embeddedworld
Exhibition&Conference

Bringing Existing Code into MISRA Compliance: Challenges and Solutions

Roberto Bagnara
University of Parma, Italy
Email: name.surname@unipr.it

Stefano Stabellini
AMD, USA
Email: name.surname@amd.com

Nicola Vetrini, Abramo Bagnara,
Simone Ballarin, Patricia M. Hill
BUGSENG, Italy
Email: name.surname@bugseng.com

Federico Serafini
BUGSENG, Italy
Email: name.surname@bugseng.com
Ca' Foscari University of Venice, Italy
Email: name.surname@unive.it

***Abstract*—Bringing an existing codebase into MISRA compliance is known to be a difficult, risky and time-consuming task. Yet, when a product needs a functional safety certification and rewriting the software is out of question, this is a necessity. Such an endeavor requires facing multiple tradeoffs and, consequently, lots of experience both on the codebase and on MISRA. The choices between deviating the guideline, and the (often, many) ways in which code may be changed and deviations may be formulated, are tough and with consequences that are not immediately evident. While, clearly, a project undertaking MISRA compliance at a late development stage is likely to rely on deviations more than other projects, one should take into account the interdependence among MISRA guidelines and that such deviations have to be rock-solid (as they will inevitably catch the assessors' attention). In this paper, we illustrate our experience and the several lessons learned while undertaking MISRA compliance work in several projects. This includes closed-source projects (which cannot be disclosed for confidentiality reasons) as well as open-source projects, most notably the Zephyr RTOS and the Xen hypervisor, both used in many embedded systems.**

I. Introduction

Open-source software (OSS) is quickly finding its way in safety-critical embedded systems. The reason why this is desirable are well known: access to innovative technologies and products emerged from (and, quite often, well established in) the OSS world; avoidance of vendor lock-in; leverage of larger ecosystems allowing for cost reduction and shorter development cycles. The lower layers in a typical software stack are particularly suited to this move toward open-source solutions: firmware, hypervisors and operating systems are complex, critical common building blocks upon which a public ranging from individual contributors to large corporations is willing to invest. Notable examples include:

Trusted Firmware A collection of projects providing a reference implementation of secure software for Armv8-A, Armv9-A and Armv8-M. That code, which is the preferred implementation of Arm specifications, forms the foundations of a *Trusted Execution Environment* on application processors, or the *Secure Processing Environment* of microcontrollers.[1] The project is backed up, among others, by Arm, Google, ST, Renesas, NXP, and Linaro.

Xen Project Hypervisor A flexible, scalable and secure hypervisor supporting multiple guest operating systems, including Linux, Windows, NetBSD, FreeBSD and Zephyr, and multiple Cloud platforms: CloudStack and OpenStack. Xen has a solid track record including millions of installations worldwide.[2] The project is backed up, among others, by AMD, Arm, AWS, EPAM, and the Linux Foundation.

Zephyr RTOS A small, scalable real-time operating system targeted at connected, resource-constrained and embedded devices. Zephyr is used in lots of existing products and supports more than 600 boards.[3] The project is backed up, among many others, by Analog Devices, Google, Intel, Meta, NXP, and the Linux Foundation.

This is just a selection of the open-source projects that are currently targeting certification for use in safety-related products. Among the many initiatives in this area, it is worth mentioning the ELISA project: named after the initials of *Enabling Linux In Safety Applications*, ELISA aims to make it easier for companies to build and certify Linux-based safety-critical applications.[4] The project is backed up, among many others, by Boeing, Red Hat, Arm, Bosch, Huawei, and the Linux Foundation.

A. Challenges on the Safety-qualification of Existing OSS

Despite its theoretical desirability, the qualification of existing OSS for use in safety-related development presents significant challenges involving different, though interconnected, aspects [1]:

Processes Open-source projects do not follow a development process that complies with functional-safety standards.

[1] https://www.trustedfirmware.org/
[2] https://xenproject.org/
[3] https://www.zephyrproject.org/
[4] https://elisa.tech/

That is not to say that they do not follow any process at all. To the contrary, many of them follow a reasonably formalized process involving the use coding standards and style guides [2], [3], systematic testing based on continuous integration systems, static analysis, dynamic analysis and constraints on test coverage, informal reviews. For instance, the SQLite project is known in the open-source community for its quality management and thorough testing standards.[5] Nonetheless, while the established development processes of many open-source projects satisfy the quality-related requirements of functional-safety standards, safety-related requirements remain largely uncovered: for example, a *technical safety concept* (as defined, e.g., in ISO 26262 [4, Clause 7]) is typically missing; as another example, formal inspections (as defined, e.g., in ISO 26262 [5, Clause 3.82]) are usually not conducted.

Artifacts Safety-related development requires the creation and maintenance of a significant amount of artifacts that, due to the incremental nature of open-source and to its reliance on volunteers, are often not present or maintained. Traceable requirements constitute the typical example: they are missing in most open-source projects.

Governance The governance model adopted by the majority of open-source projects clashes with the requirements for safety-certification. Being largely based on volunteer work, liability and responsibility for the proper use of OSS is, by necessity, left to end users. Moreover, safety-certification is often of interest to only a part of the community, with the result that the rest of the community might be unwilling to bear the extra burden and costs that safety-related development brings with itself.

These aspects compound in negative ways with the following:

High configurability Many OSS projects are highly configurable:[6] as a consequence, obvious economic considerations imply that safety-qualification may only be done for a tiny subset of the available configurations. This requires configuration mechanisms that are aligned with the objectives of functional-safety standards. For instance, as all code should be traceable to requirements, if only a subset of the project is subject to the safety qualification, then it must be proved that what ends up in the final deliverable is exactly what is expected to be there, and nothing more. Moreover, there is the risk of community fragmentation arising from the distinction between project parts that are safety-qualifiable and those that are not. And how to handle the code that is in the intersection?

Pace of development Many OSS projects evolve with such a high pace that updating the required safety artifacts may be infeasible.[7]

Virtuoso culture Traditionally, OSS projects have been an excellent training ground for virtuoso programming. While code for functional-safety should be boring and obviously correct to peer reviewers and assessors, highly-regarded (and truly talented!) individuals in OSS communities take pride in writing concise and possibly very efficient code, often at the expense of readability. This is often combined with a resistance to: writing documentation (because "code speaks by itself"); following procedures (because "speed of development is reduced"); complying to standards (because "we of course know better"). One is tempted to paraphrase the famous sentence coined by David Clark in 1992 [6] to partially capture an attitude still very popular in many open-source circles: "We reject: processes, coding standards, and metrics. We believe in: rough consensus and running code."

B. Possible Approaches to Safety-qualification of Existing OSS

Safety-qualification of existing open-source software is not an easy matter. What has to be accounted for carefully is that part of the savings due to not having to write the software from scratch will have to be invested into the safety-qualification effort. In the following sections we review, following [1], what the main alternatives are. It has to be noted that they are not mutually exclusive and that a combination of them is desirable and, often, the only way to go.

1) Retrofitting Safety: For existing OSS that is believed to be reasonably close to the spirit of the applicable functional safety standards, one possibility is to create the missing safety artifacts for qualification. This typically involves

a) the creation/adaptation of requirements and of tests for those requirements;
b) the establishment of traceability between the requirements, the implementation and the tests;
c) the definition of suitable coding standards (very often these are based on the MISRA standards [7], [8], [9]);
d) adherence to the chosen coding standards.

Points (a) and (d) are, by far, the most resource-demanding ones. However, while point (a) may only require limited changes to the main codebase (i.e., the implementation may need fixes to address the mistakes possibly uncovered by the new tests), point (d) may involve extensive code modifications: it is not uncommon for projects that never enforced, say, the MISRA coding standards, to start with a number of violations in the order of hundred of thousands or millions. Which involves the choice between propagating the changes upstream and forking the project. Forking may be the only viable option in case the project community has divergent views about the objectives and/or means of conducting the safety-qualification. Not that pushing the changes upstream does not have its share of issues: acceptance of the changes will sometimes be painful, especially if project contributors are not adequately trained. In order for this to scale up effectively, there needs to be

[5] See https://www.sqlite.org/qmplan.html and https://www.sqlite.org/testing.html.

[6] In many cases, their high configurability (e.g., for operating systems, the range of supported boards) is a key factor for their popularity and attractiveness.

[7] Again, fast-evolving projects are popular also for that very reason.

automation in place to repeat the coding standard compliance checks: projects that have an active community and to which changes are incorporated daily can only keep up if every code change is promptly and automatically checked.

2) Fork: While sometimes forking the project is unavoidable due to a mismatch between the project governance and the safety-qualification requirements, it is not something that can be decided lightly. Changes in the upstream project will have to be backported, whether they are interesting new features or bugfixes: in any case a decision will have to be made to either ignoring the change or to importing it with all the reworking that is required not to negatively impact functional safety. And in some cases, such as the ones of newly discovered vulnerabilities, the decision cannot be neglected. On the other hand, the cost of backporting the upstream changes will increase over time if the projects keep diverging. Summarizing, permanent forking might be advantageous only for projects that are considered very stable and fit for purpose for the foreseeable future. Yet, temporary forking might be required in the final phases of safety-qualification. This is due to the *Pareto principle*, which can be formulated as follows: "The final 20% of the safety-qualification work requires 80% of the total resources invested." In other words, due to the typical governance of OSS, while upstreaming the initial 80% of the changes might be done in a time-effective way, the final 20% of the changes have usually to be treated differently in order to honor the deadlines of safety-qualification; for these cases, a temporary fork might be appropriate to buffer the changes with the hope they will be accepted upstream in due course.

3) Refrain from Safety-qualification: Another possibility is not to safety qualify the OSS, or to safety-qualify only a part of it. In the first case, the correct operation of the OSS will have to be monitored by appropriate subsystems and suitable mitigations must be in place. In the second case, which is not disjoint from the first one, only part of the OSS will be safety-qualified. In both cases, a proof must be provided that there is no interference between the non safety-qualified components and the safety-qualified ones (e.g., the monitors): luckily there are tools that help automating that process almost completely so that the checks can be redone effortlessly at any update of the OSS [10].

C. Role and Scope of this Paper

In this paper, we illustrate our experience and the several lessons learned while undertaking MISRA compliance work in several projects. This includes closed-source projects (which cannot be disclosed for confidentiality reasons) as well as open-source projects, most notably the Zephyr RTOS and the Xen hypervisor, both used in many embedded systems.

The paper is structured as follows: Section II explains the role of the MISRA coding standards in functional safety; Section III discusses two different ways in which existing code can be considered for inclusion into a project seeking MISRA compliance and why "tailoring" of the MISRA guidelines is essential to deal with it; Section IV presents the *Xen Hypervisor* project; Section V illustrates some points worth of mention arising from the work we did on the MISRA compliance of Xen; Section VI discusses lessons learned and take-home messages; Section VII concludes the paper.

II. MISRA Compliance and Functional Safety

Functional-safety standards (such as ISO 26262 [11], CENELEC EN 50128 [12] and EN 50657 [13], IEC 61508 [14], IEC 62304 [15], IEC 60335 [16], and RTCA DO-178C [17]) concur on the fact that 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 programming languages are allowed provided the following points are taken into account:

1) they are standardized (e.g., by ISO or other authoritative standardization bodies);
2) programs have a well-defined semantics (i.e., no undefined or unspecified behaviors);
3) translators are qualified (to reduce the risk they would introduce defects in compiled code);
4) programs refrain from using error-prone or difficult-to-understand constructs, so as to maximize readability and understandability for the sake of more effective peer review;
5) program units (e.g., functions) are of limited complexity to ensure testability and, again, improve readability and understandability.

For the C and C++ languages, all these concerns are addressed by the MISRA coding standards [7], [8], [9]. MISRA C and MISRA C++ define subsets of the standardized versions of C and C++, respectively, addressing most of the points above [2], [3], [18]. Concerning the limitation of complexity, the MISRA standards recommend the use of metrics, even though they leave the choice of the metrics to be used to the individual organizations. Regarding the qualification of translators, the MISRA standards do their part in limiting the use of non-standard language extensions.

While the MISRA coding standards are unanimously considered the most authoritative ones for safety-related development in C and C++, their adoption on existing code can be challenging. For some OSS projects a partial adoption, possibly following the rationale-based classification proposed in [18], or the adoption of a less rigorous coding standard, such as BARR-C:2018 [19] might be the right choice. As explained in [20] adoption of BARR-C:2018 might be, for some OSS projects, a good intermediate step towards the adoption (upstream or downstream) of MISRA C as well as a way to fulfill the prescription of a defined coding style that, as in the case of metrics, MISRA recommends while leaving freedom of choice.

III. MISRA Compliance of Existing Code

It is well known that the greatest benefits from the adoption of the MISRA coding standards can be obtained when the coding standard is enforced since the very beginning of the

project. In fact, blindly imposing the MISRA guidelines upon an existing code base with a proven track record may be counterproductive if not done properly. However, one thing that is often misunderstood is that MISRA does not ask for blind adoption: to the contrary, the MISRA coding standards clearly state that code quality always comes first. MISRA compliance is one way to assess code quality, but neither is it exhaustive nor is it the only way: code can be of very high quality and yet have many violations. Indeed, a MISRA violation is just a call for attention: raising a deviation is, in many cases, the best course of action [21].

It is important to distinguish two different uses of so-called *legacy code* in a safety-related project:

1) Existing code that is considered fit-for-purpose in its present form, and thus (after having been properly assessed) needs not be read, understood, let alone modified by anyone; such code, as far as MISRA is concerned, can be considered *adopted code*. The MISRA coding standards have special, simplified prescriptions for the compliance of adopted code [21].
2) Existing code that is imported into the safety-related project as a useful starting point, but that requires further development and adaptation work.

While this paper is only concerned with the second case (that is, legacy code that cannot be considered *adopted code* under the MISRA definition), some considerations can be made that apply to both cases.

A. Tailoring the Guideline Selection and Individual Guidelines

Working effectively with legacy code requires distinguishing those aspects of the MISRA guidelines that relate to undefined or unspecified behavior from those related to possible developer confusion [18]. While the former aspects cannot be underestimated and have to be addressed one way or another, the latter aspects can, after due consideration, be discounted for legacy code: this can significantly reduce the MISRA compliance effort.

Consider MISRA C Rule 10.1 (*Operands shall not be of an inappropriate* essential type) as an example: this places restrictions on the types that operands can have for each of the many C operators. Violating some restrictions might cause undefined behavior: this is the case for negative shift counts in shift operations. Other violations concern extremely fishy things, such as involving Booleans in arithmetic operations. However, other violations concern possible developer confusion and/or implementation-defined behavior, such as using integers in a Boolean context (a pattern that is very common in legacy code) or performing bitwise operations on signed integers. As the number of Rule 10.1 violations in legacy code can be very high (on the order of tens of thousands for medium-sized embedded software projects), violations for code that is safe and well understood by the relevant community may be deviated globally in the tool configuration, in order to allow developers to focus the effort on code issues that more likely constitute software defects. For example:

- the value-preserving conversions of integer constants are safe;
- shifting non-negative integers to the right is safe if the shift count is non-negative and not too large;
- shifting non-negative integers to the left is safe if the shift count is non-negative and not too large, and if the result is still non-negative;
- bitwise logical operations on non-negative integers are safe even if the operands are of signed type;
- the implicit conversion to Boolean for logical operator arguments is safe;
- on architectures where signed integers are represented using two's complement (i.e., all the ones currently in use, to the point that this representation will be the only one supported by C23, the next standard for the C programming language), the behavior of bitwise *and*, *or*, *xor* and negation on signed integers can be assumed to be known by all developers.

The MISRA coding guidelines represent a compromise between competing goals: program safety and programmers' freedom of writing "clever" and compact code. Safety not only requires the "bad thing" to be prevented: the "bad thing" has to *clearly and unambiguously* be prevented, i.e., code reviewers should immediately see that the "bad thing" has been prevented. A consequence is that the guidelines need to be simple which, in turn, due to the "safety first" principle, entails that guidelines are often more restrictive than strictly necessary. When code is existing before the application of the MISRA guidelines, developers typically have already taken measures to prevent the most common mistakes from happening, possibly from a different angle with respect to the one taken by the MISRA guidelines. The presence of such measures can be a hindrance to the adoption of some guidelines but, at the same time, the value they provide to the project is indisputable. Some of the most widely used approaches concern a combination of build-time and run-time assertions, the encapsulation of tricky functionality in macros and inline functions, and the definition of a coding style that all the contributors of the project must follow. Such conventions are enforced during code reviews and, in some cases, can be supported by compiler warnings and other automatic checks. When a project has its own established measures to prevent the "bad thing", insisting on full adherence to the MISRA guidelines is counterproductive: there is always the risk of inadvertently introducing new defects in a way that escapes the usual testing activities. Therefore, it is often the case that project-defined measures need to be special-cased, assuming that the tool used to prove compliance supports them. Indeed, when dealing with existing code, the ability to "tailor" the MISRA guidelines is therefore a crucial ingredient to success. This is 100% in line with the MISRA spirit: guideline selection and justified deviation are part of the very concept of "MISRA compliance" [21]. For guideline selection, the MISRA categorization of *mandatory*, *required* and *advisory* guidelines is designed to adapt the coding standards to the particular

project. Note also how, for problematic language features the MISRA approach joins very strict advisory rules, disallowing all uses of the feature, to required or mandatory rules, which allow less problematic uses of such features. The rationale is as follows: if the advice about not using the feature altogether can be taken, a whole class of code issues can be avoided in a way that does not require further checks; otherwise, if a strong motivation exists for using the feature (e.g., due to the presence of infrastructure code that would be too expensive to rewrite in a fully MISRA compliant manner, or code that is imported periodically from other external sources), then the coding standard provides guidance on how to use it in a way that is, on average, safer.

B. Highly Configurable Projects

One of the common reasons why existing code might be attractive is its high configurability, as this increases the likelihood that it can be suitably configured for the project at hand. This is an important factor to be considered when bringing such projects towards MISRA compliance. For those, there will typically be modules and configurations for which MISRA compliance is not currently sought but, due to shared headers and source files, they cannot simply be ignored. This requires particular care in limiting the perimeter of the code that needs to comply to the guidelines and in the handling of the cases that are on the boundary (e.g., violations that manifest themselves in a source file meant to comply, but are due to non-compliant code in file that is not meant to comply).

C. The Special Case of Open Source Projects

One of the peculiarities of OSS software is that generally anyone is welcome to contribute to it, and the review process for incorporating changes is often public and arbitrarily thorough. One of the conditions that should be met in order to effectively take on a MISRA compliance process is to reach an agreement in the community (or at least the main maintainers) on the desirability of such process, for otherwise the proposed changes may be perceived as unmotivated and then rejected. Even when such generic consensus has been reached, each modification suggested by the violation of MISRA guidelines should be motivated in detail so that reviewers can appreciate the resulting code quality improvement. One of the problems is that the vast majority of OSS contributors are not in possession of the main MISRA publications. Note also that OSS communities comprise contributors with very diverse backgrounds and areas of expertise and very few of them have received suitable MISRA training.

IV. The Xen Hypervisor Project

Before going into the experience of the authors in their work on MISRA compliance of the Xen hypervisor, this section introduces the system and its role in the development of safety-critical systems.

A. The Xen Hypervisor Project

Xen, originating in 2003 from Ian Pratt's team at the University of Cambridge's Computer Laboratory, stands as the first open-source hypervisor. Over the years, Xen has gained widespread industry adoption: it is integrated into all major Linux distributions and has become the backbone of the largest cloud infrastructures starting from Amazon AWS.

Xen is a type-1 hypervisor with a microkernel design. It runs independently at a higher privileged mode, separately from any operating system. Everything else runs on top of Xen as a Virtual Machine (VM). In a traditional Xen architecture, the first VM to boot, known as *dom0*, typically uses Linux and holds privileged capabilities, such as creating and destroying other VMs. In 2018, the introduction of the *dom0less* feature marked a significant evolution of the project, allowing for fully functional Xen systems without the dom0 control domain. This setup involves statically pre-configuring the system before system-start and enabling Xen to create all VMs in parallel at boot time.

Xen was originally designed for the x86 architecture. In 2011, an effort started to port Xen to ARMv7 with virtualization extensions (32 bits). Today, Xen's compatibility extends beyond x86 and ARMv7 to ARMv8 (64 bits) and other architectures like RISC-V, PowerPC, ARMv7-R, and ARMv8-R. The expansion to ARM ushered Xen into the realm of embedded devices. Today, Xen is the reference open-source hypervisor for the embedded and automotive sectors, thanks to its microkernel design, the vibrant community, and rich feature set.

B. Xen Role in Safety-Critical Systems

Xen boasts a rich array of features, making it an excellent choice for embedded and automotive applications. In particular, Xen is instrumental in achieving *independence* and *freedom from interference* as defined in ISO 26262 [11]. For instance, Xen adeptly runs a real-time operating system (RTOS) for critical applications, alongside other, larger and non-critical VMs, such as Linux. Xen ensures that the RTOS' interrupt latency and execution time are low and deterministic, even when other VMs are under heavy load. One of its standout features, *cache coloring*, allows Xen to partition the last-level cache, effectively eliminating cache interference between VMs and enhancing isolation. This feature empowers Xen to consistently guarantee an interrupt latency of just 4 microseconds, regardless of the system's overall activity.

V. MISRA Compliance of Xen: Selected Highlights

As one of the most critical components in the system, Xen is an ideal candidate for the highest levels of safety certification. In 2023, AMD, together with the Xen Community, kicked off a series of activities to make Xen safety-certifiable according to the ISO 26262 automotive standard [11] and the IEC 61508 industrial standard [14], spanning AMD x86 and ARM architectures. Additionally, they have laid the foundation of the

testing infrastructure, based on GitLab, and started drafting safety requirements.

During the first nine months of the new endeavor, 80% of the relevant MISRA C rules have been integrated into the Xen coding style and, with the help of BUGSENG consultants, numerous MISRA C violations were resolved. BUGSENG also provided the initial MISRA C training to all maintainers and other Xen contributors. The *ECLAIR Software Verification Platform* has been integrated into the upstream Xen Continuous Integration (CI) loop to prevent new unjustified MISRA C violations from entering the codebase. The target of the project is to achieve compliance with respect to MISRA C:2012 Revision 1 with Amendment 2 [7], [22] following the prescriptions of MISRA Compliance:2020 [21].

In this section, we draw from this experience to illustrate how the principles outlined in the previous sections can be applied effectively to existing codebases. The code snippets reported in the sequel are mostly taken from the Xen project main Git repository[8] and are possibly redacted for presentation purposes. Xen is based on C99 [23], the configurations to be qualified are for the AMD x86-64 and the ARM64 architectures. The translation toolchain is based on suitable versions of the GNU C compiler.

A. Pointer Type Safety

MISRA C Rule 7.4 requires that string literals are assigned only to variables of type pointer to `const`-qualified `char` where, in MISRA C parlance, assignment covers also parameter passing to and value return from functions. The rule is there to prevent C99 Undefined Behavior (UB) 30 (*The program attempts to modify a string literal*) but, for this, the restriction to pointers to `char` is unnecessary: assignees that are pointers to `const`-qualified (non-`char`) type are equally good. This observation is sufficient to justify the deviation for code like

```
ret = dbgp_bulk_write(dbgp,
                      USB_DEBUG_DEVNUM,
                      dbgp->out.endpoint,
                      "\n", 1, &ctrl);
```

where the string literal `"\n"` corresponds to a formal parameter of type `const void*`. Modifying the signature of the function to take a `const char*` is undesirable, as this is only one of its use cases. As this is quite common in Xen, a tool configuration is used to express a project deviation whereby the rule is relaxed by allowing a string literal to be assigned to any `const`-qualified type. This measure, of course, relies on compliance with Rule 11.8 (*A cast shall not remove any* const *or* volatile *qualification from the type pointed to by a pointer*).

Rule 11.3 (*A cast shall not be performed between a pointer to object type and a pointer to a different object type*) is there to prevent C99 UB 22 (*Conversion between two pointer types produces a result that is incorrectly aligned*) and UB 34 (*An object has its stored value accessed other than by an lvalue of an allowable type*). As an exception, the rule allows converting pointer to objects to pointer to char, as the two UBs do not apply to such cases. An example of code that violates the rule can be found in Xen's x86 emulator, which is a delicate and carefully crafted piece of code that has been in use for a long time: it is the typical software module you do not want to modify unless forced to do so, as the risk of introducing new defects is quite high.

```
switch ( dst.bytes )
{
    case 1: *(uint8_t  *)src.reg
        = (uint8_t)dst.val; break;
    case 2: *(uint16_t *)src.reg
        = (uint16_t)dst.val; break;
    case 4: *src.reg
        = (uint32_t)dst.val; break;
        /* 64b reg: zero-extend */
    case 8: *src.reg
        = dst.val; break;
}
```

Here, the expressions `src.reg` and `dst.val` have type `unsigned long*` and `unsigned long`, respectively, where `unsigned long` is 64 bits and has the strictest alignment requirement among the integer types used in the example. While case 1 is compliant by the rule's exception (`uint8_t` is a char type), cases 2 and 4 violate the rule. A static analyzer, such as ECLAIR, that can reliably obtain from the compiler the alignment constraints of each type allows locally relaxing Rule 11.3 so as to discount all cases where alignment is correct and UB 22 does not apply. Exclusion of UB 34 requires checking that the types used to read and write the registers are always consistent.

B. Declarations and Definitions

The work on violations of MISRA C Rule 8.4 (*A compatible declaration shall be visible when an object or function with external linkage is defined*) led to some improvements that reflected positively on the overall code quality of Xen. First, it pointed out missing header file inclusions in source files defining several functions: this incurs the risk of definitions getting out of sync with the respective declarations. Secondly, this activity uncovered several variables and functions that had external linkage unnecessarily: these were all changed so as to have internal linkage, thereby complying with Rule 8.4. A third line of improvement led to the introduction of a pseudo-attribute macro `asmlinkage`, which was first introduced in the Linux kernel for objects and functions that are meant to be used only by asm modules. As the need to have external linkage but are not otherwise used by the surrounding C code, a suitable tool configuration allows globally deviating them.

C. Preprocessing Directives

Certain C codebases rely heavily on the C preprocessor's stringification and token pasting operators to implement rough (but very convenient) forms of generic programming. As such operators are associated to undefined and unspecified behaviors, MISRA C Rule 20.10 (*The # and ## preprocessor*

[8] https://gitlab.com/xen-project/xen

operators should not be used) advises not to use them. A project like Xen, which has numerous uses of these operators in a lot of infrastructure code, has compelling reasons to deviate this rule. This is completely in line with the spirit of MISRA C, which supplements advisory Rule 20.10 with required rules such as Rule 20.12 (*A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators*). Rule 20.12 norms the interaction of macro arguments and text replacement operators to mitigate the risk of developer confusion: macro replacement does not happen when an argument is used as an operand of the # or ## operators; therefore, if a macro argument is itself a macro its usage in the macro body as an ordinary argument is subject to further macro replacement, whereas its usage with the aforementioned operators does not. Thus, in macros that violate Rule 20.12 the same argument is expanded in two different ways, a fact that is prone to being overlooked even by experienced developers and can go unnoticed during reviews. This has the potential of silently introducing subtle bugs in the code that are hard to spot unless the preprocessed code is manually inspected. Even trickier is the fact that a code may be as intended in the current project state, but suddenly become defective if an argument to a macro that is itself not a macro is later redefined as such. Note that there are also a few cases where this is deliberate and well-understood, such as in assertion macros. For instance, this is the definition of the `ASSERT` macro in debug builds for Xen:

```
#define ASSERT(p) \
    do { if ( unlikely(!(p)) ) \
         assert_failed(#p); } while (0)
```

These macros can be deviated individually after code review.

D. Essential Type System

MISRA C mitigates the rather weak type system of C by means of an *essential type system* whereby C standard types and C expressions are given an *essential type* that encodes their intended use. Every syntactic context in which an expression may occur has an "expectation" about the essential type of the expression. Whenever such expectation is unfulfilled, some MISRA C rule is violated. For instance, the guards of conditional and iteration constructs expect an expression that is *essentially Boolean*: whenever this is not the case, MISRA C Rule 14.4 is violated.

Rules related to the essential type system are one of the most critical areas for MISRA compliance: when they are suddenly introduced into existing projects a very large number of violations are to be expected. Let us consider MISRA C Rule 10.1 (*Operands shall not be of an inappropriate* essential type): enforcing all the constraints imposed by the rule at once would be a roadblock to project development but, as we suggested in Section III, we can filter out provably safe constructs to allow developers with extensive domain expertise to sort out the remaining cases.

A very common source of violations of this rule is due to the freedom with which signed and unsigned quantities are used within existing codebases. One such case is the following:

```
align &= -align; // Violates Rule 10.1
```

The type of `align` is `unsigned int`, therefore it has *essentially unsigned* type, but it is used as the operand of the unary minus operator, which expects an *essentially signed* operand, whence the violation. As Xen is only meant to run on architectures using the two's complement representation for signed integers, there is no risk for safety, and the code is equivalent to

```
align = align & (~align + 1);
```

and it has the effect of clearing all bits of `align` except for the least significant bit set: there is no way for this operation to overflow or otherwise give unexpected results, therefore the rule can be safely deviated in this case. However, the best way forward is to encapsulate this operation using a macro, since this is a fairly common pattern throughout the project:

```
/*
 * 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))
```

Now the `ISOLATE_LSB` can be deviated, as far as Rule 10.1 is concerned, and the code above can be rewritten as

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

with a clear gain also in terms of readability. Another option, which would give additional safety due to type checking, would have been using a `static inline` instead of a macro: this was not possible in this case, due to the use of the construct in contexts where a constant integer expression is required, most notably the size specifier in (non-VLA) array declarations (see [23, Clause 6.7.5.2]).

E. Initialization

Failure to initialize objects in C is a serious mistake. Reading an uninitialized automatic variable is undefined behavior, for which MISRA C has Rule 9.1 (*The value of an object with automatic storage duration shall not be read before it has been set*). In other cases, the language guarantees default initialization to zero and the issue addressed by the MISRA guidelines is whether reliance on such default initialization was intentional or not. An example is given by Rule 9.3 (*Arrays shall not be partially initialized*). When an array is partially initialized some of its elements may inadvertently be left out. The rule has three exceptions for

1) `{0}` initializers, a common idiom requesting zero-initialization of all array objects and subobjects;
2) array initializers using designated initializers only;
3) string literal array initializers.

An example of violation in Xen was triggered by an array-typed field of a struct:

```
struct dmi_system_id {
  int (*callback)
    (const struct dmi_system_id *d);
  const char *ident;
  struct dmi_strmatch matches[4];
  void *driver_data;
};
```

In some cases, less than four initializers were given for the `matches` array, thereby violating the rule, for example in

```
{
  .ident = "Sun Microsystems Machine",
  .matches = {
      DMI_MATCH(DMI_SYS_VENDOR,
                "Sun Microsystems")
/*    <-------------------------->
      Violation of Rule 9.3, expanded to
      { DMI_SYS_VENDOR,
        "Sun Microsystems" }
*/
  },
},
```

While this was absolutely intentional, the Xen community agreed on the fact that there was a real danger of omitting initializers by mistake. It was thus decided to define helper macros using designated initializers and specifying, in the macro name, how many elements are initialized. For instance, the macro specifying that only the first element is initialized and the remaining three elements are intentionally zero-initialized is

```
#define DMI_MATCH1(m1) \
    .matches = { [0] = m1 }
```

so that the above fragment can be rewritten as

```
{
  .ident = "Sun Microsystems Machine",
  DMI_MATCH1(
      DMI_MATCH(DMI_SYS_VENDOR,
                "Sun Microsystems")
  ),
},
```

which is compliant to Rule 9.3 by its exception 2.

F. Overly Restrictive Guidelines

Some MISRA guidelines impose constraints on code structuring to facilitate reviewers in their assessment of code correctness. An example is Rule 16.3 (*An unconditional* break *statement shall terminate every* switch-clause). The rationale of this guideline is to prevent unintended fallthrough in non-empty clauses due to mistyping or subsequent additions of clauses. This is particularly important because the `switch` statement grammar in C is very liberal,[9] whereas MISRA C attempts restricting it to the discipline of Pascal's `case` statement. In many programming communities the restriction imposed by Rule 16.3 is perceived as too strict. Indeed there are other ways to avoid fallthrough. To start with, `return`, `goto` and `continue` are as good as `break` in this respect.

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

Indeed, `break` statements placed just after `return`, `goto` or `continue` would be unreachable, which would violate MISRA C Rule 2.1 (*A project shall not contain* unreachable code).

Another legitimate possibility is allowing non-empty switch statements to be terminated by calls to functions that have the `_Noreturn` function specifier, such as the `panic()` function below in the following example:

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

One can also manifest the intention of falling through with a `/* fallthrough */` comment or, even better, by defining a pseudo-keyword, if the compiler supports it as is the case for the GNU C compiler:[10]

```
#if (!defined(__clang__) \
     && (__GNUC__ >= 7))
#define fallthrough \
    __attribute__((__fallthrough__))
#else
#define fallthrough \
    do {} while (0)  /* fallthrough */
#endif
```

Once `fallthrough` has been defined this way it can be used as in

```
switch ( action )
{
    /* ... */
    case CPU_UP_PREPARE:
        INIT_PAGE_LIST_HEAD(list);
        fallthrough;
    case CPU_DOWN_FAILED:
}
```

[9] *Duff's device* is a famous example of what can be achieved thanks to the flexibility of the `switch` statement in C: see https://en.wikipedia.org/wiki/Duff%27s_device for details.

[10] See https://gcc.gnu.org/onlinedocs/gcc/Statement-Attributes.html for details.

G. Reachability and Decidable Guidelines

Projects that are highly configurable often need to exclude certain sections of code from being executed depending on the build configuration. A popular method that some OSS projects use to accomplish this is the definition of the build configuration in a domain-specific language (DSL) that defines a set of compile-time constants used to conditionally exclude sections of code.[11] A popular choice for the DSL is Kconfig, which is used by the likes of the Linux kernel,[12] the Zephyr RTOS,[13], and the Xen hypervisor.

Suppose a configuration option `CONFIG_FOO` exists: if `CONFIG_FOO=y` in the Kconfig file, then the macro `CONFIG_FOO` will expand to `1`, otherwise the macro will not be defined. In addition, a macro `IS_ENABLED(x)` is defined so that `IS_ENABLED(CONFIG_FOO)` evaluates, at compile-time, to `1`, when `CONFIG_FOO=y`, and to `0`, otherwise. This allows the conditional exclusion of certain code sections in a way that is easy to understand and fits nicely into the control structure of the program. Consider the following snippet:

```
if (IS_ENABLED(CONFIG_PCI_PRI)
    && reg & IDR0_PRI)
    smmu->features |= ARM_SMMU_FEAT_PRI;
```

If `CONFIG_PCI_PRI` is not enabled, then the compound assignment statement will not be executed. An alternative way to write this would be

```
#ifdef CONFIG_PCI_PRI
if ( reg && IDR0_PRI )
    smmu->features |= ARM_SMMU_FEAT_PRI;
#endif
```

This is not entirely equivalent because, in the latter snippet, the whole `if` statement is discarded during the preprocessing phase, while in the former snippet the `if` statement is still present after preprocessing in all configurations. The main disadvantage of the latter approach, and the reason why it is vehemently opposed by the mentioned OSS communities, is that it is harder to read and write, especially in the case of nested `#ifdef` sections.

We have thus a divergence between the MISRA view that excluded code should be removed by preprocessing and the view whereby nothing changes if excluded code is removed by later compiler translation phases. Note that the divergence is more serious than it might appear at first sight: decidable MISRA rules apply to all code that is present after the preprocessing phase, whether that code is reachable or not. As such, if the violation of a decidable rule is present in the first snippet inside the `if` statement, it must be reported by any MISRA C analyzer that claims to fully cover the rule. For undecidable rules the situation is completely different: for those, the analyzer can, and usually will take reachability into account so that code that is guarded by `if(0)` does not contribute to violations.[14]

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

Fig. 1. Reachable code despite `CONFIG_FOO` being disabled

Is it possible to reconcile the MISRA view with the one of projects using `IS_ENABLED`, also as far as decidable rules are concerned? Yes, provided that:

1) rules are deviated taking into account the following points;
2) a guarantee can be obtained that the compiler will indeed eliminate code that is guarded by `if(0)` unless jumping inside it is possible;
3) no jumps inside code excluded by `if(0)` is possible from outside.

Point 2 requires checking the compiler documentation and, possibly, confirming the indications therein with suitable compiler validation activities. Regarding point 3, consider the following example given in Figure 1: there, the second clause of the switch statement would be excluded from compliance whenever `CONFIG_FOO` is disabled, but the clause is still reachable because `x` can have the value `2`, and therefore control-flow can jump past the `IS_ENABLED` check, because the code is of course still present after the preprocessing phase. This can have a significant impact on functional safety but, luckily, complying with a few MISRA rules prevents this situation. Indeed, if the following rules are fully complied with, there is no way to jump in the middle of a block and skip the `IS_ENABLED` check (or any other condition):

- Required Rule 15.3 (*Any label referenced by a* goto *statement shall be declared in the same block, or in any block enclosing the* goto *statement*);
- Required Rule 15.6 (*The body of an* iteration-statement *or a* selection-statement *shall be a* compound-statement);
- Required Rule 16.2 (*A* switch label *shall only be used when the most closely-enclosing compound statement is the body of a* switch *statement*).

Incidentally, this example shows the importance, when defining a tailoring of the MISRA guidelines, of taking into account any interdependence among the guidelines and the overall

[11] https://www.kernel.org/doc/html/latest/process/coding-style.html#conditional-compilation

[12] https://docs.kernel.org/kbuild/kconfig-language.html

[13] https://docs.zephyrproject.org/latest/build/kconfig/index.html

[14] We refer the interested reader to [24] for a systematic study of undecidable MISRA rules.

effect of their deviation. This aspect is discussed further in Section VI-B

VI. Discussion

In this section, we discuss some of the key findings we (re)discovered in our work on the application of the MISRA guidelines on existing software, in particular our work on the Xen hypervisor. Some of the findings are particularly relevant to OSS, but many apply equally to other software development models where MISRA compliance is sought for existing software.

A. Compliance vs Deviation

While it is certain that the judicious application of the MISRA guidelines to existing projects requires a considerable number of deviations, it is also the source of many occasions for improving code quality. During the work on MISRA compliance for the Xen hypervisor, it has been often the case that guidelines' violations pointed to code that maintainers agreed could be significantly improved. In these cases, the violations helped identifying better ways of accomplishing the same task as well as code that has become stale over time (e.g., certain typedefs or macros that are being gradually phased out in favor of others). These changes need not be massive and intrusive, and the community is typically willing to accept them because of their value. In any case, it is always a good idea splitting change sets in small chunks that are easy to review and coherent among themselves (e.g., addressing violations of one rule at a time, or a small set of tightly-related rules at a time, in a single file, or in a set of related files).

B. Interdependence among the Guidelines

Interdependence among the guidelines is another important issue that must be taken into account during the tailoring process and when modifying the source code to address MISRA violations. Interdependence is relevant in two ways:

1) The most straightforward way to address a violation of a guideline may collide with restrictions imposed by other guidelines. In this regard, having a CI system in place is priceless: the submitter can thus be warned in advance of possible faux pas by committing to a local tree and automatically perform the analysis to ensure that no new violations of any guidelines are introduced. This is crucial to lighten the overall burden on the maintainers' side, by not having to think of these interactions (often not trivial and quite numerous) when reviewing a pull request. This aspect is elaborated further in Section VI-D.
2) The MISRA guidelines are meant to complement each other in providing a convincing safety argument. As a result, even if a guideline is tailored by disregarding some of its aspects with a sound justification, the guidelines considered as a whole may not give the same guarantees as before. An important consequence of this fact is that the impact of tailoring can only be assessed globally and not guideline by guideline.

C. The Importance of Tool Configurability

As we observed earlier, successful application of the MISRA guidelines to existing software requires extensive tailoring. This is fully supported by MISRA, thanks to guideline categorization and its deviation process. However, successful tailoring crucially depends on the configurability of the static analysis tool used to check compliance: tailorings that are not supported by the tool are simply unfeasible. For example, the MISRA guidelines involving `switch` statements do not depend on the type of the controlling expression. However, a project might wish to tailor MISRA C Rule 16.4 (*Every* switch *statement shall have a* default *label*) so that switches controlled by expressions of enumerated types are treated differently[15] but this can only be done effectively if the static analysis tool supports this differentiation.

Another example regards the possibility of considering code exclusion based on Kconfig's `IS_ENABLED` mechanism the same way as preprocessor-based conditional compilation: this is only possible if the static analysis tool has been designed for high configurability at the outset.

In general, while text-based deviations are essential for expressing specific deviations, i.e., deviations that apply to a particular instance at a particular locations, they are bad for anything else: they do not scale and are particularly contraindicated for existing software, where you want to avoid all unnecessary code churn. This, again, requires a high configurability of the static analysis tool: it has to support the specification of the exact conditions upon which a certain deviation applies and the association of its justification to those conditions. The specifiable conditions may be logical combinations of syntactic conditions, type conditions and even semantic conditions on the relevant and surrounding code.

D. The Role of Continuous Integration

During the last decade, the adoption of Continuous Integration (CI) systems has spread widely in the software industry. CI systems allow easily keeping the project evolution under control in a sustainable and scalable way, especially if the project has a large number of contributors. For large OSS projects, CI systems provide a first triage for the billions of code lines from possibly untrusted contributors and help ensure that the overall code quality is consistent at all times.

Initially, CI systems were intended solely for carrying out tests, but their flexibility makes them suitable for very generic tasks: each scriptable operation that is likely to be performed after a repository-specific event should be triggered by a CI pipeline in a fully automatic manner. For an effective adoption of coding standards such as the MISRA one, integrating a state-of-the-art static analyzer in a CI system is a crucial ingredient to success.

The repository history holds pretty valuable information, especially in OSS projects with many occasional contributors,

[15] E.g., the project might decide not to have a `default` label in enum-controlled switches to avoid preventing the useful warning provided by the GNU C compiler with the `-Wswitch` option.

who often rely on this information when approaching a task, as it gives insight into the conventions and constraints adopted by the project, and their motivations. This includes, of course, the conventions that are in place for MISRA compliance. All the new contributions should undergo an analysis by a static analysis tool prior to their evaluation by the maintainers of the involved subsystems, so that the tool findings can be addressed right away, sparing the need for extra review cycles. Only after receiving positive feedback from the analyzer, will the contribution be evaluated by the maintainers, making the work simpler and more accurate thanks to the automation. This process is often referred to as *triaging* or *gating*.

Gating is not as easy as one might think. The meaning of that word is pretty obvious when it refers to testing activities (i.e., no test should fail after applying the proposed modifications), but it is not trivial when we talk about static analyzers' findings. Maintainers may decide to accept commits introducing violations and leave to a later time the decision whether to deviate or address these findings. The final decision lies in the hands of the maintainers, supported but not constrained by the analyzer.

A first possible gating criteria is the number of findings. Having zero findings with respect to a guideline means that it is generally accepted by the entire community and effectively enforced by the maintainers. We likely want to keep the project clean, so for a selection of truly adopted guidelines we expect the number of findings not to increase: if that happens, then it should be investigated why a violation is introduced, and whether it is desirable to deviate if such a violation is shown not to impact safety.

Another gating criteria is the introduction of new findings. A project may contain a considerable amount of findings for some guidelines that are generally undesired, but with no real intention of addressing them. In these cases, for the community it is enough to avoid the introduction of new ones. This is not equivalent to saying that the number of findings should not increase, because in this way a commit can introduce new (and perhaps different) violations if it resolves an higher number. This criterion requires tools with the ability of recognizing new findings from those already present in previous analyses, for instance by comparing their fingerprints. The fingerprint should be stable and not be affected by minor changes (such as a report location being moved within a file because of an unrelated change). This ability is not very common: only static analyzers that have been designed with that use case in mind can support this gating criterion.

The integration of a static analyzer into a CI system should be as smooth as possible for an effective adoption. Providing a view of the findings in an easy way to all contributors is a key point: suitable static analysis tools have to support advanced reporting facilities, like online reports browsing, differential outputs, and support for the combination of results from the analysis of the different build configurations. The more powerful the report browsing is (filters, IDE integration, ...), the more likely are contributors to use it. For OSS projects, a publicly available CI pipeline is essential to discuss the adoption of rules and address the violations that were found through static analysis.

Having a MISRA compliance check as part of a project's CI system, paired with a set of appropriate gating criteria, leads to the establishment of a process that can be referred to as "Continuous MISRA Compliance," where every time something is changed, its impact on the overall MISRA compliance is assessed and recorded in a suitable format. This is in contrast to the more common paradigm of assessing project compliance only periodically throughout the development cycle, and typically only when the audit date is approaching. Reversing this paradigm can lead to a better estimate of the compliance targets that can be achieved within a given time frame, and avoid potential pitfalls such as merging some elaborate and time-consuming code changes, only to find out later that these subtly introduced a great number of additional violations that could have been prevented more easily if they were identified earlier. All this is well known in the software engineering community [25]: we believe it is time to systematically leverage the possibilities offered by CI systems for MISRA compliance as well.

E. MISRA and Open Source Software

It has been argued that MISRA and OSS are not a good match, but none of the arguments are really convincing. One of those concerns the fact that key MISRA publications are not available for free, which is true, but the same applies to the C and C++ standards, which are actually much more expensive: nobody thinks that C/C++ and OSS are not a good match because of that. Moreover, "open-source" is not a synonym for "resource-less:" many OSS projects are backed up by large commercial organization that can and do invest money on their safety-qualification.

Another widely held but unjustified belief concerns the presumed incompatibility between the discipline required by MISRA compliance and the governance model of most OSS projects. Indeed, our experience shows the opposite. In particular, in OSS projects there is less the attitude "do as you are told" that you can find in some closed source projects. While this, on the one hand, can sometimes result in a slower initial adoption of the MISRA guidelines, the results attained are often of very high quality. In other words, in OSS projects, once the message "a MISRA violation is a call to examine the code critically" has passed, then the critical examination (as opposed to the blind compliance with the rule) does really take place and code quality is significantly improved. Moreover, due to the open nature of the projects, the discussions and alternatives considered, with their pros and cons, are often publicly available. This transparency enables the full traceability of the compliance process, which can further strengthen the argument of compliance, since the full evidence of the MISRA compliance process and spirit being followed can be obtained as a byproduct of the openness of the development processes.

VII. Conclusion

In this paper, we have looked at some of the challenges posed by the need to make existing software compliant with respect to the MISRA guidelines. We have also illustrated how such challenges can be tackled successfully by what we might call "the three Ts": suitable *training* of the personnel involved, judicious *tailoring* of the MISRA guidelines, and powerful *tooling* for their effective enforcement. Even though our recommendations are based on the experience gained over several years on a number of projects in different industry sectors, the paper's examples are all based on recent work on the compliance of the Xen hypervisor, and for which all the results are publicly visible to anyone on the Internet. While the paper is particularly concerned with open-source software, most of the recommendations apply to other software development models.

References

[1] M. Neukirchner. (2023, Oct.) The magic touch? How to safety-qualify open-source software. [Online]. Available: https://linkedin.com/pulse/magic-touch-how-safety-qualify-open-source-software-neukirchner

[2] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software," in *Static Analysis: Proceedings of the 25th International Symposium (SAS 2018)*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 11002. Freiburg, Germany: Springer International Publishing, 2018, pp. 5–23.

[3] ——, "The MISRA C coding standard: A key enabler for the development of safety- and security-critical embedded software," in *embedded world Conference 2019 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2019, pp. 543–553.

[4] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 3: Concept phase*. Geneva, Switzerland: ISO, Dec. 2018.

[5] ——, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 1: Vocabulary*. Geneva, Switzerland: ISO, Dec. 2018.

[6] D. D. Clark. (1992, Jul.) A cloudy crystal ball: Visions of the future. Plenary presentation at 24th meeting of the Internet Engineering Task Force, Cambridge, Mass., 13–17 July 1992. [Online]. Available: https://groups.csail.mit.edu/ana/People/DDC/future_ietf_92.pdf

[7] MISRA, *MISRA-C:2012 — Guidelines for the use of the C language critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2019, third edition, first revision.

[8] ——, *MISRA C:2023 — Guidelines for the use of the C language critical systems*. Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Apr. 2023, third edition, second revision.

[9] ——, *MISRA C++:2023 — Guidelines for the use of C++17 in critical systems*. Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Oct. 2023.

[10] R. Bagnara, A. Bagnara, and P. M. Hill, "Formal verification of software architectural constraints," in *embedded world Conference 2023 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2023, pp. 271–279.

[11] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.

[12] CENELEC, *EN 50128:2011/A2:2020: Railway applications — Communication, signalling and processing systems — Software for railway control and protection systems*. Brussels, Belgium: CENELEC, Aug. 2020, amendment A2 to EN 50128:2011.

[13] ——, *EN 50657:2017/A1:2023: Railway applications — Rolling stock applications — Software on Board Rolling Stock*. Brussels, Belgium: CENELEC, Nov. 2023, amendment A1 to EN 50657:2017.

[14] IEC, *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. Geneva, Switzerland: IEC, Apr. 2010.

[15] ——, *IEC 62304:2006/Amd 1:2015: Medical device software — Software life cycle processes — Amendment 1*. Geneva, Switzerland: IEC, Jun. 2015.

[16] ——, *IEC 60335-1:2020: Household and Similar Electrical Appliances — Safety — Part 1: General Requirements*. Geneva, Switzerland: IEC, Sep. 2020.

[17] RTCA, SC-205, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Dec. 2011.

[18] R. Bagnara, A. Bagnara, and P. M. Hill, "A rationale-based classification of MISRA C guidelines," in *embedded world Conference 2022 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2022, pp. 440–451.

[19] M. Barr, *BARR-C:2018 — Embedded C Coding Standard*. www.barrgroup.com: Barr Group, 2018.

[20] R. Bagnara, M. Barr, and P. M. Hill, "BARR-C:2018 and MISRA C:2012 (with Amendment 2): Synergy between the two most widely used C coding standards," in *embedded world Conference 2021 DIGITAL — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2021, pp. 378–391.

[21] MISRA, *MISRA Compliance:2020 — Achieving compliance with MISRA Coding Guidelines*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.

[22] ——, *MISRA C:2012 Amendment 2 — Updates for ISO/IEC 9899:2011 Core functionality*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.

[23] ISO/IEC, *ISO/IEC 9899:1999/Cor 3:2007: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2007, Technical Corrigendum 3.

[24] R. Bagnara, A. Bagnara, and P. M. Hill, "Coding guidelines and undecidability," in *embedded world Conference 2023 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2023, pp. 488–499.

[25] C. Jones and O. Bonsignour, *The Economics of Software Quality*, 1st ed. Addison-Wesley Professional, 2011.