By DR. DENNIS KENGO OKA, Principal Automotive Security Strategist, Synopsys Software Integrity Group www.synopsys.com/software, and DR. RALF HUUCK, CEO & Founder, Logilica, www.logilica.com

Practical Considerations for MISRA and AUTOSAR Coding Compliance

Automotive is shifting left, introducing safety and security assurances earlier in the development process. We show you how to shift left with MISRA and AUTOSAR code compliance for ISO 26262 and ISO/SAE 21434. Learn how to create practical strategies to get to market faster safely.

Modern automotive software has increasingly stringent requirements on safe and secure development. Besides development ECUs in classical domains such as powertrain, chassis, and body electronics, more advanced new systems are being developed such as digital cockpits, infotainment systems, autonomous driving systems, and connectivity units.

Standards such as ISO 26262[1] and ISO/SAE 21434[2] provide guidance to automotive organizations on a higher level regarding how to consider safety and security. However, more specifically for coding and considering source-code quality, safety, and security, there are specific coding guidelines such as MISRA[3,4] and AUTOSAR coding guidelines.[5]

What is MISRA and AUTOSAR?

The Motor Industry Software Reliability Association (MISRA) defines a set of guidelines and directives for C/C++ software development mostly applied to safety critical systems in domains such as automotive, defense, and avionics.

MISRA-C:2012[3] including Amendment 1[6] and 2,[7] which is the latest version currently available, defines 175 development guidelines split into 158 *rules* and 17 *directives,* and further categorizes each rule into *mandatory*, *required* and *advisory*. Moreover, to assist organizations on how to validate the guidelines, each guideline is defined as *decidable* or *undecidable*. Decidable guidelines can be verified using a static-analysis tool; however, for the undecidable guidelines, static analysis can only generate an incomplete picture, thus producing potential false positives/negatives.

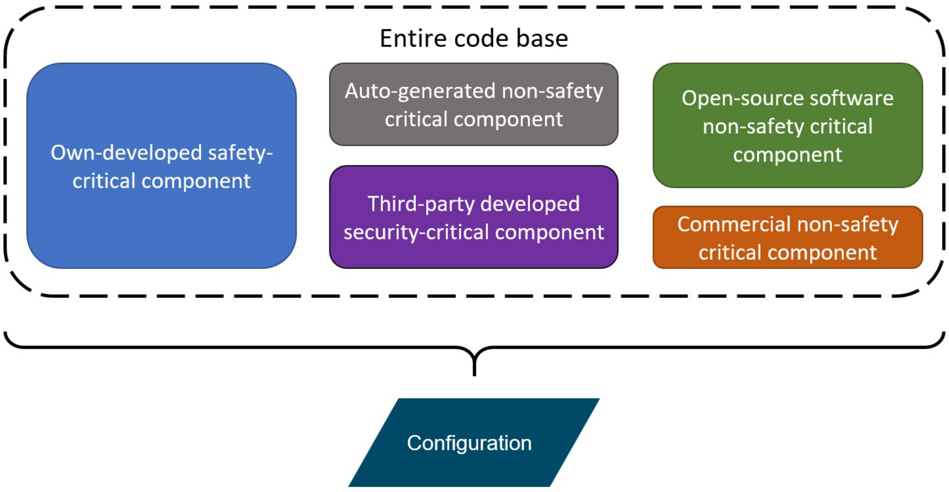
In addition, AUTomotive Open System ARchitecture (AUTOSAR) defines a set of guidelines called AUTOSAR Coding Guidelines C++.[5] These are considered as an update of the MISRA-C++:2008[4] standard and are currently being revised by the MISRA C++ committee.

Furthermore, in 2016, MISRA published the MISRA Compliance:2016 document,[8] which serves as a guide for organizations on how to achieve MISRA compliance. This document was updated in 2020[9] and serves as the formal guide for the compliance process. It defines deliverables for process compliance, including a *Guideline Enforcement Plan*, a *Guideline Reclassification Plan*, and a *Compliance Summary*.

Those deliveries are supported by optional *Deviation Records* and *Deviation Permits*. Specifically, it's worth mentioning that rules categorized as advisory may be reclassified as disapplied in *Guideline Reclassification Plan*, meaning that those rules would not be applicable to the specific project.

In addition, the MISRA Compliance document defines rules and constraints around what compliance means, under which constraints guidelines can be reclassified, and under which circumstances deviations from the guidelines are permitted. On top of that, guidance is given on how to deal with so-called *Adopted Code*, such as third-party binaries or open-source software.

Although the MISRA Compliance document provides guidance, many organizations seem to be struggling by often just applying vendor tools with MISRA or AUTOSAR coding checkers enabled. They're not well-aware of the compliance definitions and the freedom within the compliance process.


Entire code base
Own-developed safety-critical component
Auto-generated non-safety critical component
Third-party developed security-critical component
Open-source software non-safety critical component
Commercial non-safety critical component
Configuration

1. Example of code segmentation, where the code base is segmented into various software components.

Challenges

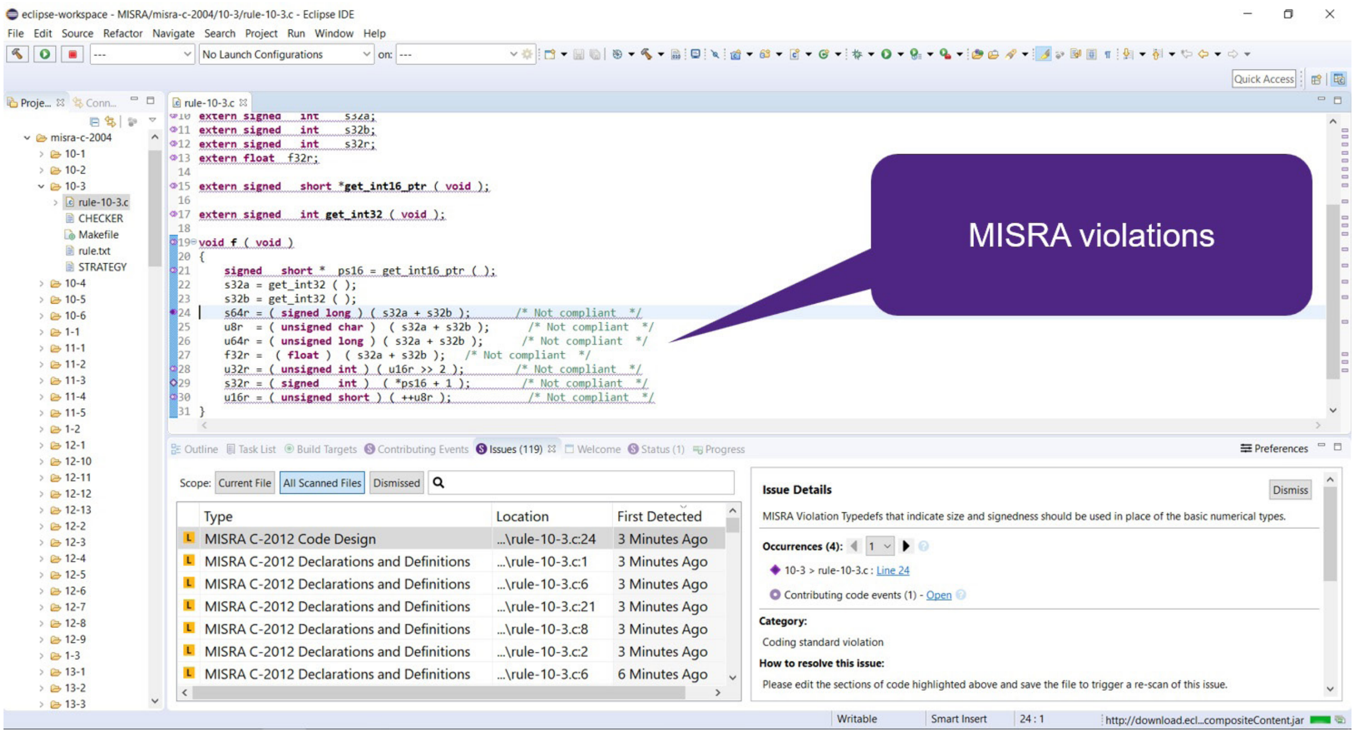
While on the surface it may seem straightforward to achieve compliance by following some coding guidelines during development, in practice there are multiple challenges. First, software in new systems such as digital cockpits, infotainment systems, autonomous driving systems, and connectivity units often consists of code from various sources, including own-developed code, third-party-developed code, commercial software, auto-generated code, and open-source software.

Trying to achieve coding compliance on the entire code base is a major challenge since some parts of the software may not have been developed with the MISRA or AUTOSAR coding guidelines in mind. Thus, scanning the entire code base for coding compliance to MISRA or AUTOSAR would typically generate an exceptionally large number of coding violations, which is unfeasible for an organization to realistically process.

More importantly, many of these findings may have low priority based on the specific guideline type, or the type of software component that's perhaps deemed not relevant to safety or security. The large number of findings make it challenging for an organization to identify the top priority issues that must be addressed first. This challenge is further exacerbated by the fact that code bases contain increasingly growing amounts of software from various sources.

Overview of the Solution

It's crucial for organizations to have a clear understanding of the parts of the code base and the coding guidelines with the highest priority so that violations are efficiently handled.


eclipse-workspace - MISRA/misra-c-2004/10-3/rule-10-3.c - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
No Launch Configurations
rule-10-3.c
misra-c-2004
rule-10-3.c
CHECKER
Makefile
rule.txt
STRATEGY
Outline Task List Build Targets Contributing Events Issues (119) Welcome Status (1) Progress
Preferences
Scope: Current File All Scanned Files Dismissed
Dismiss
Writable Smart Insert 24 : 1
MISRA violations

```
extern signed   int    s32a;
extern signed   int    s32b;
extern signed   int    s32r;
extern float  f32r;

extern signed   short *get_int16_ptr ( void );

extern signed   int get_int32 ( void );

void f ( void )
{
    signed   short *  ps16 = get_int16_ptr ( );
    s32a = get_int32 ( );
    s32b = get_int32 ( );
    s64r = ( signed long ) ( s32a + s32b );        /* Not compliant  */
    u8r  = ( unsigned char )  ( s32a + s32b );      /* Not compliant  */
    u64r = ( unsigned long ) ( s32a + s32b );      /* Not compliant  */
    f32r =  ( float )  ( s32a + s32b );   /* Not compliant  */
    u32r = ( unsigned int ) ( u16r >> 2 );        /* Not compliant  */
    s32r = ( signed   int )  ( *ps16 + 1 );        /* Not compliant  */
    u16r = ( unsigned short ) ( ++u8r );           /* Not compliant  */
}
```

Type	Location	First Detected
MISRA C-2012 Code Design	...\rule-10-3.c:24	3 Minutes Ago
MISRA C-2012 Declarations and Definitions	...\rule-10-3.c:1	3 Minutes Ago
MISRA C-2012 Declarations and Definitions	...\rule-10-3.c:6	3 Minutes Ago
MISRA C-2012 Declarations and Definitions	...\rule-10-3.c:21	3 Minutes Ago
MISRA C-2012 Declarations and Definitions	...\rule-10-3.c:8	3 Minutes Ago
MISRA C-2012 Declarations and Definitions	...\rule-10-3.c:2	3 Minutes Ago
MISRA C-2012 Declarations and Definitions	...\rule-10-3.c:6	6 Minutes Ago

Issue Details

MISRA Violation Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Occurrences (4): 1

10-3 > rule-10-3.c : Line 24

Contributing code events (1) - Open

Category:

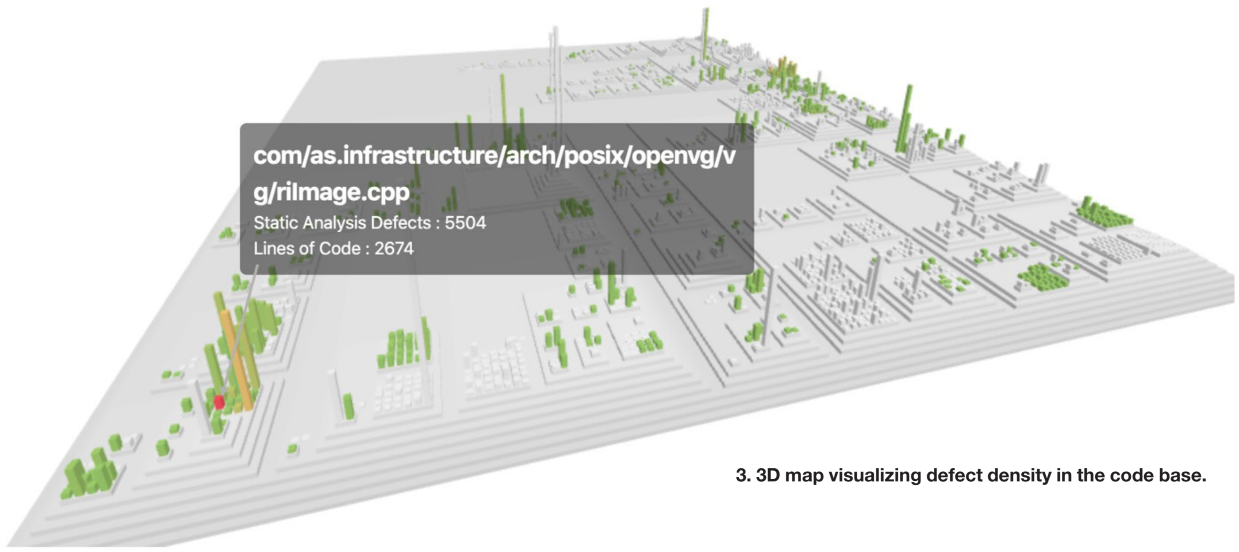
Coding standard violation

How to resolve this issue:

Please edit the sections of code highlighted above and save the file to trigger a re-scan of this issue.

http://download.ecl...compositeContent.jar

2. Examples of MISRA findings detected by a static code analysis tool.


com/as.infrastructure/arch/posix/openvg/v
g/rlImage.cpp
Static Analysis Defects : 5504
Lines of Code : 2674

3. 3D map visualizing defect density in the code base.

As mentioned, a naïve approach is to scan the entire code base using a static-code-analysis tool with all coding guideline checkers enabled, which generates numerous findings. To address this challenge, a solution based on a two-step process is described as follows.

As a first step, organizations need to identify relevant coding guidelines for relevant parts in the code base and create an appropriate configuration for the software of the target system. It may be possible to apply results from a hazard analysis and risk assessment (HARA) or threat analysis and risk assessment (TARA) to more accurately identify safety- and security-relevant software components in the code base.

That code base can then be segmented into various components, e.g., own-developed safety-critical component, auto-generated non-safety critical component, third-party developed security-critical component, open-source software non-safety critical component, commercial non-safety critical component etc. A simplified example is depicted in *Figure 1*.

Furthermore, the organization can determine which coding guidelines are appropriate for which parts of the code base. For example, certain rules may be more applicable to safety- and security-critical components, and less applicable to non-safety- and non-security-critical components. Then a static-code-analysis tool is configured accordingly. It can be used to scan the code base regularly to only check for certain coding guidelines relevant to the specific software components to achieve more efficient scanning.

It's important to note that an organization should use a static-code-analysis tool with a broad coverage of coding guideline checkers to achieve better results.[10,11] Examples of MISRA findings identified by a static-code-analysis tool called Coverity[12] is shown in *Figure 2*.

The static-code-analysis tool generates results that are processed by a data-analytics tool in the second step. The objective is to explore the result set of potentially still thousands if not hundreds of thousands of findings and create a customer burn-down strategy.

The Logilica Insights tool[13] provides analytics capabilities similar to those found in Business Intelligence solutions and combines them with visual representations. These allow an organization to explore the findings and gain insights more easily into the relevant MISRA Compliance strategy for the code base.

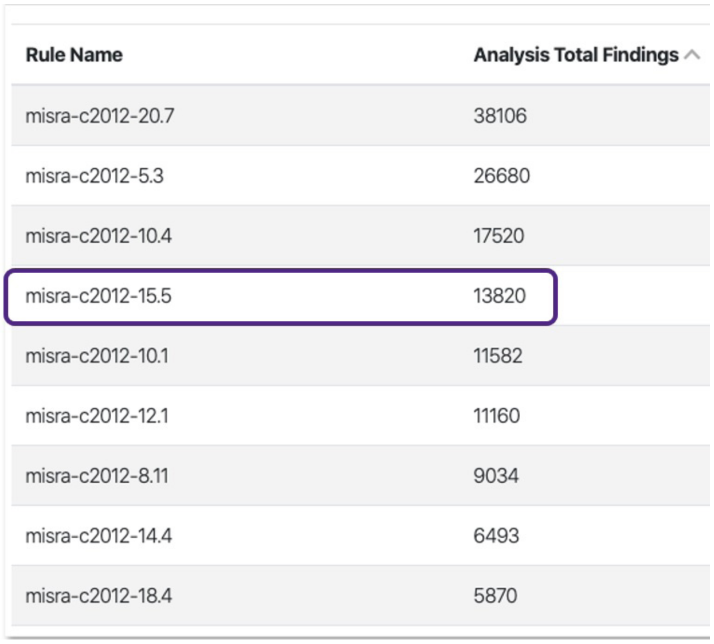
Besides common charting and visualization techniques, Logilica employs so-called CodeCities,[14] which are 3D maps of software repositories. Each file is displayed as a building and folders are displayed as platforms. Metrics can be overlayed to determine the size and color of the buildings. An example of MISRA findings generated by a static-code-analysis tool using this technique is illustrated in *Figure 3*.

The height of the building reflects the size of the file and the color of the building indicates the MISRA defect density (findings per code size). For example, the building colored in red in the figure has a high defect density and may indicate that this is something an organization should look at first with higher priority. Moreover, this visual representation can help identify hotspots, i.e., particular code areas that contain large numbers of violations. Organizations can then further investigate what may be causing these hotspots.

Benefits

This solution has a number of benefits. For instance, it can help organizations better identify the top offending rules. Furthermore, it can provide an understanding of the location of these violations—i.e., which components and which files—and help define a compliance strategy.

In the first step, it's imperative to get the tooling and processes right. The static-analysis tool is configured using the specific configuration of relevant coding guidelines for the target software to allow for more efficient scanning that enables the scan to be performed regularly (e.g., daily). In the second step, it's possible for developers and engineering management to gain clear insights into the current status of the project using the data-analytics tool. For instance, it would be possible for an organization to easily identify whether a large number of findings are detected in *adopted*



Rule Name	Analysis Total Findings
misra-c2012-20.7	38106
misra-c2012-5.3	26680
misra-c2012-10.4	17520
misra-c2012-15.5	13820
misra-c2012-10.1	11582
misra-c2012-12.1	11160
misra-c2012-8.11	9034
misra-c2012-14.4	6493
misra-c2012-18.4	5870

4. Top offending MISRA rules from an example project.

code (e.g., an open-source software component), or if certain specific rules generate a significant number of findings.

Based on those insights, the organization can define an appropriate compliance strategy, perhaps to start burn-down in non-adopted code or exclude advisory rules. For example, as shown in *Figure 4*, among the top offending rules for an example project is *Rule 15.5* with 13,820 findings. This rule is categorized as *advisory* and for this project could be reclassified as *disapplied*, meaning that it would be possible to ignore these 13,820 findings. These strategies help organizations to prioritize and allow software developers to focus on working on the right areas.

As automotive systems continue to advance and contain more complex software, including software from various sources such as own-developed code, third-party developed code, commercial software, and open-source software components, software compliance is naturally becoming a greater challenge. To overcome these challenges and put coding compliance into practice, automotive organizations need to establish proper workflows and adopt appropriate technical solutions.[15]

Dr. Dennis Kengo Oka is an automotive cybersecurity expert with more than 15 years of global experience in the automotive industry. He received his Ph.D. in automotive security, focusing on solutions for the connected car. Dennis has over 60 publications consisting of conference papers, journal articles, and books, and is a frequent public speaker at international automotive and cybersecurity conferences and events.

Dr. Ralf Huuck is CEO and founder of Logilica, Australia, a leading analytics solution provider for the software lifecycle space. Ralf has been in the automotive application security and software testing space for over 20 years. He served in executive and senior technical roles at Synopsys, Red Lizard Software, and Australia's national R&D lab NICTA. Ralf is an Adjunct Associate Professor at UNSW, Australia, in the field of software technology with over 50 peer reviewed publications to his name.

References

1. ISO, "ISO 26262 - Road vehicles—Functional safety," 2018.
2. ISO/SAE International, "ISO/SAE DIS 21434 - Road Vehicles—Cybersecurity engineering," 2020.
3. MISRA, "MISRA C:2012 Guidelines for the use of the C language in critical systems," 2013.
4. MISRA, "MISRA C++:2008 Guidelines for the Use of the C++ Language in Critical Systems," 2008.
5. AUTOSAR, "Guidelines for the use of the C++14 language in critical and safety-related systems," 2019.
6. MISRA, "MISRA C:2012 Amendment 1 - Additional security guidelines for MISRA C:2012," 2016.
7. MISRA, "MISRA C:2012 Amendment 2 - Updates for ISO/IEC 9899:2011 Core functionality," 2020.
8. MISRA, "MISRA Compliance:2016 - Achieving compliance with MISRA Coding Guidelines," 2016.
9. MISRA, "MISRA Compliance:2020 - Achieving compliance with MISRA Coding Guidelines," 2020.
10. Synopsys, "Coverity Support for MISRA Coding Standards," 2020.
11. Synopsys, "Coverity Support for AUTOSAR Coding Standards," 2020.
12. Synopsys, "Coverity Static Application Security Testing," https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.
13. Logilica Insights, https://logilica.com.
14. R. Wettel, and M. Lanza, "CodeCity: 3D visualization of large-scale software" Proceedings - International Conference on Software Engineering, 2008. 921-922. 10.1145/1370175.1370188.
15. D. K. Oka, and R. Huuck, "How to Put MISRA and AUTOSAR Coding Compliance into Practice," *Embedded World*, 2021