Common Misconceptions in ECSS Cat-A Flight SW Qualification

ESA SW Product Assurance Conference

Andoni Arregi

2025-09-25, ESTEC

Content

1. Motivation

2. Misconceptions

3. New Challenges with AI

4. Conclusions

Thanks

This work is the result of a long collaboration with ESA. We want to explicitly thank:

- Andreas Jung
- Cristina Almaraz
- Isabelle Conway

Motivation

Software Qualification is an Art

- We have **requirements** and **methods** but then there is a **culture** and a **tradition** of how to apply them.
- There are many **myths** about ECSS Cat-A software qualification and its costs:
  - Because of MC/DC
  - Because of *what needs to be done with the object code*
- Over the years we have seen many **errors** and **misconceptions.**

Software Qualification is an Art

- We have **requirements** and **methods** but then there is a **culture** and a **tradition** of how to apply them.
- There are many **myths** about ECSS Cat-A software qualification and its costs:
  - Because of MC/DC
  - Because of *what needs to be done with the object code*
- Over the years we have seen many **errors** and **misconceptions.**

Good News

- We believe we can help to **debunk** many of them!

Misconceptions

MC/DC is Very Expensive

Relevant example: MC/DC necessary

```
bool func(bool a, bool b, bool c) {
    bool result;

    if (a && b || c)
        result = switch_on();
    else
        result = switch_off();
    return result;
}
```

MC/DC is Very Expensive

Relevant example: MC/DC necessary

```
bool func(bool a, bool b, bool c) {
    bool result;

    if (a && b || c)
        result = switch_on();
    else
        result = switch_off();
    return result;
}
```

MC/DC Implementation is Prohibitively Expensive

- **License cost** of proprietary tools
- **Effort** to increase the number of tests

MC/DC is Very Expensive

Relevant example: MC/DC necessary

```
bool func(bool a, bool b, bool c) {
    bool result;

    if (a && b || c)
    result = switch_on();
    else
    result = switch_off();
    return result;
}
```

MC/DC is Very Expensive

Relevant example: MC/DC necessary

```
bool func(bool a, bool b, bool c) {
    bool result;

    if (a && b || c)
    result = switch_on();
    else
    result = switch_off();
    return result;
}
```

Open-source tooling reduces the costs

- Available **Open-Source** tools eliminate the license cost
- Integration in CI pipelines ensures the tests are added continuously
- On average only one additional test for every complex decision

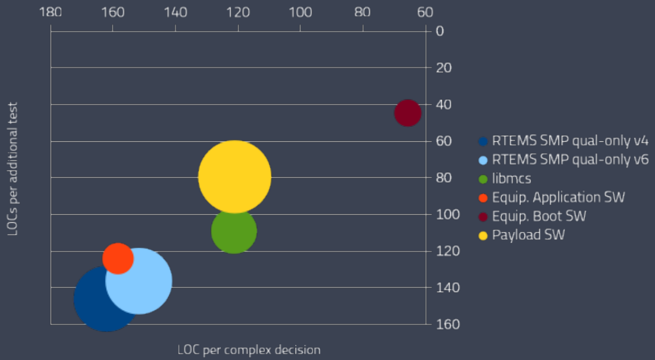
MC/DC is Very Expensive

Available open-source tools:

- GCC `gcov` is capable of MC/DC since GCC 14
- The `gcovr` tool also creates HTML reports for MC/DC
- GTD's `mcdc-checker` also enables older GCCs for MC/DC
- GTD's `mcdc-checker` helps to assess the extra effort to achieve MC/DC
- Clang/LLVM is also capable of MC/DC

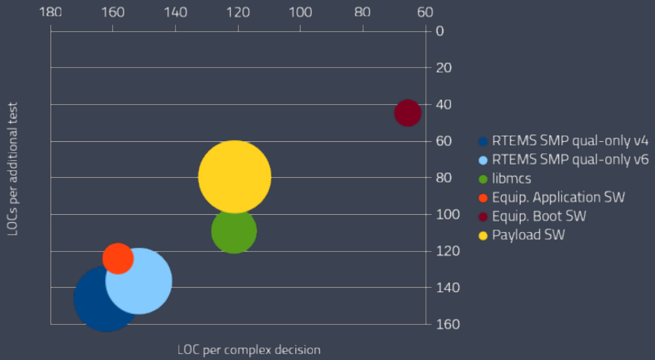
MC/DC is Very Expensive

Flight SW is not always so dense in decisions:



MC/DC is Very Expensive

Flight SW is not always so dense in decisions:

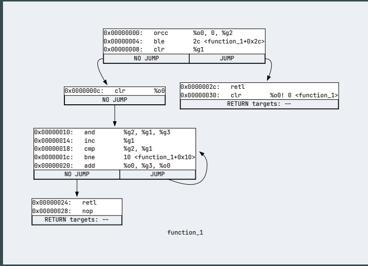


The whole qualified RTEMS OS with over 20 kLoC would need about 160 additional tests.

Object Code Coverage is Costly and Complex

Relevant example: Object Code Coverage necessary

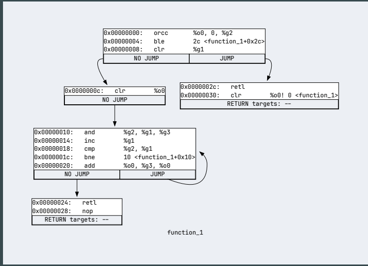
```
int function_1 (int n) {
    int total = 0;
    for (int i = 0 ; i < n ; i++) {
        total += i & n;
    }
    return total;
}
```



Object Code Coverage is Costly and Complex

Relevant example: Object Code Coverage necessary

```
int function_1 (int n) {
    int total = 0;
    for (int i = 0 ; i < n ; i++) {
        total += i & n;
    }
    return total;
}
```



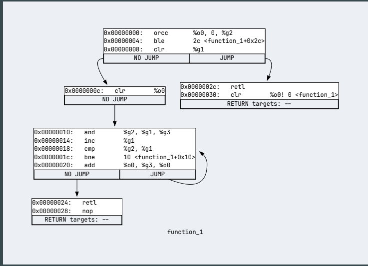
Object code coverage implementation is costly and complex

- **License cost** of proprietary tools
- **Complex** test set-up
- High additional test **effort**

Object Code Coverage is Costly and Complex

Relevant example: Object Code Coverage necessary

```c
int function_1 (int n) {
    int total = 0;
    for (int i = 0 ; i < n ; i++) {
        total += i & n;
    }
    return total;
}
```

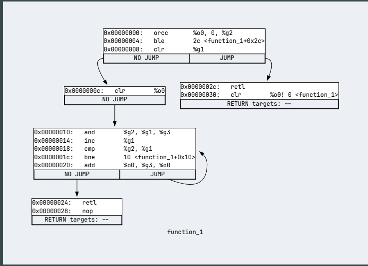


[1] The percentage refers to Basic Blocks in object code.

Object Code Coverage is Costly and Complex

Relevant example: Object Code Coverage necessary

```c
int function_1 (int n) {
    int total = 0;
    for (int i = 0 ; i < n ; i++) {
        total += i & n;
    }
    return total;
}
```



Assessment on empirical data shows otherwise

- **Open-source** tools can be used to cover most needs
- Detailed assessment shows gaps as low as 0.48%[1] on Cat-B SW (LibmCS)

[1] The percentage refers to Basic Blocks in object code.

Coverage on Object Code Coverage Better than on Source

Relevant example: Source code based testing necessary

```
uint16_t filter_sensor(uint16_t raw, uint16_t prev, uint8_t alpha)
{
    int32_t d = (int32_t)raw - (int32_t)prev;     // arithmetic: difference
    int32_t c = (d * alpha) >> 8;                 // arithmetic + bitwise shift
    uint16_t out = prev + c;                      // filtered output
    return out;
}
```

Coverage on Object Code Coverage Better than on Source

Relevant example: Source code based testing necessary

```
uint16_t filter_sensor(uint16_t raw, uint16_t prev, uint8_t alpha)
{
    int32_t d = (int32_t)raw - (int32_t)prev;     // arithmetic: difference
    int32_t c = (d * alpha) >> 8;                 // arithmetic + bitwise shift
    uint16_t out = prev + c;                      // filtered output
    return out;
}
```

Object code coverage can replace source code coverage

- If object code coverage is complete, every instruction has been exercised.
- Then, source code level unit testing will have no added value.

Coverage on Object Code Coverage Better than on Source

Relevant example: Source code based testing necessary

```
uint16_t filter_sensor(uint16_t raw, uint16_t prev, uint8_t alpha)
{
   int32_t d = (int32_t)raw - (int32_t)prev;     // arithmetic: difference
   int32_t c = (d * alpha) >> 8;                 // arithmetic + bitwise shift
   uint16_t out = prev + c;                      // filtered output
   return out;
}
```

Coverage on Object Code Coverage Better than on Source

Relevant example: Source code based testing necessary

```
uint16_t filter_sensor(uint16_t raw, uint16_t prev, uint8_t alpha)
{
    int32_t d = (int32_t)raw - (int32_t)prev;       // arithmetic: difference
    int32_t c = (d * alpha) >> 8;                   // arithmetic + bitwise shift
    uint16_t out = prev + c;                        // filtered output
    return out;
}
```

Both are complementary and necessary

- DO-178 CAST-17 clarifies that this assumption is wrong.
- Object and source code coverage are complementary.
- MC/DC and logic/arithmetic error detection potential is lost if only object code coverage is assessed.

Source Code Coverage Analysis Must Use Optimized Cross-Compilations

- The mantra says: *Test what you fly, fly what you test*.
- This includes the optimization flag `-O2` for unit test execution.
- `gcov` does not work well with optimized compilations thus, `gcov` is problematic to use.

Source code coverage *assessment* is done on *source code*

- Compiling with optimization and for coverage is nonsense:

  → Coverage instrumentation impedes optimization and optimization renders coverage instrumentation useless.
- First *assess* structural coverage of a test set:

  → Compile with `-O0` and for coverage, then execute the tests
- Later execute unit tests to check pass/fail criteria:

  → Compile with `-O2` without coverage then execute the tests again

Validation Tests Are More Valuable than Unit Tests

Unit tests have limited value compared to higher-level testing

- *Only validation tests test the real integrated flight software.*
- *If I already validated my software with validation tests, why do I need to still add unit tests just for the sake of coverage?*

Validation Tests Are More Valuable than Unit Tests

Unit tests give early and detailed defect information

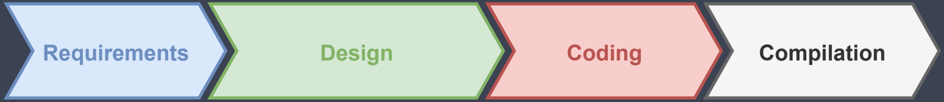
- Unit tests enable early and detailed defect detection. This costs more on validation tests.
- Validation tests, when used to assess coverage, produce a lot of *incidental coverage*.
  - Yields very positive coverage values
  - The covered code has been exercised but its behavior not verified.

New Challenges with AI

AI Produced Autocoding Adds New Challenges

- FSW engineering is a bit like the Peanut Butter Jelly Sandwich game
- This is even more valid for Cat-A flight software
- The computer needs exact instructions
- Natural language is inherently inexact and ambiguous

AI Produced Autocoding Adds New Challenges



- SW engineering consists in pressing out the ambiguity
- We get the exact instructions for the computer as output
- The verification process ensures every step is done correctly

Risk of AI Autocoding

- AI Autocoding cannot be used to jump over the complete design process.
- What you win in coding speed you loose in additional verification.

AI Produced Autocoding Adds New Challenges

- AI autocoding is BSH[a] generation *per se*
- Generated code is at best only one valid instance of the very large set of programs that comply with an ambiguous prompt.
- Very high pressure on verification to understand what the generated software does.
- Category A process is all about understanding exactly what we fly.

[a] BSH: Bullshit, to be understood in its Frankfurtian sense; *On Bullshit*, Harry Frankfurt, 1986

Conclusions

Conclusions

Good News

- Not necessarily so difficult nor expensive:

  $\longrightarrow$ open source tools, often less work than assumed
- Many of the points also *healthy* for lower criticality software
- All explained aspects are valid across different target processors, operating systems, cross-compiler toolchains, and programming languages.

Challenges

- More $\rightarrow$ Faster $\rightarrow$ Cheaper $\rightarrow$ Crash:

  $\longrightarrow$ Newspace awareness of software product assurance
- AI cannot replace software engineering processes.