound statements, the latter also known as blocks or statement blocks.

A block groups a sequence of simple statements. The compiler treats such blocks as a single statement. C++, for example, uses curly braces `{}` for grouping. Achieving 100% statement coverage gives 100% statement *block* coverage, and vice versa.

Statement coverage remediates the formatting dependency seen earlier in Line Coverage. In block coverage, complete statement (block) coverage subsumes complete line coverage. The disadvantage of statement (block) coverage is its weakness to simple-if statements. Simple-if structures have no `else` clause. Thus, complete statement coverage can be achieved for simple-if structures, regardless of the decision's truth outcome.

3.3.2 Relevance for Safety Standards

ISO 26262*	IEC 61508	DO-178C	EN 50128
A = ++	1 = +	A = ++	0 = +
B = ++	2 = ++	B = ++	1 = ++
C = +	3 = ++	C = ++	2 = ++
D = +	4 = ++	D = N/A	3 = ++
-	-	E = N/A	4 = ++

Table 2
Key: + denotes recommended; ++ denotes highly recommended. A-D, 1-4, A-E, 0-4 are the respective standard's Safety Integrity Levels (SILs).

Statement coverage is mandatory under the ISO 26262 standard for lower SILs, which do not highly recommend more stringent coverage levels. (For example, for ASIL D, Modified Condition/Decision Coverage (MC/DC) is required, which subsumes 100% statement coverage.) The DO-178C standard requires no statement coverage (or any higher-order metric) for levels D or E, where there are minor failure conditions or no effect on the system, respectively.

3.4 Decision (Branch) Coverage

3.4.1 Definition

The decision (or branch) coverage is the number of executed statement blocks and decisions divided by the total number of statements and decisions. Here, each decision counts twice: once for the true case and once for the false case.

We can achieve 100% decision coverage in our program with a minimum of two invocations:

```
1
2 int main()
3 {
4         isANumber("-123")
5         isANumber("ABC")
6 }
```

Note that this will automatically result in 100% statement, line, and function coverage.

Reaching 100% decision coverage ensures that all decision outcomes have been met, a shortcoming inherent in statement coverage. That said, this metric does not consider branches with Boolean expressions resulting from the logical operators, e.g., `&&` , `||` .

Taking a look at line 16:

```
16                  ! (( c >= '0' && c <= '9' ) || c == '.' )
```

A truth table reveals the untested conditional expressions:

Cond.	Truth val.
c >= 0	T (✓) F (X)
c <= 9	T (✓) F (✓)
c = '.'	T (X) F (✓)

Thus, Boolean expressions leading to the decision are not considered. This weakness is addressed in the condition metrics discussed in the next sections.

3.4.2 Relevance for Safety Standards

ISO 26262*	IEC 61508	DO-178C	EN 50128
A = +	1 = +	A = ++	0 = N/A
B = ++	2 = +	B = ++	1 = +
C = ++	3 = ++	C = N/A	2 = +
D = ++	4 = ++	D = N/A	3 = ++
-	-	E = N/A	4 = ++

Table 3
Key: + denotes recommended; ++ denotes highly recommended. A-D, 1-4, A-E, 0-4 are the respective standard's Safety Integrity Levels (SILs).

The DO-178C standard requires that decision coverage be met "with independence," i.e., code verification undertaken by validators who did not author the code under analysis.

3.5 Modified Condition/Decision Coverage (MC/DC) Coverage

3.5.1 Definition

In Modified Condition/Decision Coverage (MC/DC), each condition in a decision must be evaluated twice: once for a true outcome and once for a false outcome, while all truth values of all other conditions in the decision remain fixed. It is required that each Boolean outcome independently affects the decision outcome.

100% MC/DC coverage is achieved with a minimum of 4 tests:

```
1 int main()
2 {
3         isANumber("-123")
4         isANumber("1.23")
5         isANumber("ABC")
6         isANumber("%123")
7 }
```

Our characteristic truth table is then:

c >= '0'	c <= '9'	c == ' . '	Decision	Evaluation
FALSE	-	TRUE	FALSE	Uniquely evaluated by invocation: 1.23
TRUE	FALSE	FALSE	TRUE	Uniquely evaluated by invocation: ABC
FALSE	-	FALSE	TRUE	Uniquely evaluated by invocation: %123
TRUE	TRUE	-	FALSE	Evaluated by invocations: –1.23, 1.23

Compared to decision coverage, which requires that every control structure in the code has taken all possible decisions (or branches), MC/DC further requires that every condition in a decision takes every possible outcome, *and* each condition in a decision is shown to independently affect the decision's outcome.

Note the empty spaces in our decision table above. This is due to the short-circuit evaluations inherent in the `C` and `C++` languages.

But is our decision table complete? No. We need an additional test to cover the following combination:

c >= '0'	c <= '9'	c == ' . '	Decision
TRUE	FALSE	TRUE	FALSE

For this case, we turn to MCC, Multiple Condition Coverage, discussed in the next section.

3.5.2 Relevance for Safety Standards

ISO 26262*	IEC 61508	DO-178C	EN 50128
A = +	1 = +	A = ++	0 = N/A
B = +	2 = +	B = N/A	1 = +
C = ++	3 = +	C = N/A	2 = +
D = ++	4 = ++	D = N/A	3 = ++
-	-	E = N/A	4 = ++

Table 4
Key: + denotes recommended; ++ denotes highly recommended. A-D, 1-4, A-E, 0-4 are the respective standard's Safety Integrity Levels (SILs).

In the EN 50128 standard, MC/DC (or MCC) is recommended for SIL levels 1,2 and highly recommended for SIL levels 3,4.

3.6 Multiple Condition Coverage (MCC) Coverage

3.6.1 Definition

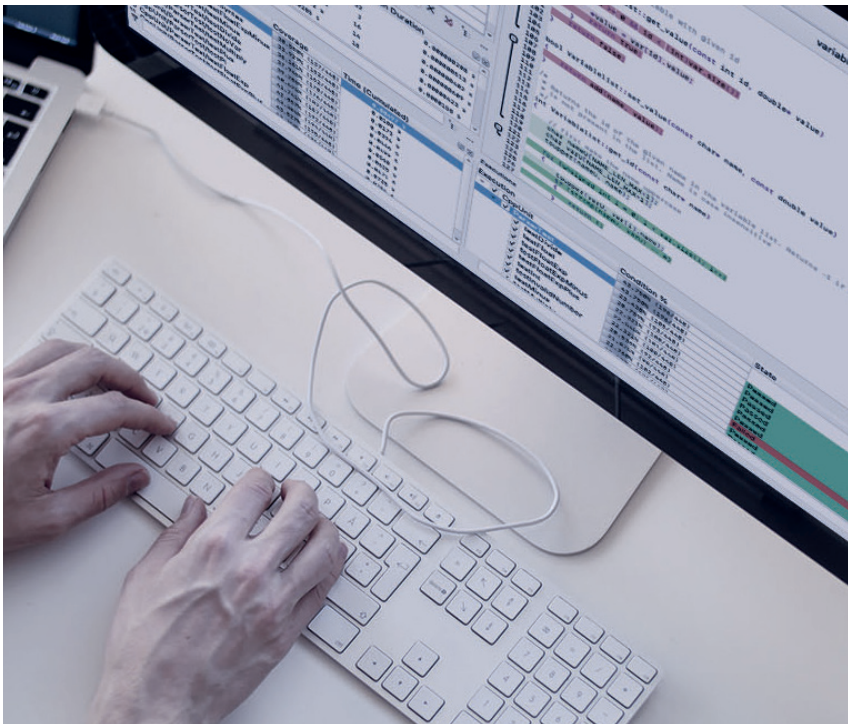
In Multiple Condition Coverage (MCC), every combination of condition outcomes within a decision occurs at least once to reach full coverage. The coverage is measured by taking the number of executed statement blocks and condition combinations divided by their total number in the program. With MCC, a complete decision table would be needed for full coverage. To determine the required tests in the decision table, substitute the number of conditions for N in 2^N.

Our above invocations for complete MC/DC coverage results in 93.750% MCC coverage. In our program, it is not technically possible to write a test which completes the decision table (owing to the single variable — c — which is included in every condition.)

3.6.2 Relevance for Safety Standards

Research has been conducted comparing the error-detection probability between MC/DC and MCC, in the context of testing efficiency and overhead trade-offs. We encourage readers to review this research as it applies to their own programs.

Of the four standards in this paper, the DO-178C and EN 50128 standards recommend MCC (or MC/DC) in their requirements. Generally, the MC/DC metric requires N + 1 tests, where N is again the number of conditions. Required tests in the MCC metric can 'explode' exponentially with large numbers of conditions. The MC/DC metric was created to compromise between plain condition/decision coverage and MCC.



4. Conclusion

The four standards presented in this paper are unique in their coverage metric requirements, but all share the common thread of minimizing system failures to prevent human fatalities. Software development of safety-critical systems requires **sophisticated code coverage tools** to permit a “test smart” vs. “test more” methodology not only to achieve safety certification but to deliver products within increasingly constrained frameworks. Automating the code coverage, where possible, is key to reducing human errors, which are the base cause of software defects.

Two things hold true about safety-critical software. First, their systems cannot be made safer once they are already in use. Second, due largely to continuing technological advancements in computing, these systems will play an increasingly ubiquitous part in human life. Therefore, prioritizing quality assurance as a means to ensuring confidence and fitness of the software for use is paramount.