PRQA
Programming Research

WHITE PAPER

Achieving EN 50128 *Compliance with QA·C and QA·C++*

Jason Masters / Jill Britton
March 2015

This paper discusses the Functional Safety Standard EN 50128:2011 ("Railway applications - Communication, signaling and processing systems - Software for railway control and protection systems.").

First we explore how EN 50128 compares with other process standards, identifying some of the key differences and similarities. We then look specifically at the fit of PRQA's tools and how these can be deployed to help to comply with EN 50128.

Introduction

Electronic equipment is increasingly being used in safety critical environments, and the software used in these products is becoming more and more complex. Exhaustive testing to ensure that there is no situation in which a failure could occur is rarely possible and, therefore, systems must be designed in such a way to prevent failure or ensure controlled behavior if failures arise.

The introduction of standards has been an important factor in ensuring the development of robust software in safety critical applications. Coding standards such as MISRA, which mandate the use of a specific subset of a programming language, have been a major factor in the improvement of software quality. The international standard EN 50128 mandates the use of improved development processes, including the use of coding standards to encourage further gains in software quality.

This paper is split into two sections. The first discusses EN 50128 and how this compares to other process standards, highlighting some of the key differences and similarities. The second section looks in depth at how PRQA tools can be used to help to comply with EN 50128.

Section 1

Introduction to EN 50128

EN 50128 (Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems) provides a set of requirements with which the development, deployment and maintenance of any safety-related software intended for railway control and protection applications shall comply. It defines requirements concerning organizational structure, the relationship between organizations and the division of responsibility involved in the development, deployment and maintenance activities

This European Standard is part of a group of related standards. The others are EN 50126-1:1999 "*Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety(RAMS) – Part 1: Basic requirements and generic process*" and EN 50129:2003 "*Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling*".
EN 50126-1 addresses system issues on the widest scale, while EN 50129 addresses the approval process for individual systems which can exist within the overall railway control and protection systems.

Currently the systems included under EN 50128 include signaling, railway control and train protection. The intention is to extend the scope to incorporate the entire railway system, including rolling stock.

EN 50128 and other Safety Standards

The starting point for EN 50128 was IEC 61508 General electrical / programmable electronic devices), so there are considerable similarities between the two standards. However, EN 50128 is software specific unlike IEC 61508 which covers the whole system.

IEC 61508 is used in a variety of industries such as oil and gas, but also forms the basis of a range of other closely related industry sector specific standards including:

- ISO 26262 (Road vehicles - Functional Safety)
- IEC 62304 (Medical device software - Software life-cycle processes)
- IEC 60880 (Nuclear power plants - Instrumentation and control systems important to safety - Software aspects for computer-based systems performing category A functions)

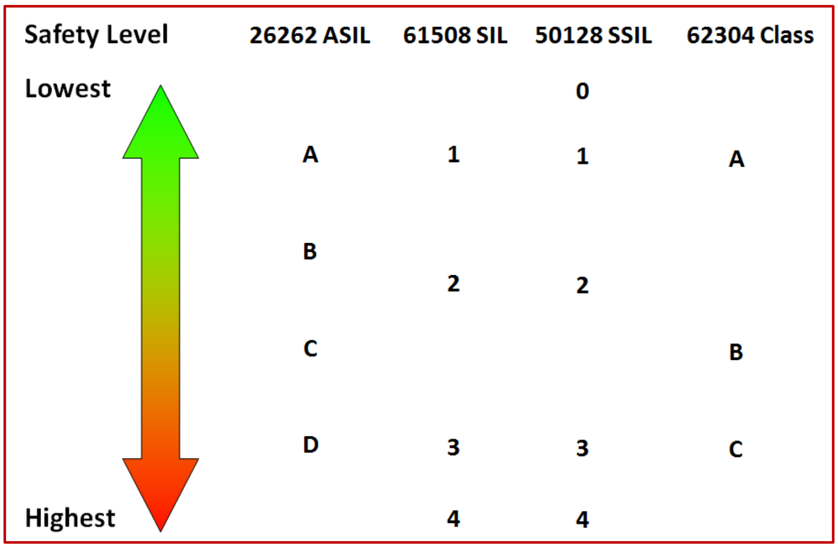
It is a worthwhile exercise to examine some of these to determine where they are similar, where they differ and especially *why* they differ.

SDLC

All the standards offer guidance on the core software development process. Most do not prescribe the use of a specific methodology. However, in practice, it is evident from the content of each standard which approach is being endorsed. Thus, EN 50128 process is based on Waterfall and V-model, whereas ISO 26262 describe the software lifecycle under a V-model framework. It is permissible to use Agile development for all standards, but the traceability and order of tasks within each sprint must be observed.

Safety Integrity Levels

Each standard provides a number of pre-defined safety level categories to which each system is assigned; with the higher safety systems requiring more checks and stringent controls. Although the logic is similar across all the standards each uses a slightly different terminology: Safety Integrity Level (SIL), Software Safety Integrity Level (SSIL), Automotive Safety Integrity Level (ASIL) and Classes. EN 50128, for example, defines five Software Safety Integrity Levels (SSIL), 0 through 4, where SSIL4 represents the highest level, and SSIL 0 represents the lowest level of safety integrity. The different terminologies used by each standard and their relative relationships are summarized below:



Safety Level	26262 ASIL	61508 SIL	50128 SSIL	62304 Class
Lowest			0	
	A	1	1	A
	B			
		2	2	
	C			B
	D	3	3	C
Highest		4	4	

Figure 1. Comparison of Safety Levels Between Different Safety Standards

It is important to recognize that each industry has its own distinctive characteristics and drivers, and that the creation of the safety level categories has to take account of these differences. For example:

- The use of medical devices tends to be controlled and operated directly by users. Risk assessments reflect the fact the actions and reactions of these users need be included and are a critical factor in the overall system. Risk mitigation, therefore, places a greater emphasis on end-user documentation and proving appropriate levels of feedback to the user.
- Generally electronic devices in a car are isolated from the driver who has no direct influence over how they function. However, there are situations where the driver clearly has a major role to play. Additionally, the capability of different drivers varies dramatically. Driver error is the most common cause of fatal car accidents [2] and most people accept a higher risk when travelling by car compared to by train or air. The ASIL levels are, therefore, determined by a combination of three factors – severity, probability and (driver) controllability.
- A railway network comprises a number of very large and complex, but tightly controlled systems. The number of operators / drivers is small, and they are trained to follow well defined and documented procedures. Whilst the overall probability of a malfunction might be lower, a single safety related failure can clearly have a very severe impact on multiple individuals / passengers.

Coding Standards

All the standards recognize the importance of coding standards, and in respect to complying with the most stringent SILs a coding standard is always Highly Recommended / Mandatory. However, it is worth noting that none of IEC 61508 family of standards explicitly states which coding standard to use. ISO 26262 comes closest, highlighting the MISRA C Coding Standards [1], but it stops short of mandating them. (Note that in practice MISRA is "de facto" within the automotive industry that any project opting not to use MISRA would raise eyebrows). MISRA is one of the longest established and most respected of coding standards, with the first revision, MISRA C:1998 "Guidelines for the Use of the C Language in Vehicle Based Software", published more than 17 years ago. Additionally it is also important to highlight the fact that MISRA has been adopted in every market that creates safety critical software – including rail. Indeed, the title of the most recent MISRA C:2012 standard "Guidelines for the use of C language in critical systems" [3] clearly signals the broader scope.

Tools

All modern software development is accompanied by a supporting cast of software tools, from modeling, compiling and debugging to testing and analyzing. With respect to these tools all the standards adopt a very similar approach. They recognize the fact that all tools are not equal, and they define, typically three, classes of tool which are ranked according of the potential impact on the software if the *tool* malfunctions. All the standards then define sets of criteria which much be met to ensure that the tools, within each class are, "fit for purpose".

Summary

Despite the differences between the standards, it is unlikely that a medical device developed using ISO 26262 or a railway device developed using IEC 62304 would be inherently unsafe or unusable, but it may not be as suitable as if it were developed using the appropriate standard. Additionally, it is not possible to say that one particular standard is worse or less stringent than another standard. However, tailoring the standard and processes to the industry reflects of the balance of risk / cost of risk mitigation appropriate to that industry.

Section 2

About PRQA, QA·C / QA·C++ and MISRA

PRQA pioneered coding standard inspection and is recognized worldwide as the coding standards expert due to its industry-leading software inspection and standards enforcement technology. PRQA's QA·C and QA·C++ static analysis tools offer two of the most comprehensive parsers available today, providing detailed information and accurately enforcing coding standards and best practices.

QA·C 8.1.2 with MISRA C (referred to as "QA·C")and QA·C++ 3.1 with an extended MISRA C++ (referred to as "QA·C++") have been certified by TÜV SAAR as fit for purpose to develop safety-related software up to SIL 4 according to EN 50128 when used as described in the Safety Manual. The MISRA C++ Extended Compliance module adds some additional rules over those in MISRA C++ to meet some of the standard's requirements.

EN 50128 – Classification of Tools

EN 50128 introduces three classes of tools; *T1*, *T2* and *T3* and all tools must be assigned to one of these classes depending on their potential to affect the executable code, as follows:

Tool Class	Description	Examples
T1	Tool output does not contribute to executable code	Text editor, VCS
T2	Tool tests / verifies design or executable code; cannot introduce defects into the executable code, but may fail to detect existing defects	Static analysis tool, Code coverage test tool
T3	Tool output contributes to executable code	Compiler, Linker

Class *T1* tools are relatively straight forward. They cannot introduce defects into the code, even if they malfunction. There is, therefore, no requirement to formally justify *T1* tools. By contrast, a malfunction in a Class *T3* tools can / will introduce defects directly into the compiled code. This can be due to bugs in the compiler or linker, or the developer using areas of the language where the compiler does something different to what was expected, or optimizations to remove defensive code that the programmer has added to protect against hardware faults.

The potential impact of Class *T2* tools is slightly more complex and subtle. Tools which test or verify code, such as static analysis tools, *cannot* themselves *introduce* a fault into the code; however, they can fail to detect existing faults. The significant risk with *T2* tools is that they create a false sense of confidence, with developers wrongly assuming that the tool will detect a specific type of defect and, therefore, not having any other mechanism in place to catch this defect. Note also that the failure of the tool to find a defect is not necessarily due to a deficiency in the tool, though the quality of tools can vary considerably [7]. Some defects are inherently undecidable, that is to say, there is not a general algorithm that can be used to detect all violations of a particular issue under all situations.

Class *T2* and *T3* tools must be justified – there must be evidence that the tools can meet the requirements demanded of them [3]. Additionally, tools in class *T2* and *T3* must be deployed in accordance with a ‘Safety Manual’ which ensures that the tool is installed, configured and operated correctly.

The manner of justification of *T2,* and *T3,* tools is not explicitly described in the EN 50128, just that ‘the tool must satisfy ‘the Software Requirements Specification at the required software safety integrity level.’[5]. Proof that this justification is valid can be performed by the tool vendors, with appropriate inspection being carried out by a recognized independent certification company. Note that it is also legitimate for the company developing the EN 50128 system to perform this exercise themselves. However, in practice this can be very time consuming, expensive and higher risk. It requires detailed knowledge of the exact issues that the tool claims to detect and numerous test cases to prove that it can detect these issues, while also noting the situations where it fails to detect an issue.

EN 50128 – Annex B and Annex D Compliance Tables – where the tools are used

Within the standard, Phase 7.5 (Software Component Implementation) together with Annex A (Criteria for the Selection of Techniques and Measures) address software development, placing requirements on the initiation of software development; software architectural design and software unit design and implementation. This is the main area where the PRQA tools are used; however, some of the information generated from the tools can be used to assist in later stages, particularly testing.

The following tables identify where the PRQA tools can be used to achieve the standard’s requirements. The tables consist of a number of techniques or measurements that must be used when developing the code. For each technique the requirements are stated for each SSIL, using the following abbreviations:

Abbreviation	Meaning
M	Mandatory
HR	Highly Recommended
R	Recommended
-	No requirement

Mandatory techniques and measures must be used. Highly Recommended techniques do not have to be followed; alternative techniques must be described, recorded and justified in the Software Quality Assurance Plan.

Not every technique and measure must be used at every SSIL. A subset of techniques can be used, the choice and combination depending on the SSIL.

Software Design and Implementation

The following tables are from the normative Annex A and show where the PRQA tools can be used to meet the required technique or measurement. Where a table is indented it is referred to from the previous table.

Table A. 4

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
4. Modular Approach	HR	M	M	M	M	Y	Y

Both QA·C and QA·C++ calculate a number of industry standard metrics that can be used to measure the modularity of a software component, for example, the number of parameters in a function definition. Thresholds can be set on all metrics, so developers can be notified during development that a function is exceeding these limits. Early detection and remedy of issues such as this can save much time later.

Decomposing code into small, functional modules is good design, but the downside is the increase in interfaces, which can provide an area where defects can arise. The PRQA tools can assist by identifying misuse of interfaces along with metrics, such as ‘number of function parameters’ and ‘number of functions calling this function’.

Table A. 4

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
5. Components	HR	HR	HR	HR	HR	Y	Y

Table A. 20

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
1. Information Hiding	-	-	-	-	-	Y	Y
2. Information Encapsulation	R	HR	HR	HR	HR	Y	Y
3. Parameter Number Limit	R	R	R	R	R	Y	Y

The PRQA tools analyze each Translation Unit (i.e. the result of preprocessing), which can be considered as ‘compile time’ analysis. Once all the translation units in a project have been analyzed, the tools can also perform Cross Module Analysis (CMA) which is ‘link time’ analysis.

In addition to identifying undefined behavior – something that must never exist in a safety critical system – the tools can pinpoint problems with inter module interfaces (APIs), class constructors, global variables, data visibility and encapsulation, as well as issues which may not directly cause the code to malfunction, but can cause confusion for developers. For example, it is entirely legal in C and C++ to use the same ‘typedef’ names in different modules for differing fundamental types. This can easily lead to confusion and potential problems when interfacing between the modules.

Table A. 4

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
6. Design and Coding Standards	HR	HR	HR	M	M	Y	Y

Table A. 12

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
1. Coding Standard	HR	HR	HR	M	M	Y	Y
2. Coding Style Guide	HR	HR	HR	HR	HR	Y	Y
3. No Dynamic Objects	-	R	R	HR	HR	Y	Y
4. No Dynamic Variables	-	R	R	HR	HR	Y	Y
5. Limited Use of Pointers	-	R	R	HR	HR	Y	Y
6. Limited Use of Recursion	-	R	R	HR	HR	Y	Y
7. No Unconditional Jumps	-	HR	HR	HR	HR	Y	Y
8. Limited size and complexity of Functions, Subroutines and Methods	HR	HR	HR	HR	HR	Y	Y
9. Entry / Exit Point strategy for Functions, Subroutines and Methods	R	HR	HR	HR	HR	Y	Y
10. Limited number of subroutine parameters	R	R	R	R	R	Y	Y
11. Limited use of Global Variables	HR	HR	HR	M	M	Y	Y

EN 50128 does not state which particular coding standard must be used, but ‘a coding standard’ must be used for SSIL 3 and 4 and is Highly Recommended for SIL 0, 1 and 2. While ‘Coding Standard’ is a rather general term, a robust coding standard should contain rules that:

- prevent the use of undefined or unspecified behavior
- prevent the programmer making common mistakes
- limit the use of certain constructs
- remove potential ambiguity
- restrict library usage

Most coding standards aim to subset the language to constrain developers to using only well defined areas of the language. Constructs which are hard to use correctly or may vary between implementations may be permitted, but with strict rules on how they can be used. The MISRA coding standard [8] meets this requirement.

Table A. 4

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
7. Analyzable Programs	HR	HR	HR	HR	HR	Y	Y

The MISRA coding standard suggests / requires the use of tools to enforce them and item 7 reinforces this. Programs should be written in a way that they can be easily analyzed. This usually means making the code more verbose and not deliberately obfusticating the code. C and C++ allow for very terse code and a good static analyzer (like a compiler) will understand this, but it is much harder to understand the tool’s output and identify the actual issue if the code is compact and badly formatted. There is a tendency for developers to try to do something ‘clever’ or use the latest patterns, but this is more error prone, possibly harder to test and debug, and almost certainly harder to maintain. Code should not be more complex than necessary: the system may be complex, but the idea is to remove *excessive* complexity. Simple, well laid out code will be quicker to analyze and be easier to work on and correct any defects.

Table A. 4

TECHNIQUE / MEASURE	SIL					PRQA tools	
	0	1	2	3	4	QA·C	QA·C++
8. Strongly Typed Programming Language	R	HR	HR	HR	HR	Y	Y

The C++ language is intrinsically more strongly typed than the C language, however, the rules for basic types in both languages are rather lax. Although there are types of differing sizes and signedness, converting between types is rarely illegal. In many cases the outcome is well defined, but can be unexpected. For example, converting a small negative signed number to an unsigned type of the same size will yield a large positive number, which can cause a program to behave in an unexpected way.

However, when using the MISRA subset, the dangers of such conversions are greatly reduced as the coding standard rules impose restrictions on when and what conversions care allowed. The essential type model [9] describes a type system which supports a stronger system of type checking while reducing the false positives that would arise due to some of the characteristics of the languages.

Table A. 4

TECHNIQUE / MEASURE	**SIL**					**PRQA tools**	
	0	**1**	**2**	**3**	**4**	**QA·C**	**QA·C++**
9. Structured Programming	R	HR	HR	HR	HR	Y	Y

Code can be poorly structured at the function, module and system level.

Poor system level structure is often typified by cyclic dependencies – Module A depends on Module B, which depends on Module C, which in turn depends on Module A. Not only does this affect the potential reuse of any of the modules, it makes it hard to test each module in isolation.

At the function level, excessive branching, knots (disjoints in the control flow) and control flow unrelated to function inputs result in poorly structured programs. They are hard to understand, test and maintain.

The PRQA tools can calculate many complexity metrics – for example the depth of nesting of conditions in a function and the number of knots in a function. These, together with a number of coding standard rules, can satisfy this measurement.

Table A. 4

TECHNIQUE / MEASURE	**SIL**					**PRQA tools**	
	0	**1**	**2**	**3**	**4**	**QA·C**	**QA·C++**
11. Language Subset	-	-	-	HR	HR	Y	Y

This is one of the primary aims of the MISRA coding standard: to constrain the developers to a safer subset of the language.

Verification and Testing

PRQA tools are primarily used during the coding phase, before testing. However, Table A.5 Verification and Testing contains two verification activities where the tools can be used:

Table A. 5

TECHNIQUE / MEASURE	**SIL**					**PRQA tools**	
	0	**1**	**2**	**3**	**4**	**QA·C**	**QA·C++**
2. Static Analysis	-	HR	HR	HR	HR	Y	Y

Table A. 19

TECHNIQUE / MEASURE	**SIL**					**PRQA tools**	
	0	**1**	**2**	**3**	**4**	**QA·C**	**QA·C++**
3. Control Flow Analysis	-	HR	HR	HR	HR	Y	Y
4. Data Flow Analysis	-	HR	HR	HR	HR	Y	Y

While static analysis is not Mandatory at any SSIL, it is the only practical way in which a coding standard (which is Mandatory for SSIL 3 and 4) can be enforced.

In this context the control flow analysis refers to structure without any reference to variable values – it only refers to the branching structure and not the actual execution structure. For example, in the following code snippet, it can be considered well structured.

```
void func(int x)
{
  if(x>10)
  {
      if (x == 5)
      {
      }
  }
}
```

Figure 2. Branching code with an unfeasible path

The essential cyclomatic complexity [10] of this code is 1 and any functions' control flow graph that can be reduced to 1 is said to be well structured. However, it also contains an unfeasible (syntactically reachable, but will never be reached) path, which can only be detected by examining the actual conditions.

In the context of EN 50128, data flow analysis refers to the general transformation of input data to output data or actions, rather than the flow through a function – it is a high level view where functions are considered as black boxes.

The PRQA tools contain a sophisticated dataflow engine which examines both control flow structures and data values, including those accessed via pointers, within functions and between functions in the same file. Identifying logical errors early in the development cycle can save a lot of time in the testing phase. The code above contains a logic error which probably means it does not produce the correct output.

Table A. 5

TECHNIQUE / MEASURE	**SIL**					**PRQA tools**	
	0	**1**	**2**	**3**	**4**	**QA·C**	**QA·C++**
4. Metrics	-	R	R	R	R	Y	Y

Table A.8

TECHNIQUE / MEASURE	**SIL**					**PRQA tools**	
	0	**1**	**2**	**3**	**4**	**QA·C**	**QA·C++**
1. Static Software Analysis	R	HR	HR	HR	HR	Y	Y

This technique refers to table A.19 as seen in the previous section.

This mentions a number of metrics, which can be used to predict the properties of a program, instead of looking at the development process or test results.

Although EN 50128 references metrics, it does not give exact details as to how they should be measured or give maximum limits for each metrics: it is up to each project to decide on what that limit should be. What is, perhaps, more useful is looking at the *trend* of metrics over different revisions of the source code. Looking at metrics from the start of the development as the code base grows can give insight to reveal areas of code which are accumulating too much complexity. This can predict potential issues in the code and point to areas that could be refactored.

Summary

EN 50128 is one of a family of process safety standards which is tailored for the particular demands of the railway industry. A system of Software Safety Integrity Levels ensures that the more safety critical systems require more rigorous checks and measures to help mitigate the risk and cost of software failure. QA·C and QA·C++ with the MISRA compliance modules have been certified for use with EN 50128 projects up to SSIL4. Thus the time and cost of meeting many of the standard's requirements associated with development at the software level can be reduced by using these tools.

References

1. Motor Industry Software Reliability Association: http://www.misra.org.uk/Publications/tabid/57/Default.aspx
2. ROSPA FAQs: http://www.rospa.com/faqs/detail.aspx?faq=298
3. MISRA C:2012 PRQA White Paper WP120A/02/13
4. EN 50128 section 6.7.4.2.
5. EN 50128 section 7.3.4.12.
6. Using Static Analysis and Continuous Integration to Ensure Code Quality PRQA White Paper PRQAA136
7. Synopsis of "Comparative Study of MISRA-C Compliancy Checking Tools". PRQA White Paper WP128B/01/13
8. MISRA: An Overview. PRQA White Paper WP116B/03/12
9. The essential type model, section 8.10 MISRA-C:2012, ISBN 978-1-906400-11-8
10. McCabe, T. J. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

About PRQA

Established in 1985, PRQA is recognized throughout the industry as a pioneer in static analysis, championing automated coding standard inspection and defect detection, delivering its expertise through industry-leading software inspection and standards enforcement technology.

PRQA static analysis tools, QA·C and QA·C++, are at the forefront in delivering MISRA C and MISRA C++ compliance checking as well as a host of other valuable analysis capabilities. All contain powerful, proprietary parsing engines combined with deep accurate dataflow which deliver high fidelity language analysis and comprehension. They identify problems caused by language usage that is dangerous, overly complex, non-portable or difficult to maintain. Additionally, they provide a mechanism for coding standard enforcement.

Contact Us

PRQA has offices globally and offers worldwide customer support. Visit our website to find details of your local representative.

Email: info@programmingresearch.com
Web: www.programmingresearch.com