The Qualification of Software Tools in Safety-Related Development

Roberto Bagnara
Prof., University of Parma / CTO, BUGSENG
Member, MISRA C Working Group
Member, ISO JTC1/SC22/WG14 — C std. committee

bugseng
http://bugseng.com

IWES 2020, February 9th, 2021

Outline I

Acknowledgments

Several slides are courtesy of Marcel Beemster, Solid Sands

Do You Want to Reason in Assembly?

```
f:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -20(%rbp)
    movq    $0, -8(%rbp)
    movl    $0, -12(%rbp)
    jmp     .L2
.L3:
    movl    -12(%rbp), %eax
    andl    -20(%rbp), %eax
    movl    %eax, %eax
    addq    %rax, -8(%rbp)
    addl    $1, -12(%rbp)
.L2:
    movl    -12(%rbp), %eax
    cmpl    -20(%rbp), %eax
    jb      .L3
    movq    -8(%rbp), %rax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Or in C?

```
#include <stdint.h>

uint64_t f(uint32_t n) {
  uint64_t total = 0;
  for (uint32_t i = 0; i < n; ++i) {
    total += i & n;
  }
  return total;
}
```

Programming Critical, Resource-Constrained Embedded Systems

C usage is pushed by very strong economic reasons

Unrestricted C has also very serious problems: non-definite behaviors

Ada, too, has non-definite behaviors

Other more defined high-level languages are not portable, flexible or efficient enough

Only two sensible options remain:

1. Stick to MISRA C/C++, compile C/C++ to assembly, and reason about programs at the source code level
2. Reason about programs at the assembly code level

Tools Are Badly Needed

Manually checking for MISRA C/C++ compliance is unpractical

Manually translating C/C++ to assembly is unpractical

Tools are needed, for these and for many other activities related to the development of embedded systems

To what extent can the tools be trusted?

Tool Qualification

The development of safety-critical software is regulated by standards such as:

- **CENELEC EN 50128** for railway
- **RTCA DO-178C** for airborne software
- **ECSS-Q-ST-80C** for European space applications
- **IEC 61508** for industry in general
- **IEC 62304** for medical devices
- **ISO 26262** for automotive

Tool Qualification (cont'd)

Due to the complexity of software, development and verification activity, de facto, have to rely on the use of tools

Malfunction of the tools may compromise the integrity of, or fail to detect defects in, the application software

In order to mitigate this risk, the standards prescribe integrity requirement on tools: this is usually called tool qualification

Tool Qualification (cont'd)

CENELEC EN 50128:2011

When tools are being used as a replacement for manual operations, the evidence of the integrity of tools outputs can be adduced by the same process steps as if the output was done in manual operation. These process steps might be replaced by alternative methods if an argumentation on the integrity of tools output is given and the integrity level of the software is not decreased by the replacement.

ISO 26262:2018

[The objective of the] qualification of the software tool [is] to create evidence that the software tool is suitable to be used to support the activities or tasks required by the ISO 26262 series of standards (i.e. the user can rely on the correct functioning of a software tool for those activities or tasks required by the ISO 26262 series of standards).

Tool Qualification (cont'd)

In the different standards, to a varying degree, tools are categorized depending on:

- potential impact of tool failure
- likeliness such failure is detected

Depending on the tool categorization, standards:

- put requirements on tool development
- put requirements on tool documentation
- put requirements on user skills: all software team members including tool users
- put requirements on tool qualification methods

Tool Qualification (cont'd)

An important distinction is between

- tools that can introduce defects in the application software, e.g., a C/C++ compiler
- tools that can fail to detect defects in the application software, e.g., a MISRA C compliance checker

Another crucial aspect is the scope of use of the tool!

- if the assembly code generated by the C compiler is manually verified, the qualification requirements on the compiler can be softened or eliminated
- if the MISRA C compliance checker is used to justify the elimination of testing activities its qualification requirements are increased

Tool Qualification (cont'd)

Finally, it must be taken into account that tools qualification can only be performed in the specific context of their actual use

The tool vendor can (and, in some cases, must) supply material that simplifies/enables the tool user to qualify the tool in the specific use context

However, the final responsibility of the tool choice and qualification lies with the tool user

As a consequence, all bragging about "certified tools" amounts to pure and simple marketing hoax

Tool Qualification with ISO 26262

ISO 26262:2018, Part 8, Section 11: “Confidence in the use of software tools”

Describes the process of tool qualification for a specific use case
The qualification process comprises:

1. Planning of usage
2. Evaluation
3. Qualification methods
4. Validation and mitigating actions
5. Documentation and review

1. Planning of usage

Determine:

a) tool identification
b) tool configuration
c) tool use case
d) tool execution environment
e) maximum ASIL
f) qualification methods

2. Tool evaluation

The Tool Confidence Level (TCL) classes represent the required degrees of confidence in a software tool so that it can be used in a given tool chain, for a given use case, on a given operational environment

The classes are:

- TCL1 low confidence
- TCL2 medium confidence
- TCL3 high confidence

2. Tool evaluation (cont'd)

The Tool error Detection (TD) classes are meant to capture the confidence in deployed measures that are able to prevent and/or detect malfunctions of a software tool and the consequent production of erroneous output, for a given use case, on a given operational environment

The classes are:

- **TD1** there is a high degree of confidence that malfunctions and the consequent erroneous outputs will be prevented or detected
- **TD2** there is a medium degree of confidence that malfunctions and the consequent erroneous outputs will be prevented or detected
- **TD3** there is a low or unknown level of confidence that malfunctions and the consequent erroneous outputs will be prevented or detected

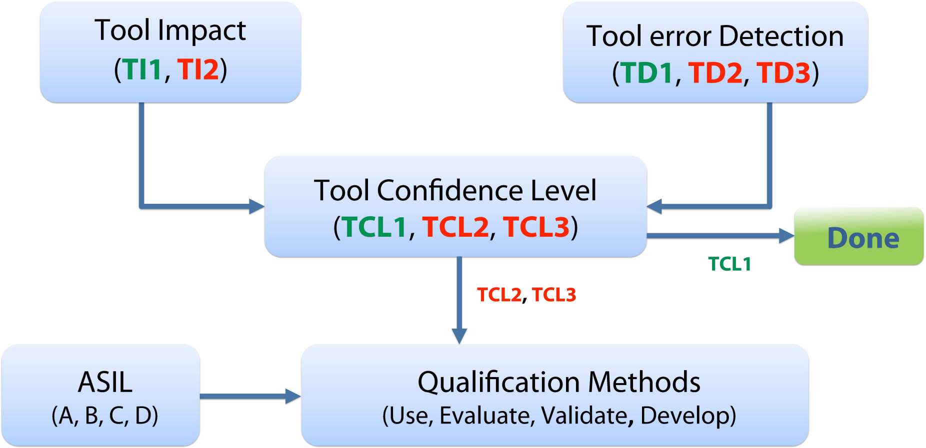
2. Tool evaluation (cont'd)

The Tool Impact (TI) classes are meant to capture the possibility that a malfunction of a software tool can introduce or fail to detect an error in a safety-related item or element under development

The classes are:

- **TI1** the tool can neither introduce nor fail to detect errors
- **TI2** the tool can introduce errors and/or fail to detect errors

2. Tool evaluation (cont'd)



3. Qualification methods

Columns of ISO 26262:2018, Part 8, Section 11, Table 4, TCL3, ASIL C–D:

(+) Increased confidence from use

(+) Evaluation of the tool development process

(++) Validation of the tool

(++) Development in accordance with a safety standard

4. Validation of the tool

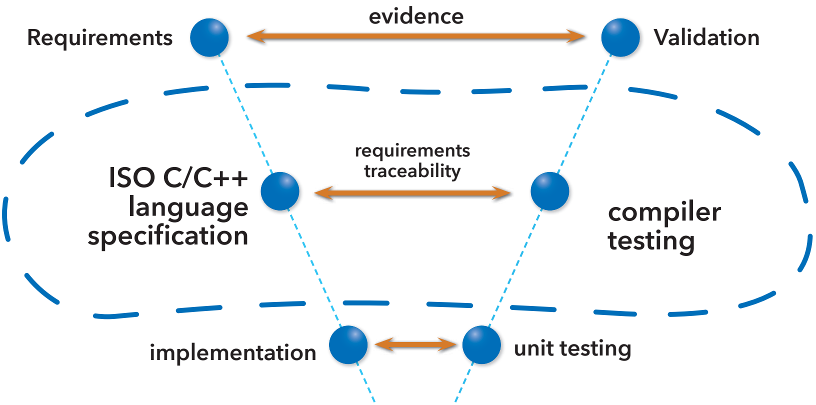
Compliance with the functional specification by testing given the specific use-case

Must take place in the user tool operational environment

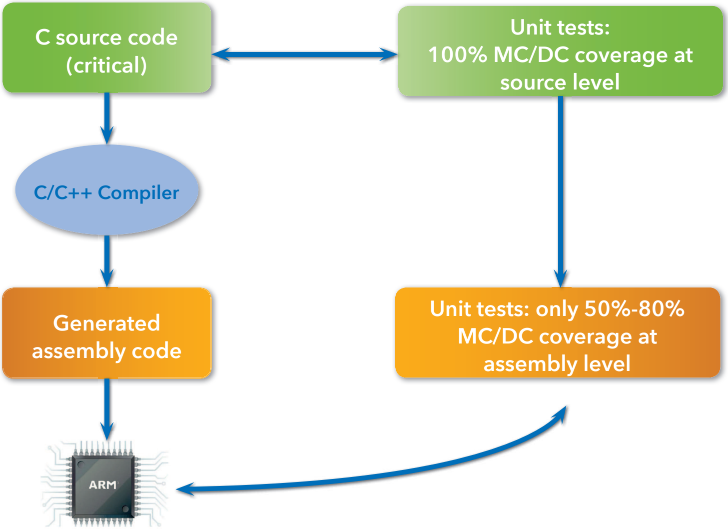
With the precise configuration used in production

Must define mitigations for tool malfunctions

Compiler Validation



Compiler Validation: Unit Testing Is Not Enough



Coverage at Source Level

```c
#include <stdint.h>

uint64_t f(uint32_t n) {
  uint64_t total = 0;
  for (uint32_t i = 0; i < n; ++i) {
    total += i & n;
  }
  return total;
}
```

Complete coverage at source level with one test: `f(1)`

Coverage at Assembly Level: -O0

```
f:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    movl     %edi, -20(%rbp)
    movq     $0, -8(%rbp)
    movl     $0, -12(%rbp)
    jmp      .L2
.L3:
    movl     -12(%rbp), %eax
    andl     -20(%rbp), %eax
    movl     %eax, %eax
    addq     %rax, -8(%rbp)
    addl     $1, -12(%rbp)
.L2:
    movl     -12(%rbp), %eax
    cmpl     -20(%rbp), %eax
    jb       .L3
    movq     -8(%rbp), %rax
    popq     %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Without optimization complete coverage also at assembly level with one test: `f(1)`

Coverage at Assembly Level: -Ofast

```
f:
.LFB0:
   .cfi_startproc
   testl    %edi, %edi
   je       .L7
   leal     -1(%rdi), %eax
   cmpl     $3, %eax
   jbe      .L8
   movl     %edi, %edx
   movdqa   .LC0(%rip), %xmm2
   movd     %edi, %xmm7
   xorl     %eax, %eax
   shrl     $2, %edx
   movdqa   .LC1(%rip), %xmm5
   pxor     %xmm1, %xmm1
   pxor     %xmm3, %xmm3
   pshufd   $0, %xmm7, %xmm6
.L4:
   movdqa   %xmm2, %xmm0
   addl     $1, %eax
   paddd    %xmm5, %xmm2
   pand     %xmm6, %xmm0
   movdqa   %xmm0, %xmm4
   punpckhdq %xmm3, %xmm0
   punpckldq %xmm3, %xmm4
   paddq    %xmm4, %xmm0
   paddq    %xmm0, %xmm1
   cmpl     %edx, %eax
   jb       .L4
   movdqa   %xmm1, %xmm0
   movl     %edi, %edx
   psrldq   $8, %xmm0
   andl     $-4, %edx
   paddq    %xmm0, %xmm1
   movq     %xmm1, %rax
   cmpl     %edx, %edi
   je       .L11
.L3:
   movl     %edi, %ecx
   andl     %edx, %ecx
   addq     %rcx, %rax
   leal     1(%rdx), %ecx
   cmpl     %edi, %ecx
   jnb      .L1
   andl     %edi, %ecx
   addq     %rcx, %rax
   leal     2(%rdx), %ecx
   cmpl     %ecx, %edi
   jbe      .L1
   andl     %edi, %ecx
   addl     $3, %edx
   addq     %rcx, %rax
   cmpl     %edx, %edi
   jbe      .L1
   andl     %edx, %edi
   addq     %rdi, %rax
   ret
.L7:
   xorl     %eax, %eax
.L1:
   ret
.L11:
   ret
.L8:
   xorl     %edx, %edx
   xorl     %eax, %eax
   jmp      .L3
   .cfi_endproc
```

With optimization, `f(1)` coverage is incomplete at assembly level

Uses of Static Analyzers in ISO 26262 (Part 6)

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL				ECLAIR
		A	B	C	D	
1a	Enforcement of low complexity	++	++	++	++	✓
1b	Use of language subsets	++	++	++	++	✓
1c	Enforcement of strong typing	++	++	++	++	✓
1d	Use of defensive implementation techniques	+	+	++	++	✓
1e	Use of well-trusted design principles	+	+	++	++	✓
1f	Use of unambiguous graphical representation	+	++	++	++	–
1g	Use of style guides	+	++	++	++	✓
1h	Use of naming conventions	++	++	++	++	✓
1i	Concurrency aspects	+	+	+	+	–

Uses of Static Analyzers in ISO 26262 (Part 6, cont'd)

Table 3 — Principles for software architectural design

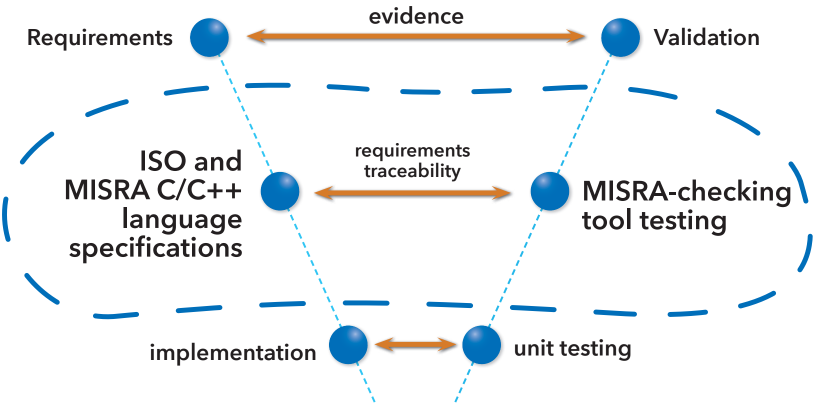
Methods		ASIL A	ASIL B	ASIL C	ASIL D	ECLAIR
1a	Appropriate hierarchical structure of software components	++	++	++	++	✓
1b	Restricted size and complexity of software components	++	++	++	++	✓
1c	Restricted size of interfaces	+	+	+	++	✓
1d	Strong cohesion within each software component	+	++	++	++	✓
1e	Loose coupling between software components	+	++	++	++	✓
1f	Appropriate scheduling properties	++	++	++	++	–
1g	Restricted use of interrupts	+	+	+	++	–
1h	Appropriate spatial isolation of the software components	+	+	+	++	–
1i	Appropriate management of shared resources	++	++	++	++	✓

Uses of Static Analyzers in ISO 26262 (Part 6, cont'd)

Table 6 — Design principles for software unit design and implementation

Methods		ASIL A	ASIL B	ASIL C	ASIL D	ECLAIR
1a	One entry and one exit point in subprograms and functions	++	++	++	++	✓
1b	No dynamic objects or variables, or else online test during their creation	+	++	++	++	✓
1c	Initialization of variables	++	++	++	++	✓
1d	No multiple use of variable names	++	++	++	++	✓
1e	Avoid global variables or else justify their usage	+	+	++	++	✓
1f	Limited use of pointers	+	++	++	++	✓
1g	No implicit type conversions	+	++	++	++	✓
1h	No hidden data flow or control flow	+	++	++	++	✓
1i	No unconditional jumps	++	++	++	++	✓
1j	No recursions	+	+	++	++	✓

MISRA Static Analyzer Validation



MISRA/HIS Checker Configuration: C is a Large Family of Languages

In C99, there are 112 implementation-defined behaviors

As each i.d.b. can be defined in 2 or more ways, there are more than $2^{112} \approx 5 \times 10^{33}$ possible languages

Actually, choosing integer and floating-types in $\{8, 16, 32, 64\}$ brings us to more than 10^{36} possible languages (dialects of C)

Alexander's star: 7.24×10^{34} different positions

C is a Large Family of Languages (cont'd)

Generally speaking, a given compiler can implement, via options, several such dialects of C

For an extreme case, GCC/x86_64 implements, via options, millions/billions of dialects of C

As a consequences, the tool must adapt to the particular dialect implemented by that compiler with that set of options (possibly for each translation unit)

Further consequence: changing even one compilation option may have important consequences, including analyzing the wrong code!

5. Documentation and review

1. Software tool criteria evaluation report
2. Software tool qualification report, typically resulting in updates to the tool safety manual
3. Confirmation review by an independent party

Qualification Kits

I they are well done, they can decrease the effort of tool qualification by one to two orders of magnitude

They must contain:

- Documentation and documentation templates: if a tool safety manual is not there it is a bad sign
- Validated test suites: this requires thousands of tests for a MISRA checker, and tens of thousands of tests for a compiler
- Possibility for users to add their own test cases
- Test automation machinery supplied in source form for inspection
- Possibility of repeating each test completely independently from the qualification kit

So-called Tool Certificates

A tool cannot be certified, it can be qualified

Marketing people will write just anything to pretend the contrary, e.g.:

- “Usable in [...] ISO 26262 up to ASIL D, TCL1 can be reached”
- “The tool is certified ISO 9001”

In the unfortunate case you end up in court, this kind of things will be demolished in 30 seconds by expert witnesses

Some certification agencies have responsibility for this malpractice

Always do examine the full reports that are an integral part of the so-called certificates

Conclusion

Tool qualification is an essential requirement for using tools in safety-related developments

We have covered the basic process for ISO 26262, which is not very different from the process described in other functional safety standards

It is a complex process if done in isolation, it is straightforward if done with the help of a good qualification kit

There are advantages besides checking the box:

- Sleeping better (not so with a so-called "tool certificate")
- Decouple application development from tool testing
- Reduced time-to-market

With a good qualification kit, or at least a good validation suite, it is not rocket science

The End


Questions?
roberto.bagnara@unipr.it
info@bugseng.com
bugseng