QA-MISRA

Compliance Matrices for

MISRA C++:2008
AUTOSAR C++14

Release 23.04, b13183398

April 18, 2023

QA Systems GmbH
powered by AbsInt Angewandte Informatik GmbH

CONTACT:

QA Systems GmbH
support@qa-systems.com
www.qa-systems.com
www.qa-systems.de/tools/qa-misra/

Known Safety Issues:

www.absint.com/known-issues/qa-misra/23.04.md

Contents

1 Introduction

QA-MISRA is a static analyzer that checks for violations of coding guidelines such as MISRA. It supports the MISRA-C:2004, MISRA C:2012, MISRA C++:2008, AUTOSAR C++14, ISO/IEC TS 17961:2013, CERT, JSF AV C++, and CWE rule sets, as well as rules for coding style and thresholds for code metrics.

Astrée (`https://www.absint.com/astree/index.htm`) is a static code analyzer that proves the absence of runtime errors and invalid concurrent behavior in safety-critical software written or generated in C or C++. Astrée and **QA-MISRA** can be seamlessly integrated. Using **QA-MISRA** in conjunction with the sound semantic analyses offered by Astrée guarantees zero false negatives and minimizes false positives on semantical rules.

1.1 Terms and Definitions

If not stated otherwise for a specific set of guidelines, the degree of rule support is classified as follows.

fully checked A rule is *fully checked (FC)* if the checks adhere exactly to the rule text and the analysis will never miss a rule violation. For fully checked rules, absence of alarms means the tool can prove the absence of violations of this rule. False alarms may be issued.

This degree of support may be raised to *fully checked + exact (FC+E)* if the absence of false alarms can be guaranteed.

partially checked A rule is *partially checked (PC)* if the checks either check only some aspects or a (simplifying) reformulation of the rule (text) and/or the rule may miss rule violations. For partially checked rules, absence of alarms does not imply absence of rule violations. False alarms may be issued.

This degree of support may be raised to *partially checked + soundly supported (PC + S)* if activating **Astrée**'s semantic analysis underpins the rule check by issuing semantic alarms for violations of the rule and by proving the absence of violations of some aspects of the rule or if the analyzer's frontend implicitly checks some aspects of the rule.

(soundly) supported A rule is classified as *(soundly) supported (S)* if there are no dedicated checks, but an analysis run may produce evidence whether or not the rule is broken. This compliance level may require that the user provides appropriate analysis stubs.

For example, the rule "No reliance shall be placed on undefined or unspecified behavior." (MISRA-C:2004, rule 1.2) is supported by **Astrée** because **Astrée** reports undefined and unspecified behavior.

not checked A rule is *not checked (NC)* if there are no dedicated checks and checking the rule is not supported by the analyzer.

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

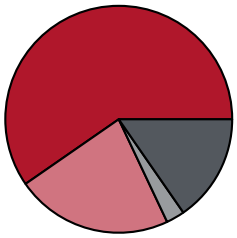
2 MISRA C++:2008

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

In total, 189 rules of the rule set – i. e. 82% of all 228 rules – are checked:

	All Rules	Required	Advisory	Documentation
fully checked	136 (59 %)	125 (63 %)	10 (55 %)	1 (8 %)
partially checked	51 (22 %)	46 (23 %)	4 (22 %)	1 (8 %)
implicitly checkable	2 (0 %)	1 (0 %)	0 (0 %)	1 (8 %)
not checked	39 (17 %)	26 (13 %)	4 (22 %)	9 (75 %)



Language independent issues		**Support**
0.1.1	A project shall not contain unreachable code. Violations of this rule are reported for code that cannot be reached by the analyzer. Such code is definitely unreachable except if the analysis terminated prematurely because of an error. It cannot be guaranteed that all unreachable code is reported.	*PC*
0.1.2	A project shall not contain infeasible paths.	*PC*
0.1.3	A project shall not contain unused variables.	*PC*
0.1.4	A project shall not contain non-volatile POD variables having only one use.	*FC+E*
0.1.5	A project shall not contain unused type declarations.	*PC*
0.1.6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	*PC*
0.1.7	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.	*FC+E*
0.1.8	All functions with void return type shall have external side effect(s).	*NC*
0.1.9	There shall be no dead code.	*PC*

continues on the next page…

Language independent issues		Support
...continued		
0.1.10	Every defined function shall be called at least once.	*PC*
0.1.11	There shall be no unused parameters (named or unnamed) in non-virtual functions.	*PC*
0.1.12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	*NC*
0.2.1	An object shall not be assigned to an overlapping object.	*PC*
0.3.1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	*NC*
0.3.2	If a function generates error information, then that error information shall be tested.	*PC*
0.4.1	Use of scaled-integer or fixed-point arithmetic shall be documented.	*NC*
0.4.2	Use of floating-point arithmetic shall be documented.	*NC*
0.4.3	Floating-point implementations shall comply with a defined floating-point standard.	*NC*

General		Support
1.0.1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1". The frontend rejects in large part violations of ISO/IEC 14882.	*S*
1.0.2	Multiple compilers shall only be used if they have a common, defined interface. This rule applies to the used compiler and cannot be checked at the source code level.	*NC*
1.0.3	The implementation of integer division in the chosen compiler shall be determined and documented.	*NC*

Lexical conventions		Support
2.2.1	The character set and the corresponding encoding shall be documented.	*NC*
2.3.1	Trigraphs shall not be used.	*FC+E*
2.5.1	Digraphs should not be used.	*FC+E*
2.7.1	The character sequence /* shall not be used within a C-style comment.	*FC+E*
2.7.2	Sections of code shall not be "commented out" using C-style comments.	*NC*
2.7.3	Sections of code should not be "commented out" using C++ comments.	*NC*
2.10.1	Different identifiers shall be typographically unambiguous.	*FC+E*
2.10.2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	*PC*
2.10.3	A typedef name (including qualification, if any) shall be a unique identifier.	*FC+E*
2.10.4	A class, union or enum name (including qualification, if any) shall be a unique identifier.	*FC+E*
2.10.5	The identifier name of a non-member object or function with static storage duration should not be reused.	*NC*
2.10.6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	*NC*
2.13.1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	*PC*
2.13.2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	*FC+E*
2.13.3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	*PC*
2.13.4	Literal suffixes shall be upper case.	*FC+E*
2.13.5	Narrow and wide string literals shall not be concatenated.	*FC+E*

Basic concepts		Support
3.1.1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	*NC*

continues on the next page…

Basic concepts		Support
...continued		
3.1.2	Functions shall not be declared at block scope.	*FC+E*
3.1.3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	*FC+E*
3.2.1	All declarations of an object or function shall have compatible types.	*PC*
3.2.2	The One Definition Rule shall not be violated.	*PC*
3.2.3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	*FC+E*
3.2.4	An identifier with external linkage shall have exactly one definition.	*PC*
3.3.1	Objects or functions with external linkage shall be declared in a header file.	*FC+E*
3.3.2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	*FC+E*
3.4.1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	*NC*
3.9.1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	*PC*
3.9.2	Typedefs that indicate size and signedness should be used in place of the basic numerical types. Only use of typedefs is checked, not whether their names indicate size and signedness. This rule check is supported by Astrée's semantic analysis: Astrée reports overflows resulting from incorrect assumptions about the size/signedness of numerical types.	*PC*
3.9.3	The underlying bit representations of floating-point values shall not be used.	*PC*

Standard conversions		Support
4.5.1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	*FC+E*

continues on the next page...

Standard conversions		Support
...continued		
4.5.2	Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator,and the relational operators <, <=, >, >=.	*FC+E*
4.5.3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.	*FC+E*
4.10.1	NULL shall not be used as an integer value.	*FC+E*
4.10.2	Literal zero (0) shall not be used as the null-pointer-constant.	*FC+E*

Expressions		Support
5.0.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	*PC*
5.0.2	Limited dependence should be placed on C++ operator precedence rules in expressions.	*PC*
5.0.3	A cvalue expression shall not be implicitly converted to a different underlying type.	*FC+E*
5.0.4	An implicit integral conversion shall not change the signedness of the underlying type.	*FC+E*
5.0.5	There shall be no implicit floating-integral conversions.	*FC+E*
5.0.6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	*FC+E*
5.0.7	There shall be no explicit floating-integral conversions of a cvalue expression.	*FC+E*
5.0.8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	*FC+E*
5.0.9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	*FC+E*
5.0.10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	*FC+E*
5.0.11	The plain char type shall only be used for the storage and use of character values.	*FC+E*

continues on the next page...

Expressions		Support
...continued		
5.0.12	signed char and unsigned char type shall only be used for the storage and use of numeric values.	*FC+E*
5.0.13	The condition of an if-statement and the condition of an iteration-statement shall have type bool.	*FC+E*
5.0.14	The first operand of a conditional-operator shall have type bool.	*FC+E*
5.0.15	Array indexing shall be the only form of pointer arithmetic.	*PC*
5.0.16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	*NC*
5.0.17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	*NC*
5.0.18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	*NC*
5.0.19	The declaration of objects shall contain no more than two levels of pointer indirection.	*FC+E*
5.0.20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	*FC+E*
5.0.21	Bitwise operators shall only be applied to operands of unsigned underlying type.	*FC+E*
5.2.1	Each operand of a logical && or || shall be a postfix expression.	*FC+E*
5.2.2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	*FC+E*
5.2.3	Casts from a base class to a derived class should not be performed on polymorphic types.	*FC+E*
5.2.4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	*FC+E*
5.2.5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	*FC+E*
5.2.6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	*FC+E*
5.2.7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	*PC*

continues on the next page...

Expressions		Support
...continued		
	Indirect conversion, e.g. via intermediate integral types, is undecidable and thus not covered.	
5.2.8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	*FC+E*
5.2.9	A cast should not convert a pointer type to an integral type.	*FC+E*
5.2.10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	*FC+E*
5.2.11	The comma operator, && operator and the || operator shall not be overloaded.	*FC+E*
5.2.12	An identifier with array type passed as a function argument shall not decay to a pointer.	*FC+E*
5.3.1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	*FC+E*
5.3.2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	*FC+E*
5.3.3	The unary & operator shall not be overloaded.	*FC+E*
5.3.4	Evaluation of the operand to the sizeof operator shall not contain side effects.	*FC*
5.8.1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	*PC*
5.14.1	The right hand operand of a logical && or || operator shall not contain side effects.	*FC*
5.17.1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	*NC*
5.18.1	The comma operator shall not be used.	*FC+E*
5.19.1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	*NC*

Statements		Support
6.2.1	Assignment operators shall not be used in sub-expressions.	*FC+E*

continues on the next page...

Statements		Support
...continued		
6.2.2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	*PC*
6.2.3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.	*FC+E*
6.3.1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	*FC+E*
6.4.1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	*FC+E*
6.4.2	All if ... else if constructs shall be terminated with an else clause.	*FC+E*
6.4.3	A switch statement shall be a well-formed switch statement.	*FC+E*
6.4.4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	*FC+E*
6.4.5	An unconditional throw or break statement shall terminate every non-empty switch-clause.	*FC+E*
6.4.6	The final clause of a switch statement shall be the default-clause.	*FC+E*
6.4.7	The condition of a switch statement shall not have bool type.	*FC+E*
6.4.8	Every switch statement shall have at least one case-clause.	*FC+E*
6.5.1	A for loop shall contain a single loop-counter which shall not have floating type.	*PC*
6.5.2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	*PC*
6.5.3	The loop-counter shall not be modified within condition or statement.	*PC*
6.5.4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.	*PC*
6.5.5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	*PC*
6.5.6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	*PC*

continues on the next page...

Statements		Support
...continued		
6.6.1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	*FC+E*
6.6.2	The goto statement shall jump to a label declared later in the same function body.	*FC+E*
6.6.3	The continue statement shall only be used within a well-formed for loop.	*PC*
6.6.4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	*FC+E*
6.6.5	A function shall have a single point of exit at the end of the function.	*FC+E*

Declarations		Support
7.1.1	A variable which is not modified shall be const qualified.	*NC*
7.1.2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	*NC*
7.2.1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	*NC*
7.3.1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	*FC+E*
7.3.2	The identifier main shall not be used for a function other than the global function main.	*FC+E*
7.3.3	There shall be no unnamed namespaces in header files.	*FC+E*
7.3.4	using-directives shall not be used.	*FC+E*
7.3.5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	*FC+E*
7.3.6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	*FC+E*
7.4.1	All usage of assembler shall be documented.	*PC*
7.4.2	Assembler instructions shall only be introduced using the asm declaration.	*PC*

continues on the next page...

Declarations		Support
...continued		
7.4.3	Assembly language shall be encapsulated and isolated.	*FC+E*
7.5.1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	*PC*
7.5.2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	*NC*
7.5.3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	*PC*
7.5.4	Functions should not call themselves, either directly or indirectly.	*PC*

Declarators		Support
8.0.1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	*FC+E*
8.3.1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments. It is impossible to decide in the general case whether two expressions initializing the same default argument have the same value.	*PC*
8.4.1	Functions shall not be defined using the ellipsis notation.	*FC+E*
8.4.2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	*FC+E*
8.4.3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	*FC+E*
8.4.4	A function identifier shall either be used to call the function or it shall be preceded by &.	*FC+E*
8.5.1	All variables shall have a defined value before they are used.	*PC*
8.5.2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	*FC+E*
8.5.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	*FC+E*

Classes		Support
9.3.1	const member functions shall not return non-const pointers or references to class-data.	*PC*
9.3.2	Member functions shall not return non-const handles to class-data.	*PC*
9.3.3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const. Violations of this rule are not reported for templates as all possible instantiations need to be known to decide whether a function can be made const.	*PC*
9.5.1	Unions shall not be used.	*FC+E*
9.6.1	When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.	*NC*
9.6.2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	*NC*
9.6.3	Bit-fields shall not have enum type.	*FC+E*
9.6.4	Named bit-fields with signed integer type shall have a length of more than one bit.	*FC+E*

Derived classes		Support
10.1.1	Classes should not be derived from virtual bases.	*FC+E*
10.1.2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	*FC+E*
10.1.3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	*FC+E*
10.2.1	All accessible entity names within a multiple inheritance hierarchy should be unique.	*PC*
10.3.1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	*FC+E*
10.3.2	Each overriding virtual function shall be declared with the virtual keyword.	*FC+E*
10.3.3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	*FC+E*

Member access control		Support
11.0.1	Member data in non-POD class types shall be private.	*FC+E*

Special member functions		Support
12.1.1	An object's dynamic type shall not be used from the body of its constructor or destructor.	*PC*
12.1.2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	*FC+E*
12.1.3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	*FC+E*
12.8.1	A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.	*FC+E*
12.8.2	The copy assignment operator shall be declared protected or private in an abstract class.	*FC+E*

Templates		Support
14.5.1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	*FC+E*
14.5.2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	*FC+E*
14.5.3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	*FC+E*
14.6.1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	*NC*
14.6.2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	*NC*
14.7.1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	*NC*
14.7.2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	*NC*

continues on the next page…

Templates		Support
...continued		
14.7.3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	*FC+E*
14.8.1	Overloaded function templates shall not be explicitly specialized.	*NC*
14.8.2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	*NC*

Exception handling		Support
15.0.1	Exceptions shall only be used for error handling.	*NC*
15.0.2	An exception object should not have pointer type.	*FC+E*
15.0.3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	*FC+E*
15.1.1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	*NC*
15.1.2	NULL shall not be thrown explicitly.	*FC+E*
15.1.3	An empty throw (throw;) shall only be used in the compound-statement of a catch handler.	*FC+E*
15.3.1	Exceptions shall be raised only after start-up and before termination of the program.	*NC*
15.3.2	There should be at least one exception handler to catch all otherwise unhandled exceptions.	*FC+E*
15.3.3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	*FC+E*
15.3.4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	*NC*
15.3.5	A class type exception shall always be caught by reference.	*FC+E*
15.3.6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	*FC+E*

continues on the next page...

Exception handling		Support
...continued		
15.3.7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	*FC+E*
15.4.1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	*PC*
15.5.1	A class destructor shall not exit with an exception.	*NC*
15.5.2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	*NC*
15.5.3	The terminate() function shall not be called implicitly.	*PC*

Preprocessing directives		Support
16.0.1	#include directives in a file shall only be preceded by other preprocessor directives or comments.	*FC+E*
16.0.2	Macros shall only be #define'd or #undef'd in the global namespace.	*FC+E*
16.0.3	#undef shall not be used.	*FC+E*
16.0.4	Function-like macros shall not be defined.	*FC+E*
16.0.5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	*FC+E*
16.0.6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	*FC+E*
16.0.7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator. Note that invalid directives reached by the preprocessor are reported as an error.	*FC+E*
16.0.8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	*FC+E*
16.1.1	The defined preprocessor operator shall only be used in one of the two standard forms.	*FC*

continues on the next page...

Preprocessing directives		Support
…continued		
16.1.2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	*FC+E*
16.2.1	The pre-processor shall only be used for file inclusion and include guards.	*PC*
16.2.2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	*PC*
16.2.3	Include guards shall be provided.	*FC+E*
16.2.4	The ’, ", /* or // characters shall not occur in a header file name.	*FC+E*
16.2.5	The \ character should not occur in a header file name.	*FC+E*
16.2.6	The #include directive shall be followed by either a <filename> or "filename" sequence.	*FC+E*
16.3.1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	*FC+E*
16.3.2	The # and ## operators should not be used.	*FC+E*
16.6.1	All uses of the #pragma directive shall be documented.	*FC+E*

Library introduction		Support
17.0.1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	*PC*
17.0.2	The names of standard library macros and objects shall not be reused.	*PC*
17.0.3	The names of standard library functions shall not be overridden.	*NC*
17.0.4	All library code shall conform to MISRA C++. Library code can be checked for MISRA C++:2008 compliance by either adding it to the checked project or analyzing it separately.	*S*
17.0.5	The setjmp macro and the longjmp function shall not be used.	*FC+E*

Language support library		Support
18.0.1	The C library shall not be used.	*FC+E*

continues on the next page…

Language support library		Support
…continued		
18.0.2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	*FC+E*
18.0.3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	*FC+E*
18.0.4	The time handling functions of library <ctime> shall not be used.	*FC+E*
18.0.5	The unbounded functions of library <cstring> shall not be used.	*FC+E*
18.2.1	The macro offsetof shall not be used.	*FC+E*
18.4.1	Dynamic heap memory allocation shall not be used.	*PC*
18.7.1	The signal handling facilities of <csignal> shall not be used.	*FC+E*

Diagnostics library		Support
19.3.1	The error indicator errno shall not be used.	*FC+E*

Input/output library		Support
27.0.1	The stream input/output library <cstdio> shall not be used.	*FC+E*

3 AUTOSAR C++14

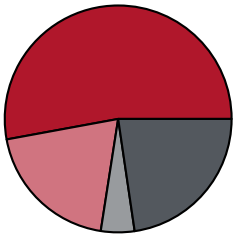
Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

The *AUTOSAR C++14* rule set is based on [4].

In total, 292 rules of the rule set – i. e. 73% of all 397 rules – are checked:

	All Rules	Required	Advisory
fully checked	210 (52 %)	193 (53 %)	17 (48 %)
partially checked	78 (19 %)	70 (19 %)	8 (22 %)
implicitly checkable	4 (1 %)	3 (0 %)	1 (2 %)
not checked	105 (26 %)	96 (26 %)	9 (25 %)



Language independent issues		**Support**
0.1.1A	A project shall not contain instances of non-volatile variables being given values that are not subsequently used.	*PC*
0.1.1M	A project shall not contain unreachable code. Violations of this rule are reported for code that cannot be reached by the analyzer. Such code is definitely unreachable except if the analysis terminated prematurely because of an error. It cannot be guaranteed that all unreachable code is reported.	*PC*
0.1.2A	The value returned by a function having a non-void return type that is not an overloaded operator shall be used.	*FC+E*
0.1.2M	A project shall not contain infeasible paths.	*PC*
0.1.3A	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.	*PC*
0.1.3M	A project shall not contain unused variables.	*PC*
0.1.4A	There shall be no unused named parameters in non-virtual functions.	*FC+E*

continues on the next page…

Language independent issues		Support
...continued		
0.1.4M	A project shall not contain non-volatile POD variables having only one use.	*FC+E*
0.1.5A	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.	*FC+E*
0.1.6A	There should be no unused type declarations.	*PC*
0.1.8M	All functions with void return type shall have external side effect(s).	*NC*
0.1.9M	There shall be no dead code.	*PC*
0.1.10M	Every defined function shall be called at least once.	*PC*
0.2.1M	An object shall not be assigned to an overlapping object.	*PC*
0.3.1M	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	*NC*
0.3.2M	If a function generates error information, then that error information shall be tested.	*PC*
0.4.1A	Floating-point implementation shall comply with IEEE 754 standard.	*NC*
0.4.1M	Use of scaled-integer or fixed-point arithmetic shall be documented.	*NC*
0.4.2A	Type long double shall not be used.	*FC+E*
0.4.2M	Use of floating-point arithmetic shall be documented.	*NC*
0.4.3A	The implementations in the chosen compiler shall strictly comply with the C++14 Language Standard. This rule applies to the used compiler and cannot be checked at the source code level.	*NC*
0.4.4A	Range, domain and pole errors shall be checked when using math functions. This rule check is supported by Astrée's semantic analysis: Astrée provides analysis stubs for library functions that raise alarms for arguments that may cause runtime errors. Without Astrée, the tool checks this rule with reduced coverage and precision.	*PC*

General		Support
1.0.2M	Multiple compilers shall only be used if they have a common, defined interface. This rule applies to the used compiler and cannot be checked at the source code level.	*NC*
1.1.1A	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features. The frontend rejects in large part violations of ISO/IEC 14882 and warns about the use of deprecated features.	*PC + S*
1.1.2A	A warning level of the compilation process shall be set in compliance with project policies. This rule applies to the used compiler and cannot be checked at the source code level.	*NC*
1.1.3A	An optimization option that disregards strict standard compliance shall not be turned on in the chosen compiler. This rule applies to the used compiler and cannot be checked at the source code level.	*NC*
1.2.1A	When using a compiler toolchain (including preprocessor, compiler itself, linker, C++ standard libraries) in safety-related software, the tool confidence level(TCL) shall be determined.In case of TCL2 or TCL3, the compiler shall undergo a "Qualification of a software tool", as per ISO 26262 - 8.11.4.6 [6]. This rule applies to the used compiler and cannot be checked at the source code level.	*NC*
1.4.1A	Code metrics and their valid boundaries shall be defined and code shall comply with defined boundaries of code metrics. The tool supports the computation of various code metrics and provides configurable threshold checks.	*S*
1.4.3A	All code should compile free of compiler warnings. The frontend in large part issues the same or similar warnings.	*S*

Lexical conventions		Support
2.3.1A	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.	*FC+E*
2.5.1A	Trigraphs shall not be used.	*FC+E*

continues on the next page…

Lexical conventions		Support
…continued		
2.5.2A	Digraphs shall not be used.	*FC+E*
2.7.1A	The character \ shall not occur as a last character of a C++ comment.	*FC+E*
2.7.1M	The character sequence /* shall not be used within a C-style comment.	*FC+E*
2.7.2A	Sections of code shall not be "commented out".	*NC*
2.7.3A	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.	*NC*
2.7.5A	Comments shall not document any actions or sources (e.g. tables, figures, paragraphs, etc.) that are outside of the file.	*NC*
2.8.1A	A header file name should reflect the logical entity for which it provides declarations.	*NC*
2.8.2A	An implementation file name should reflect the logical entity for which it provides definitions.	*NC*
2.10.1A	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	*PC*
2.10.1M	Different identifiers shall be typographically unambiguous.	*FC+E*
2.10.4A	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.	*NC*
2.10.5A	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.	*NC*
2.10.6A	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.	*NC*
2.11.1A	Volatile keyword shall not be used.	*FC+E*
2.13.1A	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.	*FC+E*
2.13.2A	String literals with different encoding prefixes shall not be concatenated.	*FC+E*
2.13.2M	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	*FC+E*

continues on the next page…

Lexical conventions		Support
...continued		
2.13.3A	Type wchar_t shall not be used.	*FC+E*
2.13.3M	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	*PC*
2.13.4A	String literals shall not be assigned to non-constant pointers.	*FC+E*
2.13.4M	Literal suffixes shall be upper case.	*FC+E*
2.13.5A	Hexadecimal constants should be upper case.	*FC+E*
2.13.6A	Universal character names shall be used only inside character or string literals.	*FC+E*

Basic concepts		Support
3.1.1A	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	*NC*
3.1.2A	Header files, that are defined locally in the project, shall have a file name extension of one of: ".h", ".hpp" or ".hxx".	*FC+E*
3.1.2M	Functions shall not be declared at block scope.	*FC+E*
3.1.3A	Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".	*FC+E*
3.1.4A	When an array with external linkage is declared, its size shall be stated explicitly.	*PC*
3.1.5A	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template. From the source code it is not derivable whether a function is "intended to be inlined".	*NC*
3.1.6A	Trivial accessor and mutator functions should be inlined.	*PC*
3.2.1M	All declarations of an object or function shall have compatible types.	*PC*
3.2.2M	The One Definition Rule shall not be violated.	*PC*
3.2.3M	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	*FC+E*
3.2.4M	An identifier with external linkage shall have exactly one definition.	*PC*

continues on the next page...

Basic concepts		Support
...continued		
3.3.1A	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.	*FC+E*
3.3.2A	Static and thread-local objects shall be constant-initialized.	*FC+E*
3.3.2M	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	*FC+E*
3.4.1M	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	*NC*
3.8.1A	An object shall not be accessed outside of its lifetime.	*PC*
3.9.1A	Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types. Only use of typedefs is checked, not whether their names indicate size and signedness. This rule check is supported by Astrée's semantic analysis: Astrée reports overflows resulting from incorrect assumptions about the size/signedness of numerical types.	*PC*
3.9.1M	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	*PC*
3.9.3M	The underlying bit representations of floating-point values shall not be used.	*PC*

Standard conversions		Support
4.5.1A	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [ ], the assignment operator =, the equality operators == and ! =, the unary & operator, and the relational operators <, <=, >, >=.	*FC+E*
4.5.1M	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	*FC+E*
4.5.3M	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.	*FC+E*
4.7.1A	An integer expression shall not lead to data loss.	*NC*

continues on the next page...

Standard conversions		Support
…continued		
	This rule check is supported by Astrée's semantic analysis: Astrée raises alarms for conversion overflows and arithmetic overflows.	
4.10.1A	Only nullptr literal shall be used as the null-pointer-constant.	*FC+E*
4.10.1M	NULL shall not be used as an integer value.	*FC+E*
4.10.2M	Literal zero (0) shall not be used as the null-pointer-constant.	*FC+E*

Expressions		Support
5.0.1A	The value of an expression shall be the same under any order of evaluation that the standard permits.	*PC*
5.0.2A	The condition of an if-statement and the condition of an iteration statement shall have type bool.	*FC+E*
5.0.2M	Limited dependence should be placed on C++ operator precedence rules in expressions.	*PC*
5.0.3A	The declaration of objects shall contain no more than two levels of pointer indirection.	*FC+E*
5.0.3M	A cvalue expression shall not be implicitly converted to a different underlying type.	*FC+E*
5.0.4A	Pointer arithmetic shall not be used with pointers to non-final classes.	*FC+E*
5.0.4M	An implicit integral conversion shall not change the signedness of the underlying type.	*FC+E*
5.0.5M	There shall be no implicit floating-integral conversions.	*FC+E*
5.0.6M	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	*FC+E*
5.0.7M	There shall be no explicit floating-integral conversions of a cvalue expression.	*FC+E*
5.0.8M	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	*FC+E*
5.0.9M	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	*FC+E*

continues on the next page…

Expressions		Support
...continued		
5.0.10M	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	*FC+E*
5.0.11M	The plain char type shall only be used for the storage and use of character values.	*FC+E*
5.0.12M	signed char and unsigned char type shall only be used for the storage and use of numeric values.	*FC+E*
5.0.14M	The first operand of a conditional-operator shall have type bool.	*FC+E*
5.0.15M	Array indexing shall be the only form of pointer arithmetic.	*PC*
5.0.16M	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	*NC*
5.0.17M	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	*NC*
5.0.18M	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	*NC*
5.0.20M	Non-constant operands to a binary bitwise operator shall have the same underlying type.	*FC+E*
5.0.21M	Bitwise operators shall only be applied to operands of unsigned underlying type.	*FC+E*
5.1.1A	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.	*FC*
5.1.2A	Variables shall not be implicitly captured in a lambda expression.	*FC+E*
5.1.3A	Parameter list (possibly empty) shall be included in every lambda expression.	*FC+E*
5.1.4A	A lambda expression object shall not outlive any of its reference-captured objects.	*NC*
5.1.6A	Return type of a non-void return type lambda expression should be explicitly specified.	*FC+E*
5.1.7A	A lambda shall not be an operand to decltype or typeid.	*FC+E*
5.1.8A	Lambda expressions should not be defined inside another lambda expression.	*FC+E*

continues on the next page...

Expressions		Support
...continued		
5.1.9A	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.	*PC*
5.2.1A	dynamic_cast should not be used.	*FC+E*
5.2.2A	Traditional C-style casts shall not be used.	*FC+E*
5.2.2M	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	*FC+E*
5.2.3A	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	*FC+E*
5.2.3M	Casts from a base class to a derived class should not be performed on polymorphic types.	*FC+E*
5.2.4A	reinterpret_cast shall not be used.	*FC+E*
5.2.5A	An array or container shall not be accessed beyond its range.	*NC*
5.2.6A	The operands of a logical && or || shall be parenthesized if the operands contain binary operators.	*FC+E*
5.2.6M	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	*FC+E*
5.2.8M	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	*FC+E*
5.2.9M	A cast should not convert a pointer type to an integral type.	*FC+E*
5.2.10M	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	*FC+E*
5.2.11M	The comma operator, && operator and the || operator shall not be overloaded.	*FC+E*
5.2.12M	An identifier with array type passed as a function argument shall not decay to a pointer.	*FC+E*
5.3.1A	Evaluation of the operand to the typeid operator shall not contain side effects.	*FC+E*
5.3.1M	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	*FC+E*
5.3.2A	Null pointers shall not be dereferenced.	*PC*
5.3.2M	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	*FC+E*

continues on the next page...

Expressions		Support
...continued		
5.3.3A	Pointers to incomplete class types shall not be deleted.	*FC+E*
5.3.3M	The unary & operator shall not be overloaded.	*FC+E*
5.3.4M	Evaluation of the operand to the sizeof operator shall not contain side effects.	*FC*
5.5.1A	A pointer to member shall not access non-existent class members.	*NC*
5.6.1A	The right hand operand of the integer division or remainder operators shall not be equal to zero.	*PC*
5.8.1M	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	*PC*
5.10.1A	A pointer to member virtual function shall only be tested for equality with null-pointer-constant.	*PC*
5.14.1M	The right hand operand of a logical && or || operator shall not contain side effects.	*FC*
5.16.1A	The ternary conditional operator shall not be used as a sub-expression.	*FC+E*
5.17.1M	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	*NC*
5.18.1M	The comma operator shall not be used.	*FC+E*
5.19.1M	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	*NC*

Statements		Support
6.2.1A	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.	*NC*
6.2.1M	Assignment operators shall not be used in sub-expressions.	*FC+E*
6.2.2A	Expression statements shall not be explicit calls to constructors of temporary objects only.	*FC+E*
6.2.2M	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	*PC*

continues on the next page...

Statements		Support
...continued		
6.2.3M	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.	*FC+E*
6.3.1M	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	*FC+E*
6.4.1A	A switch statement shall have at least two case-clauses, distinct from the default label. The number of required cases is configurable. The default is 2.	*FC+E*
6.4.1M	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	*FC+E*
6.4.2M	All if ... else if constructs shall be terminated with an else clause.	*FC+E*
6.4.3M	A switch statement shall be a well-formed switch statement.	*FC+E*
6.4.4M	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	*FC+E*
6.4.5M	An unconditional throw or break statement shall terminate every non-empty switch-clause.	*FC+E*
6.4.6M	The final clause of a switch statement shall be the default-clause.	*FC+E*
6.4.7M	The condition of a switch statement shall not have bool type.	*FC+E*
6.5.1A	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.	*NC*
6.5.2A	A for loop shall contain a single loop-counter which shall not have floating-point type.	*PC*
6.5.2M	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	*PC*
6.5.3A	Do statements should not be used.	*FC+E*
6.5.3M	The loop-counter shall not be modified within condition or statement.	*PC*
6.5.4A	For-init-statement and expression should not perform actions other than loop-counter initialization and modification.	*NC*
6.5.4M	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.	*PC*

continues on the next page...

Statements		Support
...continued		
6.5.5M	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	*PC*
6.5.6M	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	*PC*
6.6.1A	The goto statement shall not be used.	*FC+E*
6.6.1M	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	*FC+E*
6.6.2M	The goto statement shall jump to a label declared later in the same function body.	*FC+E*
6.6.3M	The continue statement shall only be used within a well-formed for loop.	*PC*

Declaration		Support
7.1.1A	Constexpr or const specifiers shall be used for immutable data declaration.	*NC*
7.1.2A	The constexpr specifier shall be used for values that can be determined at compile time.	*NC*
7.1.2M	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	*NC*
7.1.3A	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.	*NC*
7.1.4A	The register keyword shall not be used.	*FC+E*
7.1.5A	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.	*FC*
7.1.6A	The typedef specifier shall not be used.	*FC+E*
7.1.7A	Each expression statement and identifier declaration shall be placed on a separate line.	*PC*

continues on the next page...

Declaration		Support
...continued		
7.1.8A	A non-type specifier shall be placed before a type specifier in a declaration.	*NC*
7.1.9A	A class, structure, or enumeration shall not be declared in the definition of its type.	*FC+E*
7.2.1A	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	*PC*
7.2.2A	Enumeration underlying base type shall be explicitly defined.	*FC+E*
7.2.3A	Enumerations shall be declared as scoped enum classes.	*FC+E*
7.2.4A	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.	*FC+E*
7.2.5A	Enumerations should be used to represent sets of related named constants.	*NC*
7.3.1A	All overloads of a function shall be visible from where it is called.	*NC*
7.3.1M	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	*FC+E*
7.3.2M	The identifier main shall not be used for a function other than the global function main.	*FC+E*
7.3.3M	There shall be no unnamed namespaces in header files.	*FC+E*
7.3.4M	using-directives shall not be used.	*FC+E*
7.3.6M	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	*FC+E*
7.4.1A	The asm declaration shall not be used.	*FC+E*
7.4.1M	All usage of assembler shall be documented.	*PC*
7.4.2M	Assembler instructions shall only be introduced using the asm declaration.	*PC*
7.4.3M	Assembly language shall be encapsulated and isolated.	*FC+E*
7.5.1A	A function shall not return a reference or a pointer to a parameter that is passed by reference to const.	*PC*
7.5.1M	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	*PC*

continues on the next page...

Declaration		Support
...continued		
7.5.2A	Functions shall not call themselves, either directly or indirectly.	*PC*
7.5.2M	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	*NC*
7.6.1A	Functions declared with the [[noreturn]] attribute shall not return.	*FC*

Declarators		Support
8.0.1M	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	*FC+E*
8.2.1A	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.	*FC*
8.3.1M	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments. It is impossible to decide in the general case whether two expressions initializing the same default argument have the same value.	*PC*
8.4.1A	Functions shall not be defined using the ellipsis notation.	*FC+E*
8.4.2A	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	*FC+E*
8.4.2M	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	*FC+E*
8.4.3A	Common ways of passing parameters should be used.	*NC*
8.4.4A	Multiple output values from a function should be returned as a struct or tuple.	*PC*
8.4.4M	A function identifier shall either be used to call the function or it shall be preceded by &.	*FC+E*
8.4.5A	"consume" parameters declared as X && shall always be moved from.	*FC+E*
8.4.6A	"forward" parameters declared as T && shall always be forwarded.	*FC+E*
8.4.7A	"in" parameters for "cheap to copy" types shall be passed by value.	*FC+E*

continues on the next page...

Declarators		Support
...continued		
8.4.8A	Output parameters shall not be used.	*PC*
8.4.9A	"in-out" parameters declared as T & shall be modified.	*NC*
8.4.10A	A parameter shall be passed by reference if it can't be NULL	*NC*
8.4.11A	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	*NC*
8.4.12A	A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	*NC*
8.4.13A	A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a const lvalue reference to express that the function retains a reference count.	*NC*
8.4.14A	Interfaces shall be precisely and strongly typed.	*NC*
8.5.0A	All memory shall be initialized before it is read.	*PC*
8.5.1A	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.	*FC+E*
8.5.2A	Braced-initialization {}, without equals sign, shall be used for variable initialization.	*PC*
8.5.2M	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	*FC+E*
8.5.3A	A variable of type auto shall not be initialized using {} or ={} braced-initialization.	*FC+E*
8.5.4A	If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors.	*FC+E*

Classes		Support
9.3.1A	Member functions shall not return non-const "raw" pointers or references to private or protected data owned by the class.	*PC*

continues on the next page...

Classes		Support
…continued		
9.3.1M	const member functions shall not return non-const pointers or references to class-data.	*PC*
9.3.3M	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const. Violations of this rule are not reported for templates as all possible instantiations need to be known to decide whether a function can be made const.	*PC*
9.5.1A	Unions shall not be used. The exception for "tagged unions" is not taken into account. It is suggested to encapsulate and annotate unions which shall be excluded from this check.	*FC+E*
9.6.1A	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes. Data types used to interface to hardware or conforming to communication protocols have to be explicitly configured.	*FC*
9.6.1M	When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.	*NC*
9.6.2A	Bit-fields shall be used only when interfacing to hardware or conforming to communication protocols. Structs/unions used to interface to hardware or conforming to communication protocols have to be explicitly configured.	*FC*
9.6.4M	Named bit-fields with signed integer type shall have a length of more than one bit.	*FC+E*

Derived Classes		Support
10.0.1A	Public inheritance shall be used to implement "is-a" relationship.	*NC*
10.0.2A	Membership or non-public inheritance shall be used to implement "has-a" relationship.	*NC*
10.1.1A	Class shall not be derived from more than one base class which is not an interface class.	*FC+E*

continues on the next page…

Derived Classes		Support
...continued		
10.1.1M	Classes should not be derived from virtual bases.	*FC+E*
10.1.2M	A base class shall only be declared virtual if it is used in a diamond hierarchy.	*FC+E*
10.1.3M	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	*FC+E*
10.2.1A	Non-virtual public or protected member functions shall not be redefined in derived classes.	*FC+E*
10.2.1M	All accessible entity names within a multiple inheritance hierarchy should be unique.	*PC*
10.3.1A	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.	*FC+E*
10.3.2A	Each overriding virtual function shall be declared with the override or final specifier.	*FC+E*
10.3.3A	Virtual functions shall not be introduced in a final class.	*FC+E*
10.3.3M	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	*FC+E*
10.3.5A	A user-defined assignment operator shall not be virtual.	*FC+E*
10.4.1A	Hierarchies should be based on interface classes.	*NC*

Member access control		Support
11.0.1A	A non-POD type should be defined as class.	*FC+E*
11.0.1M	Member data in non-POD class types shall be private.	*FC+E*
11.0.2A	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.	*FC+E*
11.3.1A	Friend declarations shall not be used.	*FC+E*

Special member functions		Support
12.0.1A	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.	*FC+E*
12.0.2A	Bitwise operations and operations that assume data representation in memory shall not be performed on objects.	*NC*
12.1.1A	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.	*FC+E*
12.1.1M	An object's dynamic type shall not be used from the body of its constructor or destructor.	*PC*
12.1.2A	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.	*FC+E*
12.1.3A	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.	*FC+E*
12.1.4A	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	*FC+E*
12.1.5A	Common class initialization for non-constant members shall be done by a delegating constructor.	*NC*
12.1.6A	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.	*NC*
12.4.1A	Destructor of a base class shall be public virtual, public override or protected non-virtual.	*FC+E*
12.4.2A	If a public destructor of a class is non-virtual, then the class should be declared final.	*FC+E*
12.6.1A	All class data members that are initialized by the constructor shall be initialized using member initializers.	*PC*
12.7.1A	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined.	*PC*
12.8.1A	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.	*NC*
12.8.2A	User-defined copy and move assignment operators should use user-defined no-throw swap function.	*FC*

continues on the next page...

Special member functions		Support
...continued		
12.8.3A	Moved-from object shall not be read-accessed.	*NC*
12.8.4A	Move constructor shall not initialize its class members and base classes using copy semantics.	*FC+E*
12.8.5A	A copy assignment and a move assignment operators shall handle self-assignment.	*NC*
12.8.6A	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.	*FC+E*
12.8.7A	Assignment operators should be declared with the ref-qualifier &.	*FC+E*

Overloading		Support
13.1.2A	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.	*FC+E*
13.1.3A	User defined literals operators shall only perform conversion of passed parameters.	*NC*
13.2.1A	An assignment operator shall return a reference to "this".	*FC+E*
13.2.2A	A binary arithmetic operator and a bitwise operator shall return a "prvalue".	*FC+E*
13.2.3A	A relational operator shall return a boolean value.	*FC+E*
13.3.1A	A function that contains "forwarding reference" as its argument shall not be overloaded.	*PC*
13.5.1A	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.	*FC+E*
13.5.2A	All user-defined conversion operators shall be defined explicit.	*FC+E*
13.5.3A	User-defined conversion operators should not be used.	*FC+E*
13.5.4A	If two opposite operators are defined, one shall be defined in terms of the other.	*PC*
13.5.5A	Comparison operators shall be non-member functions with identical parameter types and noexcept.	*PC*

continues on the next page...

Overloading		Support
...continued		
13.6.1A	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.	*FC+E*

Templates		Support
14.1.1A	A template should check if a specific template argument is suitable for this template.	*NC*
14.5.1A	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type.	*NC*
14.5.2A	Class members that are not dependent on template class parameters should be defined in a separate base class.	*PC*
14.5.3A	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations.	*FC+E*
14.5.3M	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	*FC+E*
14.6.1M	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	*NC*
14.7.1A	A type used as a template argument shall provide all members that are used by the template. The frontend rejects the instantiation of a template with a type if that type is missing required members and the templated code requiring the members is used.	*S*
14.7.2A	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.	*FC+E*
14.8.2A	Explicit specializations of function templates shall not be used.	*FC+E*

Exception handling		Support
15.0.1A	A function shall not exit with an exception if it is able to complete its task.	*NC*

continues on the next page...

Exception handling		Support
…continued		
15.0.2A	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee	*NC*
15.0.3A	Exception safety guarantee of a called function shall be considered.	*NC*
15.0.3M	Control shall not be transferred into a try or catch block using a goto or a switch statement.	*FC+E*
15.0.4A	Unchecked exceptions shall be used to represent errors from which the caller cannot reasonably be expected to recover.	*NC*
15.0.5A	Checked exceptions shall be used to represent errors from which the caller can reasonably be expected to recover.	*NC*
15.0.6A	An analysis shall be performed to analyze the failure modes of exception handling. In particular, the following failure modes shall be analyzed: (a) worst time execution time not existing or cannot be determined, (b) stack not correctly unwound, (c) exception not thrown, other exception thrown, wrong catch activated, (d) memory not available while exception handling.	*NC*
15.0.7A	Exception handling mechanism shall guarantee a deterministic worst-case time execution time.	*NC*
15.0.8A	A worst-case execution time (WCET) analysis shall be performed to determine maximum execution time constraints of the software, covering in particular the exceptions processing.	*NC*
15.1.1A	Only instances of types derived from std::exception should be thrown.	*FC+E*
15.1.1M	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	*NC*
15.1.2A	An exception object shall not be a pointer.	*FC+E*
15.1.2M	NULL shall not be thrown explicitly.	*FC+E*
15.1.3A	All thrown exceptions should be unique.	*NC*
15.1.3M	An empty throw (throw;) shall only be used in the compound-statement of a catch handler.	*FC+E*
15.1.4A	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	*NC*

continues on the next page…

Exception handling		Support
…continued		
15.1.5A	Exceptions shall not be thrown across execution boundaries.	*NC*
15.2.1A	Constructors that are not noexcept shall not be invoked before program startup.	*FC+E*
15.2.2A	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception.	*NC*
15.3.1M	Exceptions shall be raised only after start-up and before termination of the program.	*NC*
15.3.2A	If a function throws an exception, it shall be handled when meaningful actions can be taken, otherwise it shall be propagated.	*NC*
15.3.3A	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions. The separation of the handling of the various exception classes listed by this rule is not checked.	*PC*
15.3.3M	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	*FC+E*
15.3.4A	Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.	*NC*
15.3.4M	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	*NC*
15.3.5A	A class type exception shall be caught by reference or const reference.	*FC+E*
15.3.6M	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	*FC+E*
15.3.7M	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	*FC+E*
15.4.1A	Dynamic exception-specification shall not be used.	*FC+E*

continues on the next page…

Exception handling		Support
…continued		
15.4.2A	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition >), then it shall not exit with an exception.	*PC*
15.4.3A	The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider.	*PC*
15.4.4A	A declaration of non-throwing function shall contain noexcept specification.	*NC*
15.4.5A	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.	*NC*
15.5.1A	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.	*PC*
15.5.2A	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.	*PC*
15.5.3A	The std::terminate() function shall not be called implicitly.	*PC*

Preprocessing directives		Support
16.0.1A	The pre-processor shall only be used for unconditional and conditional file inclusion and include guards, and using the following directives: (1) #ifndef, (2) #ifdef, (3) #if, (4) #if defined, (5) #elif, (6) #else, (7) #define, (8) #endif, (9) #include.	*PC*
16.0.1M	#include directives in a file shall only be preceded by other preprocessor directives or comments.	*FC+E*
16.0.2M	Macros shall only be #define'd or #undef'd in the global namespace.	*FC+E*
16.0.5M	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	*FC+E*
16.0.6M	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	*FC+E*

continues on the next page…

Preprocessing directives		Support
...continued		
16.0.7M	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	*FC+E*
16.0.8M	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	*FC+E*
16.1.1M	The defined preprocessor operator shall only be used in one of the two standard forms.	*FC*
16.1.2M	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	*FC+E*
16.2.1A	The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.	*FC+E*
16.2.2A	There shall be no unused include directives.	*NC*
16.2.3A	An include directive shall be added explicitly for every symbol used in a file.	*NC*
16.2.3M	Include guards shall be provided.	*FC+E*
16.3.1M	There shall be at most one occurrence of the # or ## operators in a single macro definition.	*FC+E*
16.3.2M	The # and ## operators should not be used.	*FC+E*
16.6.1A	#error directive shall not be used.	*FC+E*
16.7.1A	The #pragma directive shall not be used.	*FC+E*

Library introduction - partial		Support
17.0.1A	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.	*PC*
17.0.2A	All project's code including used libraries (including standard and user-defined libraries) and any third-party user code shall conform to the AUTOSAR C++ 14 Coding Guidelines Library and any third-party user code can be checked for compliance by either adding it to the checked project or analyzing it separately.	*S*
17.0.2M	The names of standard library macros and objects shall not be reused.	*PC*
17.0.3M	The names of standard library functions shall not be overridden.	*NC*

continues on the next page...

Library introduction - partial		Support
...continued		
17.0.5M	The setjmp macro and the longjmp function shall not be used.	*FC+E*
17.1.1A	Use of the C Standard Library shall be encapsulated and isolated.	*NC*
17.6.1A	Non-standard entities shall not be added to standard namespaces.	*PC*

Language support library - partial		Support
18.0.1A	The C library facilities shall only be accessed through C++ library headers.	*FC+E*
18.0.2A	The error state of a conversion from string to a numeric value shall be checked.	*PC*
18.0.3A	The library <clocale> (locale.h) and the setlocale function shall not be used.	*PC*
18.0.3M	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	*FC+E*
18.0.4M	The time handling functions of library <ctime> shall not be used.	*FC+E*
18.0.5M	The unbounded functions of library <cstring> shall not be used.	*FC+E*
18.1.1A	C-style arrays shall not be used.	*FC+E*
18.1.2A	The std::vector<bool> specialization shall not be used.	*FC+E*
18.1.3A	The std::auto_ptr type shall not be used.	*FC+E*
18.1.4A	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.	*NC*
18.1.6A	All std::hash specializations for user-defined types shall have a noexcept function call operator.	*FC+E*
18.2.1M	The macro offsetof shall not be used.	*FC+E*
18.5.1A	Functions malloc, calloc, realloc and free shall not be used.	*FC+E*
18.5.2A	Non-placement new or delete expressions shall not be used.	*FC+E*
18.5.3A	The form of the delete expression shall match the form of the new expression used to allocate the memory.	*NC*
18.5.4A	If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.	*FC*

continues on the next page...

Language support library - partial		Support
...continued		
18.5.5A	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel.	*NC*
18.5.6A	An analysis shall be performed to analyze the failure modes of dynamic memory management. In particular, the following failure modes shall be analyzed: (a) non-deterministic behavior resulting with nonexistence of worst-case execution time, (b) memory fragmentation, (c) running out of memory, (d) mismatched allocations and deallocations, (e) dependence on non-deterministic calls to kernel.	*PC*
18.5.7A	If non-realtime implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-realtime program phases.	*NC*
18.5.8A	Objects that do not outlive a function shall have automatic storage duration.	*NC*
18.5.9A	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.	*PC*
18.5.10A	Placement new shall be used only with properly aligned pointers to sufficient storage capacity.	*PC*
18.5.11A	"operator new" and "operator delete" shall be defined together.	*PC*
18.7.1M	The signal handling facilities of <csignal> shall not be used.	*FC+E*
18.9.1A	The std::bind shall not be used.	*FC+E*
18.9.2A	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.	*FC+E*
18.9.3A	The std::move shall not be used on objects declared const or const&.	*FC+E*
18.9.4A	An argument to std::forward shall not be subsequently used.	*NC*

Diagnostics library - partial		Support
19.3.1M	The error indicator errno shall not be used.	*FC+E*

General utilities library - partial		Support
20.8.1A	An already-owned pointer value shall not be stored in an unrelated smart pointer.	*NC*
20.8.2A	A std::unique_ptr shall be used to represent exclusive ownership.	*NC*
20.8.3A	A std::shared_ptr shall be used to represent shared ownership.	*NC*
20.8.4A	A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required.	*NC*
20.8.5A	std::make_unique shall be used to construct objects owned by std::unique_ptr.	*FC+E*
20.8.6A	std::make_shared shall be used to construct objects owned by std::shared_ptr.	*FC+E*
20.8.7A	A std::weak_ptr shall be used to represent temporary shared ownership.	*NC*

Strings library		Support
21.8.1A	Arguments to character-handling functions shall be representable as an unsigned char.	*NC*

Containers library - partial		Support
23.0.1A	An iterator shall not be implicitly converted to const_iterator.	*NC*
23.0.2A	Elements of a container shall only be accessed via valid references, iterators, and pointers.	*NC*

Algorithms library		Support
25.1.1A	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied.	*NC*
25.4.1A	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation.	*NC*

Numerics library		Support
26.5.1A	Pseudorandom numbers shall not be generated using std::rand().	*FC+E*
26.5.2A	Random number engines shall not be default-initialized.	*FC+E*

Input/output library - partial		Support
27.0.1A	Inputs from independent components shall be validated.	*NC*
27.0.1M	The stream input/output library <cstdio> shall not be used.	*FC+E*
27.0.2A	A C-style string shall guarantee sufficient space for data and the null terminator.	*NC*
27.0.3A	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.	*NC*
27.0.4A	C-style strings shall not be used.	*NC*

Bibliography

[1] MISRA Limited. *MISRA-C:2004 Guidelines for the use of the C language in critical systems*. ISBN 978-0-9524156-4-0, October 2004.

[2] MISRA Limited. *MISRA-C:2012 Guidelines for the use of the C language in critical systems*. ISBN 978-1-906400-11-8, March 2013.

[3] QA Systems GmbH. ***QA-MISRA*** – *User Documentation*. Version 23.04, Build 13183398, April 18, 2023.

[4] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems (release 19-03). Release 19-03, March 2019.

[5] Carnegie Mellon University. SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. 2016.

[6] ISO. ISO/IEC TS 17961:2013(E): Information Technology–Programming Languages, Their Environments and System Software Interfaces–C Secure Coding Rules. November 2013.