# Bringing Existing Code into MISRA Compliance

## Challenges and Solutions

Roberto Bagnara[1,3] Stefano Stabellini[2] Nicola Vetrini[3]
Abramo Bagnara[3] Simone Ballarin[3] Patricia M. Hill[3]
Federico Serafini[3,4]

[1]University of Parma
[2]AMD
[3]BUGSENG
[4]Ca' Foscari University of Venice


bugSeng

# MISRA Compliance of Existing Software

Greatest benefits from the MISRA coding standards can be obtained when they are enforced since the very beginning of the project

This is often not possible, for various well-known reasons

One that is becoming more and more common is the incorporation into safety critical systems of open-source software (OSS)

bugSeng

# Why Incorporating OSS in Critical Systems?

Access to innovative technologies and products

Avoidance of vendor lock-in

Leverage of larger ecosystems allowing for cost reduction and shorter development cycles

Lower layers in a typical software stack are particularly suited to this move:

- Trusted Firmware Arm, Google, ST, Renesas, NXP, Linaro, ...
- Xen Project Hypervisor AMD, Arm, AWS, EPAM, Linux Foundation, ...
- Zephyr RTOS Analog Devices, Google, Intel, Meta, NXP, Linux Foundation, ...
- ELISA Boeing, Red Hat, Arm, Bosch, Huawei, Linux Foundation, ...

bugSeng

# Challenges in the Safety-Qualification of OSS

Notable issues are, typically, with:

**Processes** Some processes are there, but do not comply with functional-safety standards (e.g., technical safety concept is skipped, formal inspections are not conducted)

**Artifacts** In most cases they are largely missing (e.g., traceable requirements and tests)

**Governance** Based on volunteer work, only part of the community may care about safety qualification: hence, liability left to end users

bugSeng

# Challenges in the Safety-Qualification of OSS (cont'd)

**High configurability** Want to safety-qualify only a small subset: how is that defined and how to deal with common code?

**Pace of development** Many OSS projects move too fast: maintaining all safety artifacts up-to-date is hard

**Virtuoso programming** Truly talented programmers take pride in writing clever, hard-to-understand code, but safety-critical code has to be boring!

bugSeng

# Approaches to the Safety-Qualification of OSS

Some possibilities, which are not mutually exclusive:

**Retrofitting safety** Create/adapt requirements; make them traceable to the implementation and tests; adopt and enforce a suitable coding standard

**Fork** Not suitable if code is not fit for purpose as is, but might be inevitable in the final phases of certification

**Refrain from safety qualification** E.g., qualify only a small component that monitors the correct operation of the rest

bugSeng

# MISRA Compliance and Functional Safety

Safety-critical software has to be:

- traceable to documented requirements
- verifiable and verified by means of peer review, static and dynamic analyses, and testing

Use of C and C++ is allowed provided that:

1. they are standardized
2. programs have a well-defined semantics: no undefined or unspecified behaviors
3. translators are qualified
4. programs refrain from using error-prone constructs
5. program units (e.g., functions) are of limited complexity

MISRA C and MISRA C++ address these concerns

bugSeng

# MISRA Compliance of Existing Code

Legacy code comes in two flavors: adopted code and simply existing code

Adopted code is fit-for-purpose as is and needs not to be read understood or modified: simplified MISRA compliance

Simply existing code will be adapted and modified: standard MISRA compliance

bugSeng

# MISRA Guidelines Tailoring

Let us take it for granted that existing code that is candidate to inclusion in safety-critical system is of very high quality

MISRA makes it very clear that code quality always comes first

Tailoring of the MISRA guidelines is essential:

- selection of the (non-mandatory) guidelines to comply with
- global deviations for required guidelines where justifiable
- partial deviations for required guidelines to adapt them to the project

bugSeng

# The Xen Hypervisor

Xen is a type-1 hypervisor with a microkernel design

Xen is a open source community project under Linux Foundation, license GPLv2

ARM and x86 fully supported, RISC-V in progress

Xen embedded features highlights:

- Real-time and cache isolation in software (cache coloring)
- Fast parallel VMs booting and static partitioning (dom0less)
- Virtio and Xen PV Drivers support with freedom from interference
- Cortex-R52 and R82 support, Xen on MPU systems

bugSeng

# MISRA compliance of the Xen Hypervisor

Xen runs independently at a higher privileged mode, separately from any operating system: everything else runs on top of Xen as a Virtual Machine

Xen is instrumental in achieving isolation/independence/freedom-from-interference among software components running on the same hardware platform

Being one of the most critical components in a system, it needs to be qualified for safety-related use

BUGSENG is working with the Xen community, under AMD funding, to achieve MISRA compliance of selected Xen configurations for AMD's x86-64 and ARM64 architectures

bugSeng

# Partial Deviation of MISRA Guidelines: E.g. Rule 10.1

Multiple rationales: unspecified and undefined behavior, as well as implementation-defined behavior and developer confusion

But the following are safe:

- value-preserving conversions of integer constants
- shifting non-negative integers to the right, if the shift count is non-negative and not too large
- shifting non-negative integers to the left, if the shift count is as above and if the result is still non-negative
- bitwise logical operations on non-negative integers, even if the operands are of signed type
- implicit conversion to Boolean for logical operator arguments
- two's complement behavior of bitwise ops on signed integers can be assumed to be known by all developers

bugSeng

# Encapsulate and Deviate

Clever bit trick that not all know:

```
align &= -align;  // Unary minus of unsigned:
                  // Rule 10.1 violation
```

This is now encapsulated into a macro:

```
/*
 * Given an unsigned integer argument, expands to
 * a mask where just the least significant nonzero bit
 * of the argument is set, or 0 if no bits are set.
 */
#define ISOLATE_LSB(x) ((x) & -(x))
```

Which is globally deviated as far as Rule 10.1 is concerned, then:

```
align = ISOLATE_LSB(align);
```

bugSeng

# Overly Restrictive Guidelines

Some MISRA guidelines are perceived as being too restrictive, such as Rule 16.3 (*An unconditional* $break$ *statement shall terminate every* $switch\text{-}clause$)

`return`, `goto` and `continue` are as good as `break` for this:

```
switch ( state )
{
  case IO_ABORT:
    goto inject_abt;
  case IO_HANDLED:
    /* ... */
    return;
  case IO_RETRY:
    /* ... */
    return;
  /* ... */
```

bugSeng

# Overly Restrictive Guidelines (cont'd)

In addition, functions declared with the `_Noreturn` function specifier can also be used to comply with the spirit of Rule 16.3:

```
switch ( kinfo->d->arch.vgic.version )
{
  /* ... */
  default:
    panic("Unsupported GIC version\n");
}
```

bugSeng

# Conditional Compilation via Kconfg

Several OSS projects (including Linux, Zephyr and Xen) use the Kconfig DSL for managing quite complex configurations
Preprocessor-based conditional compilation is explicitly not recommended:

```
switch(x) {
  case 1:
    f();
    break;
#if IS_ENABLED(CONFIG_FOO)
  case 2:
    foo_enabled();
    break;
#endif
  default:
  break;
}
```

bugSeng

# Conditional Compilation via Kconfg (cont'd)

Instead, this is what is recommended, because:

- it is easier to write and read, especially in presence of nesting
- syntax is always checked, also in excluded parts

```
switch(x) {
  case 1:
    f();
    break;
if(IS_ENABLED(CONFIG_FOO)) {
  case 2:
    foo_enabled();
    break;
}
  default:
  break;
}
```

bugSeng

# Reachability and Decidable Guidelines

MISRA view: excluded code should be removed by preprocessing

Linux view: nothing changes if excluded code is removed by later compiler translation phases

Note that decidable guidelines apply to all code that is present after preprocessing, including code guarded by `if(0)`

A compromise is possible provided that:

1. rules are deviated taking into account the following points
2. the compiler does indeed eliminate code that is guarded by `if(0)` unless jumping inside it is possible
3. no jumps inside code excluded by `if(0)` are possible from outside: true if a bunch of other MISRA guidelines are complied with!

bugSeng

# Lessons Learned

**Compliance vs deviation**: on existing high-quality code we can expect more deviations, but the MISRA potential for suggesting improvements is still very high

**Interdependence**: MISRA guidelines are interdependent:

- the "obvious" correction of a violation for a guideline may violate another
- the safety guarantees are the result of the whole set of guidelines, so the impact of tailoring must be assessed globally

bugSeng

# Lessons Learned (cont'd)

**Tool configurability**: many of deviations that were formulated were only possible due to the flexibility and configurability of the ECLAIR static analyzer

**Continuous integration**: this is crucial to accommodate a diverse and distributed community as well as to lighten the burden of maintainers

**MISRA and OSS**: the open attitude in OSS project can initially clash with the requirements imposed by functional safety standards, but if the right message passes, high quality results can be attained

bugSeng

# Conclusion

Bringing an existing codebase into MISRA compliance is known to be a difficult, risky and time-consuming task

Sometimes this is a necessity, particularly in the case of OSS

In order to succeed a lot of experience is required, and also: training, tailoring, tooling

We have presented some of the issues faced and of the lessons learned during our work on the Xen Hypervisor Project (as well as others)

This presentation only concerned a small selection of issues and lessons: much more is in the paper!

bugSeng