

SOFTWARE QUALITY OBJECTIVES FOR SOURCE CODE

Revision table

Index	Date	Object of modification	Paragraphs modified
V1.0	15 th February 2009	Initial version	All
V2.0	5 th March 2010	<ul style="list-style-type: none"> - Improve formulation of SQRs - Explicit Code Metrics - Define systematic and potential runtime errors, safe and unreachable operations 	All
V3.0	2 nd May 2012	Add mapping with ISO 26262-6:2011 Add Rule 5.2 in the 1 st MISRA subset Add MISRA AC AGC subset Add MISRA C++ subset	§4 §3.5.1 §3.5.1, §3.5.2 §3.5.1, §3.5.2

Table of contents

REVISION TABLE	2
TABLE OF CONTENTS.....	3
1. REQUIREMENTS MAPPING TABLE	4
2. INTRODUCTION.....	4
3. SOFTWARE QUALITY OBJECTIVES (SQO)	5
3.1. SOFTWARE QUALITY OBJECTIVES OVERVIEW	5
3.2. QUALITY PLAN.....	5
3.2.1. Quality levels and number of deliveries.....	6
3.2.2. People and tools.....	7
3.2.3. Definitions of runtime errors	8
3.2.4. Standard comments and justifications.....	9
3.3. DETAILED DESIGN DESCRIPTION.....	11
3.3.1. Application level.....	11
3.3.2. Module level	11
3.3.3. File level	11
3.4. CODE METRICS	12
3.5. MISRA RULES SUBSETS	12
3.5.1. The first MISRA rules subset.....	13
3.5.2. The second MISRA rules subset.....	15
3.6. SYSTEMATIC RUNTIME ERRORS.....	18
3.7. NON TERMINATING FUNCTION CALLS AND LOOPS.....	19
3.8. UNREACHABLE BRANCHES.....	19
3.9. POTENTIAL RUNTIME ERRORS	20
3.10. DATAFLOW ANALYSIS.....	21
4. REQUIREMENT MAPPING WITH ISO 26262-6:2011	22
4.1. PURPOSE AND SCOPE	22
4.2. SUMMARY	22
4.3. DETAILS	23
4.3.1. Section 5: Initiation of product development at the software level	23
4.3.2. Section 8: Software unit design and implementation	24
4.3.3. Section 9: Software unit testing.....	25
4.3.4. Section 10: Software integration and testing.....	26
4.4. TRACEABILITY SQO LEVELS / ISO 26262 REQUIREMENTS	26
TO EASE THE READING OF THE TRACEABILITY MATRIX NEXT PAGE, THE SQRs ASSOCIATED WITH EACH SQO LEVEL ARE PROVIDED AGAIN IN THE FOLLOWING TABLE:.....	26
5. GLOSSARY	ERROR! BOOKMARK NOT DEFINED.

1. REQUIREMENTS MAPPING TABLE

Software Quality Requirements that are covered outside of the scope of this document can be mapped onto the requirements defined within this document. In this case the supplier shall provide a table detailing the mapping between the two set of requirements.

2. INTRODUCTION

The document defines a general and standard approach to measure the software quality of a product using criteria linked to code quality and dynamic execution errors.

The diagram on the right shows that the three properties: “Cost”, “Software Quality” and “Time” are interrelated. Changing the requirements for one property will impact the other two. In this context “Time” refers to the time required to deliver the product, “Quality” is the quality of the final product, and “Cost” refers to the total cost of designing and building the product.



Once the requirements for these properties have been defined the question is how to achieve them? An approach where all modules are tested until they meet the required quality could be applied, but the process of improving the quality is often stopped because the available time or budget has been used and not because we have obtained the quality objectives. A better verification process within the development process may help reduce time and cost to achieve quality.

3. SOFTWARE QUALITY OBJECTIVES (SQO)

This document defines six Software Quality Objectives (SQO) which are associated to four quality levels, level QL-1 (lowest quality) to level QL-4 (highest quality). Each Software Quality Objective consists of a set of Software Quality Requirements (SQR). Examples of SQR are complexity metrics, measure of unreachable code etc.

3.1. Software Quality Objectives overview

The following table shows the software quality criteria required to reach each of the six Software Quality Objectives:

Criteria	SQR	Objectives					
		SQO-1	SQO-2	SQO-3	SQO-4	SQO-5	SQO-6
Quality Plan	SQR-10 → SQR-100	X	X	X	X	X	X
Detailed design description	SQR-110 → SQR-130	X	X	X	X	X	X
Code metrics	SQR-140 → SQR-150	X	X	X	X	X	X
First MISRA-C:2004 rules subset	SQR-160 → SQR-170	X	X	X	X	X	X
Systematic runtime errors	SQR-200 → SQR-210		X	X	X	X	X
Non terminating constructs	SQR-220		X	X	X	X	X
Unreachable branches	SQR-230			X	X	X	X
First subset of potential runtime errors	SQR-240				X	X	X
Second MISRA-C:2004 rules subset	SQR-180 → SQR-190					X	X
Second subset of potential runtime errors	SQR-250					X	X
Third subset of potential runtime errors	SQR-260						X
Dataflow Analysis	SQR-270						X

The different criteria are described in the following paragraphs (Quality Plan in §3.2, Detailed design description in §3.3, etc.).

3.2. Quality Plan

This section describes the general information which shall be provided by a supplier. It covers information about the methods, tools and teams involved in the Software Quality Requirement fulfillment, as well as information about the project itself. This information shall help to better understand who performs the work, and how and where the work is done.

Note: We will use the word “team” to describe people who are using the tools providing information for the documentation and/or the persons writing the document.

Regarding integration to the development process, information concerning the SQO should be delivered by the supplier during the project life cycle. It is advised to perform the relevant SQO activities during the code review phase. This will make it simpler and faster to consolidate and deliver the Software Quality document for a major delivery.

3.2.1. Quality levels and number of deliveries

SQR-10 **The supplier shall associate a Quality Level for each module and justify their choices. This requirement encompasses all source code of the application, e.g. automatically generated code, legacy code, hand-written code and code produced by COTS**

SQR-20 **The supplier shall provide the number of software deliveries and a software quality plan**

The software quality plan shall consist of a table showing for each module:

- the corresponding Quality Level (QL-1 to QL-4);
- the number of times the module will be delivered during the project;
- the Software Quality Objective for every delivery of this module.

Notes:

- The manufacturer shall validate the software quality plan and the decisions taken within it.
- It is not mandatory to have the same SQO improvement from one delivery to one other for all modules.

The following example table shows the possible progressions for each quality level to achieve the final Software Quality Objectives. The number of quality levels is fixed. The minimum SQO level associated to the first, last and penultimate deliveries are also fixed. The number of deliveries (table lines) is project dependent.

Delivery	Quality Level			
	QL-1	QL-2	QL-3	QL-4
First	SQO-1	SQO-2	SQO-3	SQO-4
X Intermediates
X Intermediates	SQO-2	SQO-3	SQO-4	SQO-5
X Intermediates
Penultimate	SQO-3	SQO-4	SQO-5	SQO-6
Last	SQO-3	SQO-4	SQO-5	SQO-6

Each cell gives the lowest Software Quality Objective acceptable

As shown by the above table, different quality levels can correspond to different quality objectives. When a project is composed with several modules, each module can have a different criticality. The final SQO level can therefore be different from one module to one other. This final SQO level allows selecting the corresponding Quality Level for the module. The modules being delivered multiple times during the project, the selected Quality Level also defines the SQO level of each delivery of the module.

Example using the table above for a project composed of 3 Modules Module-1, Module-2 and Module-3 of different criticalities:

- Module-1 is considered very critical: its Quality Level is the highest, QL-4. The first delivery of Module-1 should be at least SQO-4 and the two last deliveries should be SQO-6.
- Module-2 is considered not critical: its Quality Level is the lowest, QL-1. The first delivery of Module-2 should be at least SQO-1 and the two last deliveries should be at least SQO-3.
- Module-3 is considered of medium criticality: it has been agreed at the beginning of the project to set its Quality Level to QL-3. The first delivery of Module-2 should be at least SQO-3 and the two last deliveries should be at least SQO-5.

The following table gives the Quality Level and the minimum SQO level to reach for the three modules at the end of the project:

Module	Quality Level	Minimum SQO level to reach (end of project)
Module 1	QL-4	SQO-6
Module 2	QL-1	SQO-3
Module 3	QL-3	SQO-5

SQR-30 **Once the quality levels for each module and the process to achieve them are defined by the supplier, the supplier shall justify all modifications to the plan**

3.2.2. People and tools

SQR-40 **The supplier shall provide information about people involved directly or indirectly in requirement fulfillment**

This shall contain at least:

- Company name
- Department and division name
- Geographical location
- Work done:
 - data generation (tool user), data computation, validation, ...
 - list of modules verified
 - list of requirements satisfied

SQR-50 **The supplier shall provide the list of tools and methods used**

This shall contain at least:

- Tool or method objectives
- Which requirements are affected and how
- Team responsible for the activity supported by the tool or method
- Tool or method experience, i.e. how long has the team been using the tools or methods and with what frequency. The frequency should be expressed as follows:
 - How many times has the team used these tools (or methods) per year?
 - Number of projects which used these tools or methods?

SQR-60 **The supplier shall inform and justify any modifications regarding the requirement SQR-50**

SQR-70 The supplier shall justify that methods and tools used are appropriate to achieve the requirements

This can be done, for example, by referencing the sections of the tools' documentation explaining what the tools provide, and how this helps to achieve the SQR.

SQR-80 Throughout the verification cycle the supplier shall maintain the same option set for tools used, to ease the comparison of different deliveries**3.2.3. Definitions of runtime errors**

In the rest of the document, the following definitions for runtime errors are used:

- A **systematic runtime error** is an operation which will generate an error for all executions of the application. It will typically not depend on the values of inputs of the application.
An example of a systematic runtime error is:

```
1: int foo (int a) {
2:   int b = 0;
3:   return (a / b); // Systematic Division by Zero
4: }
```

- A **potential runtime error** is an operation which will generate an error that may happen under certain circumstances, for example depending on the values of inputs of the application.
An example of a potential runtime error is:

```
1: int foo (int a) {
2:   int b = 4;
3:   return (b / a); // Potential Division by Zero
4: }
```

Note: the potential error line 3 cannot be proven safe or otherwise because the occurrence of the error depends on *a*, which is an input of the program.

- A **safe operation** is an operation (division, multiplication ...) which cannot produce a runtime error.
An example of a safe operation is:

```
1: int bar () {
2:   int x = 3;
3:   return (x);
4: }
5: int foo (int a) {
6:   int b = 4, c;
7:   c = bar ();
8:   b = a / (b - c); // Safe Division
9:   return (b / 2); // Safe Division
10: }
```

Note: the division line 8 can be proven safe by a tool or by a human review.
Indeed, $(b - c) = (4 - 3) = 1$ and is different from zero. No division by zero may occur.

- An **unreachable operation** is an operation (division, multiplication ...) which cannot be reached during the execution of the application.

An example of an unreachable operation is:

```
1: int bar () {
2:   int x = 3;
3:   return (x);
4: }
5: int foo (int a) {
6:   int b;
7:   b = bar ();
8:   if (b < 0)
9:     b = b / a;    // Unreachable operation
10:    return (b);
11: }
```

Note: line 9 cannot be reached because the condition $(b < 0)$ is always false. Therefore, no division by zero can occur on this line.

3.2.4. Standard comments and justifications

Some of the SQRs defined in this document require some operation to be concluded as proven safe or justified:

- Coding rules violations shall be corrected or justified (SQR-160 , SQR-180);
- Reviews of potential runtime errors should be performed with a defined review coverage¹ depending on the objective, and deviations should be corrected or justified (SQR-240 SQR-250 SQR-260).

For runtime errors the review coverage is defined by a percentage, indicated after the runtime error category (example: "Division by zero: 80%") which represents the number of operations concluded as proven safe or justified.

These conclusions could be performed:

- Automatically (with a tool);
- Partially automatically and completed manually;
- Totally manually.

Example: let's take an application containing 60 divisions. Let's assume that the review coverage objective is "Division by zero: 80%". Then the 80% review coverage can be reached by proving that at least 80% of the divisions are "safe operations" or "potential runtime errors" that can be justified.

Let's consider that a tool is used, which proves automatically that 45 divisions out of the 60 are "safe operations". The review objective can be reached by demonstrating that at least 3 "potential errors" can be justified, because $(45 + 3) / 60 = 80\%$.

To ease the justification review process:

- The **status** of the systematic and potential errors, and optionally the **next action** shall be provided;
- The **criticality** of the systematic and potential errors leading to the violation of a quality requirement shall be defined.

¹ Review coverage objectives are defined later in the document, in the paragraphs where corresponding SQR are described.

SQR-90 The supplier shall provide a normalized status for systematic and potential errors

This could be done by assigning a status to systematic and potential errors from the following list:

- **“Undecided”**: no status assigned yet. This status is used to explicitly defer the decision about the systematic or potential error;
- **“Investigate”**: the potential error should be investigated further. This status can be useful for example if the investigation has to be conducted by someone else later;
- **“Fix”**: the error must be fixed;
- **“Modify code/model”**: the code should be modified to make the systematic or potential error disappear. If the code is automatically generated from a model, the model should be modified. The status is different from **“Fix”** in that the reviewer may want the code to be modified even if the error does not lead to the violation of a quality requirement;
- **“Restart with different options”**: in the case of a potential error produced by a tool, this status can be used when the tool should be used with a different set of options or a different configuration;
- **“Justify with code/model annotations”**: this status may be used when the supplier wants the justification of the potential error to be persistent;
- **“No action planned”**: this status may be used when the supplier doesn’t plan to do any action with regards to the potential error. This status may be used in conjunction with a comment explaining why no action is planned.

Notes:

- A status of **“No action planned”** or **“Justify with code/model annotations”** leads to the potential error being justified in the sense of SQR-160 SQR-180 SQR-240 SQR-250 and SQR-260 .
- A status of **“Undecided”**, **“Investigate”**, **“Fix”**, **“Modify code/model”** or **“Restart with different options”** leads to the potential error being not justified in the sense of SQR-160 SQR-180 SQR-240 SQR-250 and SQR-260 .
- The supplier may add other statuses, and negotiate with the car manufacturer to determine if the statuses allow the justification or not of a potential error.

SQR-100 The supplier shall provide the criticality of systematic and potential errors leading to the violation of a quality requirement

A criticality shall be provided for all systematic and potential errors leading to the violation of a quality requirement:

- The possible criticalities are high, medium and low.
- The criticality shall at least be provided for systematic and potential errors with statuses **“Fix”** and **“Modify code/model”**.

3.3. Detailed design description

The information provided in this section will help evaluate the architecture of the application and its maturity. This will form the basis for the following sections of the document.

3.3.1. Application level

SQR-110 The supplier shall describe the architecture of the application

This shall contain at least:

- List of software modules
- How modules relate to one another
- Number of source files
- Number of header files

3.3.2. Module level

SQR-120 The supplier shall describe the structure of each module

This shall contain at least:

- List of source files used by each module
- List of header files with the file scope. Scope can be one of private, public or external.
 - 'Private' means used only by one module
 - 'Public' means used by several modules but developed internally
 - 'External' means header files provided by the operating system, compiler, device drivers or other header files which are not the intellectual property of the supplier.

3.3.3. File level

SQR-130 For each file the supplier shall provide information describing it

This shall contain at least:

- File (source and header) version based on the file management (revision control) system of the supplier.
N.B.: If the versioning is only managed at module level the supplier should only provide this information.
- Indicate the origin of each file, for example:
 - COTS
 - generated code
 - hand-written code
 - if other, give detailsN.B.: If the entire module has the same origin the information should be provided at the module level.
- Number of lines.

3.4. Code metrics

This paragraph shall help the automotive manufacturer evaluate the module characteristics and better understand the methods and tools used to demonstrate the application quality regarding the absence of runtime errors.

A recommended way is to provide the following metrics:

- Comment Density: relationship of the number of comments to the number of statements;
- Number of paths;
- Number of goto statements;
- Cyclomatic complexity “v(G)”;
- Number of Calling Functions per Function;
- Number of Called Functions per Function;
- Number of Function Parameters;
- Number of Instructions per Function;
- Number of call Levels;
- Number of return points within a function;
- Stability index: supplies a measure of the number of changes between two versions of a piece of software;
- Language Scope: indicator of the cost of maintaining/changing functions;
- Number of recursions.

Note 1: requirements related to comments in the code are optional on generated code and COTS.

Note 2: following metrics above would provide a high coverage of the metrics defined by the HIS¹ initiative.

SQR-140 The automotive manufacturer and the supplier shall choose at the beginning of the project the code metrics that will be used

SQR-150 For the chosen metrics, the supplier shall demonstrate that the modules comply with the agreed boundary limits, or justify the deviations

3.5. MISRA rules subsets

Two subsets of MISRA rules are defined. These two subsets correspond to different quality objectives; they are not to be used as different steps in reaching the final quality.

¹ **HIS: Hersteller Initiative Software.** Initiative from German automotive manufacturers (Audi, BMW Group, DaimlerChrysler, Porsche and Volkswagen) whose goal is the production of agreed standards within the area of standard software modules for networks, development of process maturity, software test, software tools and programming of ECU's. HIS specifies a fundamental set of Software Metrics to be used in the evaluation of software. See http://portal.automotive-his.de/images/pdf/SoftwareTest/his-sc-metriken.1.3.1_e.pdf

3.5.1. The first MISRA rules subset

Here is the first MISRA rules subset for C code, according to MISRA-C:2004.

A part of this subset is applicable for automatically generated code according to MISRA AC AGC.

An equivalent subset is applicable to C++ code, according to MISRA-C++:2008: for each C rule, the numbers of the closest C++ rules are provided in the last column of the table.

MISRA-C description	MISRA-C number	MISRA-C classif.	MISRA AC AGC	MISRA-C++ equivalent
Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier	5.2	required	OBL	2-10-2
The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	8.11	required	OBL	3-3-2
When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	8.12	required	OBL	3-1-3
Conversions shall not be performed between a pointer to object and any type other than integral type, another pointer to object type or a pointer to <i>void</i>	11.2	required	OBL	5-2-8
A cast should not be performed between a pointer type and an integral type	11.3	advisory	OBL	5-2-9
The underlying bit representations of floating-point values shall not be used	12.12	required	OBL	3-9-3
Floating-point expressions shall not be tested for equality or inequality	13.3	required		6-2-2
The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	13.4	required		6-5-1
The three expressions of a <i>for</i> statement shall be concerned only with loop control	13.5	required		6-5-2 6-5-3 6-5-4
The <i>goto</i> statement shall not be used.	14.4	required		6-6-1 6-6-2 6-6-4
A function shall have a single point of exit at the end of the function	14.7	required	OBL	6-6-5
Functions shall not be defined with a variable numbers of arguments	16.1	required	OBL	8-4-1
Functions shall not call themselves, either directly or indirectly	16.2	required	OBL	7-5-4
A pointer parameter in a function prototype should be declared as pointer to <i>const</i> if the pointer is not used to modify the addressed object	16.7	advisory		7-1-2
>, >=, <, <= shall not be applied to pointer types except where they point to the same array	17.3	required	OBL	5-0-18
Array indexing shall be the only allowed form of pointer arithmetic	17.4	required		5-0-15
The declaration of objects should contain no more than 2 levels of pointer indirection	17.5	advisory		5-0-19
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	17.6	required	OBL	7-5-1 7-5-2
An area of memory shall not be reused for unrelated purposes	18.3	required		N/A
Unions shall not be used	18.4	required	OBL	9-5-1
Dynamic heap memory allocation shall not be used	20.4	required		18-4-1

Software Quality Objectives

In case of C++ code, the above list is completed by the following subset of MISRA-C++:2008:

MISRA-C++ description	MISRA-C number	MISRA-C++ classif.	MISRA AC AGC	MISRA-C++ number
A base class shall only be declared virtual if it is used in a diamond hierarchy.	N/A	required	N/A	10-1-2
An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	N/A	required	N/A	10-1-3
There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	N/A	required	N/A	10-3-1
Each overriding virtual function shall be declared with the virtual keyword.	N/A	required	N/A	10-3-2
A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	N/A	required	N/A	10-3-3
Control shall not be transferred into a try or catch block using a goto or a switch statement.	N/A	required	N/A	15-0-3
An empty throw (throw;) shall only be used in the compound-statement of a catch handler.	N/A	required	N/A	15-1-3
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.	N/A	required	N/A	15-3-3
A class type exception shall always be caught by reference.	N/A	required	N/A	15-3-5
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.	N/A	required	N/A	15-3-6
Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	N/A	required	N/A	15-3-7
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.	N/A	required	N/A	15-4-1
A class destructor shall not exit with an exception.	N/A	required	N/A	15-5-1
Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	N/A	required	N/A	15-5-2

As a summary, the numbers of coding rules for the first subset are:

- 21 for C language
- 12 for automatically generated code
- 39 for C++ language

SQR-160 The supplier shall demonstrate that all the files within a module are compliant with the “first MISRA rules subset”. The supplier shall correct or justify all violations of the rules

The objective is to correct or justify all violations, i.e. zero remaining violations produced by the tool or remaining violations unjustified.

Software Quality Objectives

In this example 78.49% of MISRA-C:2004 rules listed in the first subset are in compliance or justified:

Rule	violation	commented
5.2	0	0
8.11	10	10
8.12	0	0
11.2	150	120
11.3	0	0
12.2	0	0
13.3	10	8
13.4	0	0
13.5	2	2
14.4	0	0
14.7	0	0
16.1	2	1
16.2	0	0
16.7	0	0
17.3	5	5
17.4	0	0
17.5	6	0
17.6	0	0
18.3	0	0
18.4	1	0
20.4	0	0
Total	186	146
	ratio	78,49%

SQR-170 Any modification of the first subset used shall be agreed between the supplier and the automotive manufacturer

3.5.2. The second MISRA rules subset

Here is the second MISRA rules subset for C code, according to MISRA-C:2004.

A part of this subset is applicable for automatically generated code according to MISRA AC AGC.

An equivalent subset is applicable to C++ code, according to MISRA-C++:2008: for each C rule, the numbers of the closest C++ rules are provided in the last column of the table.

MISRA-C description	MISRA-C number	MISRA-C classif.	MISRA AC AGC	MISRA-C++ equivalent
<i>typedefs</i> that indicate size and signedness should be used in place of the basic types	6.3	advisory	OBL	3-9-2
Objects shall be defined at block scope if they are only accessed from within a single function	8.7	required	OBL	3-4-1
Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures	9.2	required		8-5-2
In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised	9.3	required	OBL	8-5-3
The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression	10.3	required		5-0-7 5-0-8 5-0-9
If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	10.5	required		5-0-10

Software Quality Objectives

MISRA-C description	MISRA-C number	MISRA-C classif.	MISRA AC AGC	MISRA-C++ equivalent
Conversions shall not be performed between a pointer to a function and any type other than the integral type	11.1	required	OBL	5-2-6
A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer	11.5	required	OBL	5-2-5
Limited dependence should be placed on C's operator precedence rules in expressions	12.1	advisory		5-0-2
The value of an expression shall be the same under any order of evaluation that the standard permits	12.2	required	OBL	5-0-1
The operands of a logical && or shall be primary-expressions	12.5	required		5-2-1
The operands of a logical operators (&&, and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, and !)	12.6	advisory		4-5-1
The unary minus operator shall not be applied to an expression whose underlying type is unsigned	12.9	required	OBL	5-3-2
The comma operator shall not be used	12.10	required	OBL	5-18-1
Assignment operators shall not be used in expressions that yield a Boolean value	13.1	required		6-2-1
Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	13.2	advisory		5-0-13
Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop	13.6	required		6-5-3
The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement	14.8	required		6-3-1
All if ... else if constructs shall be terminated with an else clause	14.10	required		6-4-2
The final clause of a switch statement shall be the default clause	15.3	required		6-4-6
Identifiers shall be given for all of the parameters in a function prototype declaration	16.3	required	OBL	N/A
All exit paths from a function with non-void return type shall have an explicit return statement with an expression	16.8	required	OBL	8-4-3
A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty	16.9	required	OBL	8-4-4
C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct	19.4	required		16-2-2
Arguments to a function-like macro shall not contain tokens that look like pre-processing directives	19.9	required	OBL	16-0-5
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 ##	19.10	required	OBL	16-0-6
All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator	19.11	required	OBL	16-0-7
There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition	19.12	required	OBL	16-3-1
The validity of values passed to library functions shall be checked	20.3	required	OBL	N/A

Software Quality Objectives

In case of C++ code, the above list is completed by the following subset of MISRA-C++:2008:

MISRA-C++ description	MISRA-C number	MISRA-C++ classif.	MISRA AC AGC	MISRA-C++ number
A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code>	N/A	required	N/A	5-2-2
An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly	N/A	required	N/A	5-2-7
The comma operator, <code>&&</code> operator and the <code> </code> operator shall not be overloaded	N/A	required	N/A	5-2-11
The unary <code>&</code> operator shall not be overloaded	N/A	required	N/A	5-3-3
Member data in non- POD class types shall be private	N/A	required	N/A	11-0-1
An object's dynamic type shall not be used from the body of its constructor or destructor	N/A	required	N/A	12-1-1
The copy assignment operator shall be declared protected or private in an abstract class	N/A	required	N/A	12-8-2

As a summary, the numbers of coding rules for the first subset are:

- 29 for C language
- 16 for automatically generated code
- 36 for C++ language

SQR-180 The supplier shall demonstrate that all the files within a module are compliant with the “second MISRA rules subset”. The supplier shall correct or justify all violations of the rules

The objective is to correct or justify all violations, i.e. zero remaining violations produced by the tool or remaining violations unjustified.

SQR-190 Any modification of the second subset used shall be agreed between the supplier and the automotive manufacturer

3.6. Systematic runtime errors

SQR-200 The supplier shall demonstrate that for all files within a module a review of systematic runtime errors has been performed and that errors which have not been corrected are justified, for the following categories:

- Out-of-bounds array access
- Division by zero
- Read access to non-initialized data
- Function returning non initialized value
- Integer overflow/underflow
- Float overflow
- De-referencing through null or out-of-bounds pointer
- Usage (read or dereference) of a non-initialized pointer
- Shift amount in 0..31 (0..63) and left operand of left shift is negative
- Wrong type for argument passed to a function pointer
- Wrong number of arguments passed to a function pointer
- Wrong return type of a function or a function pointer
- Wrong return type of an arithmetic function
- Non null `this`-pointer (C++ application only)
- Positive array size (C++ application only)
- Incorrect `typeid` argument (C++ application only)
- Incorrect `dynamic_cast` on pointer (C++ application only)
- Incorrect `dynamic_cast` on reference (C++ application only)
- Invalid pointer to member (C++ application only)
- Call of pure virtual function (C++ application only)
- Incorrect type for `this`-pointer (C++ application only)

SQR-210 For each kind of runtime error the supplier shall justify the method and the process applied during the development phase to ensure the error's absence

3.7. Non terminating function calls and loops

SQR-220 The supplier shall justify the method and the process applied during the development phase to ensure the absence of non terminating calls and loops

Note: if the code intentionally contains:

- Non terminating loops like `'while(1)'` or `'for(;;)'`
- Non terminating of calls like `'exit'`, `'stop'`, `'My_Non_Returning_Function'`

these should be justified.

3.8. Unreachable branches

SQR-230 The supplier shall demonstrate that files do not contain any unjustified dead code branches

Note: all defensive code and dead code intentionally contained in the application shall be justified.

3.9. Potential runtime errors

SQR-240 The supplier shall demonstrate that for all files within a module, a review of potential runtime errors with review coverage level 1 (lowest) has been performed and that potential errors which have not been corrected are justified.

See second column in table below.

SQR-250 The supplier shall demonstrate that for all files within a module, a review of potential runtime errors with review coverage level 2 (medium) has been performed and that potential errors which have not been corrected are justified.

See third column in table below.

SQR-260 The supplier shall demonstrate that for all files within a module, a review of potential runtime errors with review coverage level 3 (highest) has been performed and that potential errors which have not been corrected are justified.

See last column in table below.

For each SQR, the supplier shall at least reach the following objectives identified in corresponding columns:

Potential runtime error	SQR-240	SQR-250	SQR-260
Out-of-bounds array access	80%	90%	100%
Division by zero	80%	90%	100%
Read access to local non-initialized data	80%	90%	100%
Read access to non local non-initialized data	60%	70%	80%
Function returning non initialized value	80%	90%	100%
Integer overflow/underflow	60%	80%	100%
Float overflow	60%	80%	100%
De-referencing through null or out-of-bounds pointers	60%	70%	80%
Usage (read or dereference) of a non-initialized pointer	60%	70%	80%
Shift amount in 0..31 (0.63) and left operand of left shift is negative	80%	90%	100%
Wrong type for argument passed to a function pointer	60%	80%	100%
Wrong number of arguments passed to a function pointer	60%	80%	100%
Wrong return type of a function or a function pointer	60%	80%	100%
Wrong return type for arithmetic functions	60%	80%	100%
Non null <code>this</code> -pointer (C++ application only)	50%	70%	90%
Positive array size (C++ application only)	50%	70%	90%
Incorrect <code>typeid</code> argument (C++ application only)	50%	70%	90%
Incorrect <code>dynamic cast</code> on pointer (C++ application only)	50%	70%	90%
Incorrect <code>dynamic cast</code> on reference (C++ application only)	50%	70%	90%
Invalid pointer to member (C++ application only)	50%	70%	90%
Call of pure virtual function (C++ application only)	50%	70%	90%
Incorrect type for <code>this</code> -pointer (C++ application only)	50%	70%	90%

3.10. Dataflow Analysis

SQR-270 The supplier shall provide for each module the data flow analysis results

This shall contain at least:

- Component call tree
- Dictionary containing read/write accesses to global variables
- List of shared variables and their associated concurrent access protection (if any)

4. REQUIREMENT MAPPING WITH ISO 26262-6:2011

4.1. Purpose and scope

This section describes a possible way to fulfill some ISO 26262 objectives at the software level.

It provides guidance for compliance with methods defined by ISO 26262. It describes how and in which detail to comply with ISO requirements, includes methods to verify code, and traces with ISO 26262 tables and paragraphs.

4.2. Summary

The table below summarizes the proposed mapped sections or tables of ISO 26262 with SQR groups. It also summarizes in the comment column what the SQO document can help achieving in context of the ISO 26262 verification process.

The mapping with the ISO standard can be of two different kinds:

- Mapping with tables: the SQR can trace with some specific methods of the tables.
- Mapping with sections: the SQR can help address part of the requirements of the sections.

The proposed requirements which the document helps to fulfill are the following:

- Avoid non-deterministic, implementation-defined, or undefined behavior.
- Achieve non functional requirement such as software robustness.

	<i>ISO 26262 section</i>	<i>ISO ref</i>	<i>SQR</i>	<i>Comment</i>
Mapping with tables	#5. Initiation of product development at the software level	• Table 1	• 140 to 190 • 270	The application of these SQRs can help support the correctness of the design and implementation
	#8. Software unit design and implementation	• Table 8 • Table 9	• 50 to 80 • 140 to 200 • 220 • 240 to 270	The application of the SQO document enables to comply with the requirements defined in table 8 and 9
Mapping with sections	#9. Software unit testing	• Section 9.4.3	• 200 to 270	The application of the SQO document enables to reduce unit tests, such as <ul style="list-style-type: none"> • absence of unintended functionality; • robustness; The example in §9.4.3.e mentions “absence of inaccessible software”, which is covered by SQR-230. The note b of Table 10 mentions the need to detect errors in case of “corrupting values of variables”. This is covered by SQR-200 to SQR-260 if full range is used for inputs of the application
	#10. Software integration and testing	• Section 10.4.3	• 230	The application of the SQO document enables to reduce integration and testing, such as <ul style="list-style-type: none"> • Robustness The example in §10.4.3.d mentions “absence of inaccessible software”, which is covered by SQR-230.

4.3. Details

Note 1: in this section, MISRA-C:2004 rules are provided as examples. The equivalent mapping is available for C++ by looking at the MISRA-C++:2008 corresponding rules in section 3.5.

Note 2: all SQRs from column “Related SQRs” shall be applied to cover the corresponding ISO 26262 Method or Section.

4.3.1. Section 5: Initiation of product development at the software level

Table 1 – Topics to be covered by modeling and coding guidelines

Methods		Related SQRs	Comments
1a	Enforcement of low complexity	140, 150	One purpose of code metrics and their boundaries is to ensure a low complexity of software
1b	Use of a language subset	160, 170, 180, 190	Both MISRA subsets defined in paragraph 3.5 contain rules ensuring the use of a subset of C and C++ languages. For example: 12.10, 13.1, 14.4, 18.4, ...
1c	Enforcement of strong typing	160, 170, 180, 190	Both MISRA subsets defined in paragraph 3.5 contain rules ensuring a strong typing in C and C++ languages. For example: 6.3, 8.12, 11.1, 16.7, 17.5, ...
1d	Use of defensive implementation techniques	160, 170, 180, 190	Second MISRA subset defined in paragraph 3.5 contain rules ensuring defensive programming in C and C++ languages. For example: 14.10, 15.3, ...
1e	Use of established design principles	160, 170, 180, 190	Both MISRA subsets defined in paragraph 3.5 contain rules ensuring the use of established design principles. For example: 8.7, 16.2, 16.8, 17.5, ...
1h	Use of naming conventions	270	The Data Dictionary enables to review naming conventions on global variables

4.3.2. Section 8: Software unit design and implementation

Table 8 – Design principle for software unit design and implementation

Methods		Related SQR	Comments
1a	One entry and one exit point in subprograms and functions	160, 170	MISRA rule 14.7
1b	No dynamic objects or variables, or else online test during their creation	160, 170	MISRA rule 20.4
1c	Initialization of variables	200, 240, 250, 260	This measure of the table can be traced with the runtime errors results, such as <ul style="list-style-type: none"> • “Read access to local non-initialized data” • “Read access to non local non-initialized data”
1d	No multiple use of variables names	160, 170	The enforcement of MISRA rule 5.2 help detect variables with same name in nested scopes
1e	Avoid global variables or else justify their usage	160, 170, 180, 190, 270	<ul style="list-style-type: none"> • The Data Dictionary containing read/write accesses to global variables provided during Dataflow analysis can help in implementing this rule • The enforcement of MISRA rules 8.11 and 8.7 can help detect variables whose scope should not be global
1f	Limited use of pointers	160, 170, 180, 190	Both MISRA subsets defined in paragraph 3.5 contain rules ensuring the use of established design principles. For example: 11.1, 11.2, 11.3, 11.5, 16.7, 17.3, 17.4, 17.5, 17.6, ...
		200, 240, 250, 260	This measure of the table can be traced with the runtime errors results, such as <ul style="list-style-type: none"> • “De-referencing through null or out-of-bounds pointer “ • “Usage (read or dereference) of a non-initialized pointer” • Function pointer with invalid dynamic arguments
1g	No implicit type conversions	160, 170, 180, 190	Both MISRA subsets defined in paragraph 3.5 contain rules ensuring the use of established design principles. For example: 11.1, 11.2, 11.3, 11.5
1h	No hidden data flow or control flow	160, 170	MISRA rule 5.2
1i	No unconditional jumps	160, 170	MISRA rule 14.4
1j	No recursions	140, 150, 160, 170	MISRA rule 16.2

Software Quality Objectives

Table 9 - Methods for the verification of software unit design and implementation

Methods		Related SQR	Comments
1a	Walk-through	200, 240	This measure of the table can be partially supported by output of SQRs 200 and 240: as soon as the list of potential runtime errors is produced, it can be walked through
1b	Inspection	200, 240, 250, 260, 270	This measure of the table can be partially supported by output of SQRs 200, 240 to 260 and 270. The code can be inspected with a highlight of systematic and potential runtime errors. Data Dictionary and Call Tree of the component can also be inspected to detect any issue.
1d	Formal verification	200, 240, 250, 260	Absence of runtime errors such as divisions by zero or overflows is an implicit specification of systems. Proving their absence is part of proving the correctness of a system against this implicit specification. This method is partially covered, because C and C++ are not formal notations.
1e	Control flow analysis	110-120	This measure of the table can be partially provided by output of SQR 110 – 120 (application level and file level description) Remark: as mentioned in note 'c' of Table 9, this requirement can be covered by using a method based on Formal verification ('1d') or Semantic code analysis ('1h')
1f	Data flow analysis	270	It is the aim of SQR 270 Remark: as mentioned in note 'c' of Table 9, this requirement can be covered by using a method based on Formal verification ('1d') or Semantic code analysis ('1h')
1g	Static code analysis	140, 160, 170, 180, 190	Complexity metrics and MISRA rules checking help to fulfill this measure of the table
1h	Semantic code analysis	70, 200, 240, 250, 260	Runtime error detection by Abstract Interpretation is a Semantic Code Analysis, as defined by the note 'd' of Table 9.

4.3.3. Section 9: Software unit testing

This section describes a possible way to fulfill the requirements defined in the section 9.4.3 of ISO 26262:

ISO unit testing objective ("shall be applied to demonstrate that the software units achieve: ...")	Related SQR	Comments
d. confidence in the absence of unintended functionality	230	Controlling the presence/absence of inaccessible pieces of code will help to detect the presence/absence of unintended functionality
	200, 240, 250, 260	Covering these SQRs help to reduce undefined behaviors which produce unintended functionalities. For example unknown values produced by components, due to data not initialized or out-of-bound array accesses. Functional tests (unit and integration) can then be focused on the functionality of the program
e. robustness	200, 240,	Analysis and correction or justification of real/potential run-

Software Quality Objectives

	250, 260	time errors will avoid some complex errors. It will help to ensure the production of a robust code. For example division by zero and dereferencing a NULL pointer variable.
--	----------	--

If SQR 200 to 260 are fulfilled, the applicant might consider focusing the software unit testing methods listed in tables 10, 11 and 12 on the other objectives defined in section 9.4.3. These objectives are listed below for information:

- a. Compliance with the software unit design specification
- b. Compliance with the specification of the hardware-software interface
- c. The specified functionality (complement to previous activity)
- f. Sufficient resources to support their functionality

4.3.4. **Section 10: Software integration and testing**

The application of the SQO document can help to demonstrate that the software components achieve the following objective defined in the section 10.4.3 of the ISO 26262 standard:

- d. robustness

Therefore the applicant might consider focusing the software integration testing methods listed in tables 13, 14 and 15 on the other objectives defined in section 10.4.3, if SQR 140 to 270 are fulfilled. These objectives are listed below for information

- a. compliance with the software architectural design
- b. compliance with the specification of the hardware-software interface
- c. the specified functionality
- e. sufficient resources to support the functionality

4.4. **Traceability SQO levels / ISO 26262 requirements**

To ease the reading of the traceability matrix next page, the SQRs associated with each SQO level are provided again in the following table:

SQO level	SQRs
SQO-1	10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170
SQO-2	Idem SQO-1, 200, 210, 220
SQO-3	Idem SQO-2, 230
SQO-4	Idem SQO-3, 240
SQO-5	Idem SQO-4, 180, 190, 250
SQO-6	Idem SQO-5, 260, 270

Software Quality Objectives

The following table lists which ISO 26262 requirements are covered for all SQO levels. It is also mentioned if the requirement is not, fully or partially covered.

ISO 26262 objective		SQO level required	No/Partial/Full coverage	
Table	Method			
Table 1	Topics to be covered by modeling and coding guidelines	1a Enforcement of low complexity	SQO-1	Full
		1b Use of language subsets	SQO-5	Full
		1c Enforcement of strong typing	SQO-5	Full
		1d Use of defensive implementation techniques	SQO-5	Partial
		1e Use of established design principles	SQO-5	Partial
		1f Use of unambiguous graphical representation	N/A	No
		1g Use of style guides	N/A	No
		1h Use of naming conventions	SQO-6	Partial
Table 2	<i>Notations for software architectural design</i>	N/A	No	
Table 3	<i>Principles for software architectural design</i>	N/A	No	
Table 4	<i>Mechanisms for error detection at the software architect. level</i>	N/A	No	
Table 5	<i>Mechanisms for error handling at the software architectural level</i>	N/A	No	
Table 6	<i>Methods for the verification of the software architectural design</i>	N/A	No	
Table 7	<i>Notations for software unit design</i>	N/A	No	
Table 8	Design principles for software unit design and implementation	1a One entry and one exit point in subprograms and functions	SQO-1	Full
		1b No dynamic objects or variables, or else online test during their creation	SQO-1	Partial
		1c Initialization of variables	SQO-6	Full
		1d No multiple use of variables names	SQO-1	Partial
		1e Avoid global variables or else justify their usage	SQO-6	Partial
		1f Limited use of pointers	SQO-6	Full
		1g No implicit type conversions	SQO-5	Full
		1h No hidden data flow or control flow	SQO-1	Partial
		1i No unconditional jumps	SQO-1	Full
		1j No recursions	SQO-1	Full
Table 9	Methods for the verification of software unit design and implementation	1a Walk-through	SQO-4	Partial
		1b Inspection	SQO-6	Partial
		1c <i>Semi-formal verification</i>	N/A	No
		1d Formal verification	SQO-6	Partial
		1e Control flow analysis	SQO-1	Partial
		1f Data flow analysis	SQO-6	Full
		1g Static code analysis	SQO-5	Full
		1h Semantic code analysis	SQO-6	Full
Table 10	<i>Methods for software unit testing</i>	N/A	No	
Table 11	<i>Methods for deriving test cases for software unit testing</i>	N/A	No	
Table 12	<i>Structural coverage metrics at the software unit level</i>	N/A	No	
Table 13	<i>Methods for software integration testing</i>	N/A	No	
Table 14	<i>Methods for deriving test cases for software integration testing</i>	N/A	No	
Table 15	<i>Structural coverage metrics at the software architectural level</i>	N/A	No	
Table 16	<i>Test environments for conducting the software safety requirements verification</i>	N/A	No	

COTS: Commercial, off-the-shelf is a term for software or hardware, generally technology or computer products, that are ready-made and available for sale, lease, or license to the general public. They are often used as alternatives to in-house developments or one-off government-funded developments. The use of COTS is being mandated across many government and business programs, as they may offer significant savings in procurement and maintenance. However, since COTS software specifications are written by external sources, government agencies are sometimes wary of these products because they fear that future changes to the product will not be under their control.

Manufacturer: is a company that uses a component made by a second company in its own product, or sells the product of the second company under its own brand. It constitutes a federally-licensed entity required to warrant and/or guarantee their products, unlike "aftermarket" which is not legally bound to a government-dictated level of liability.

HIS: Hersteller Initiative Software. Initiative from German automotive manufacturers (Audi, BMW Group, DaimlerChrysler, Porsche and Volkswagen) whose goal is the production of agreed standards within the area of standard software modules for networks, development of process maturity, software test, software tools and programming of ECU's.

HIS specifies a fundamental set of Software Metrics to be used in the evaluation of software.

See http://portal.automotive-his.de/images/pdf/SoftwareTest/his-sc-metriken.1.3.1_e.pdf

Supplier: automotive components manufacturer.

SQO: Software Quality Objectives

SQR: Software Quality Requirement