

AUTOSAR and the LDRA tool suite®

A more complete solution

www.ldra.com

* Registration required to download the document

© LDRA Ltd. This document is property of LDRA Ltd. Its contents cannot be reproduced, disclosed or utilised without company approval.

Background.....	3
AUTOSAR and ISO 26262.....	3
AUTOSAR Classic Platform: The Premise.....	4
Working with the AUTOSAR Classic Platform.....	5
AUTOSAR Classic Platform Application Development.....	5
AUTOSAR Classic Platform, ISO 26262, and the LDRA tool suite.....	6
Software Unit Design and Implementation (ISO 26262-6:2011 section 8).....	6
Software Unit Testing (ISO 26262-6:2011 section 9) and Software Integration and Testing (ISO 26262-6:2011 section 10).....	9
The AUTOSAR Adaptive Platform: Adapting to a Changing World.....	11
Working with the AUTOSAR Adaptive Platform.....	11
POSIX Compliance.....	12
AUTOSAR C++ Coding Guidelines.....	13
AUTOSAR C++ and Other Standards.....	14
AUTOSAR C++ Coding Rules Classification.....	14
The LDRA tool suite and AUTOSAR C++14.....	15
Example AUTOSAR C++ Rules and Violations.....	16
Rule A10-1-1: Addressing the “Diamond Problem”.....	16
Rule A27-0-1: AUTOSAR C++ and Security.....	16
Rule M8-5-2: Braced Initialization.....	17
Rule A4-10-1: Use of <i>nullptr</i>	18
Rule A7-2-3: Use of Scoped Enumeration.....	18
AUTOSAR C++ and ISO 26262.....	18
Example AUTOSAR C++ Rules and Violations Relating to Coding Metrics.....	19
Rule A1-4-2: Use of Defined Boundaries.....	19
A More Complete Solution.....	20
Bi-directional Traceability (ISO 26262-4:2011 and ISO 26262-6:2011).....	20
Conclusions.....	21
Works Cited.....	21
Works Cited (Cont.).....	22

Background

AUTOSAR (Automotive Open System Architecture) is a standardization initiative of a group of leading automotive OEMs and suppliers, and was founded in autumn 2003. Its ongoing mission is to develop a reference architecture for ECU software, which can overcome the growing complexity of software in modern vehicles.

In December 2017, the AUTOSAR family of standards expanded to embrace the new “Adaptive Platform,” with the existing development branch now known as the “Classic Platform” (Figure 1).

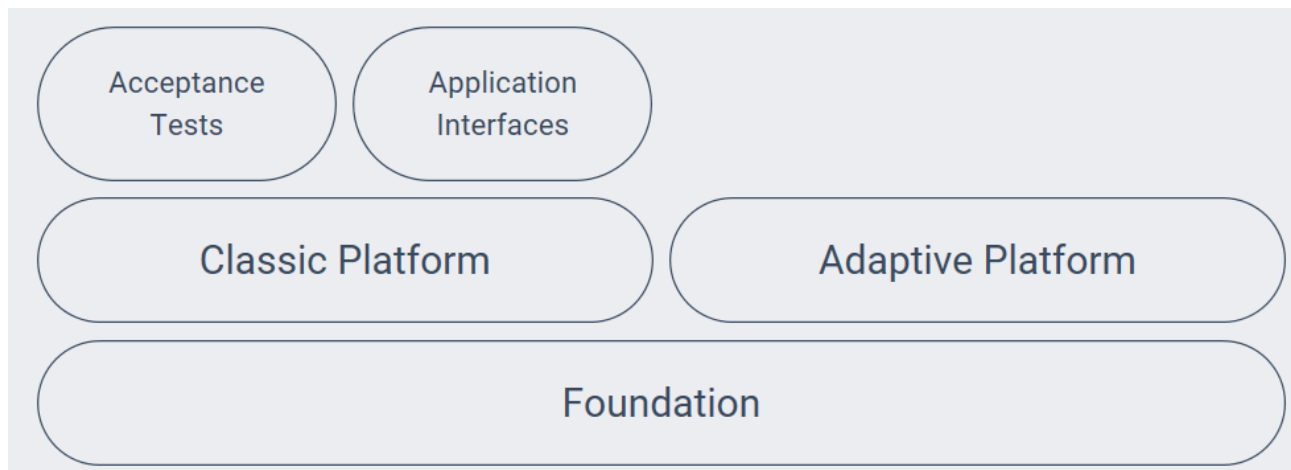


Figure 1: The AUTOSAR family of standards

The Classic Platform¹ is AUTOSAR’s established solution for embedded systems with hard real-time and safety constraints, first published in 2005.

The Adaptive Platform² is AUTOSAR’s solution for high-performance computing ECUs to build safety-related systems for use cases such as highly automated and autonomous driving. At the time of its announcement, former AUTOSAR spokesperson Thomas Rüping said that “New requirements call for new solutions... AUTOSAR also aims to provide an optimal standardized software framework for new applications in the fields of connectivity and highly automated and autonomous driving.”³ It is, then, complementary to the AUTOSAR Classic Platform, and not a replacement for it.

The purpose of the Foundation⁴ standard is to enforce interoperability between the AUTOSAR platforms. It contains requirements and technical specifications common to the AUTOSAR platforms – for example, communications protocols.

The documentation associated with the AUTOSAR standards extends into many volumes, and this document cannot reference them all. It focuses instead on how an integrated and comprehensive set of tools such as the LDRA tool suite[®] can help ease the path to compliance.

AUTOSAR and ISO 26262

Adherence to either AUTOSAR platform standard does not in itself imply compliance with the functional safety standard “ISO 26262 Road vehicles — Functional safety,”⁵ and for that reason the AUTOSAR and ISO standards need to be considered concurrently.

¹ AUTOSAR Classic Platform <https://www.autosar.org/standards/classic-platform/>

² AUTOSAR Adaptive Platform <https://www.autosar.org/standards/adaptive-platform/>

³ http://www.ai-online.com/Adv/Previous/show_issue.php?id=6881#sthash.Cvo3ABXn.oxLSMTMG.dpbs

⁴ AUTOSAR Foundation <https://www.autosar.org/standards/adaptive-platform/>

⁵ International standard ISO 26262 Road vehicles — Functional Safety

ISO 26262:2011 “Road vehicles – Functional safety” was published in response to an explosion in automotive E/E/PE system complexity, and the associated risks to public safety. Like the rail, medical device and process industries before it, the automotive sector based their functional standard on the industry agnostic functional safety standard IEC 61508⁶. The resulting ISO 26262 has become the dominant automotive functional safety standard, and its requirements and processes are becoming increasingly familiar throughout the industry.

ISO 26262:2011 consists of 10 parts with three focused on product development: system level (Part 4)⁷, hardware level (Part 5)⁸, and software level (Part 6)⁹. ISO 26262-6:2011 provides detailed industry specific guidelines for the production of all software for automotive systems and equipment, whether it is safety critical or not.

The relationship between the system-wide ISO 26262-4:2011 and the software specific sub-phases found in ISO 26262-6:2011 can be represented in a V-model. Figure 2 shows how LDRA and other complementary tools can be used throughout the development lifecycle.

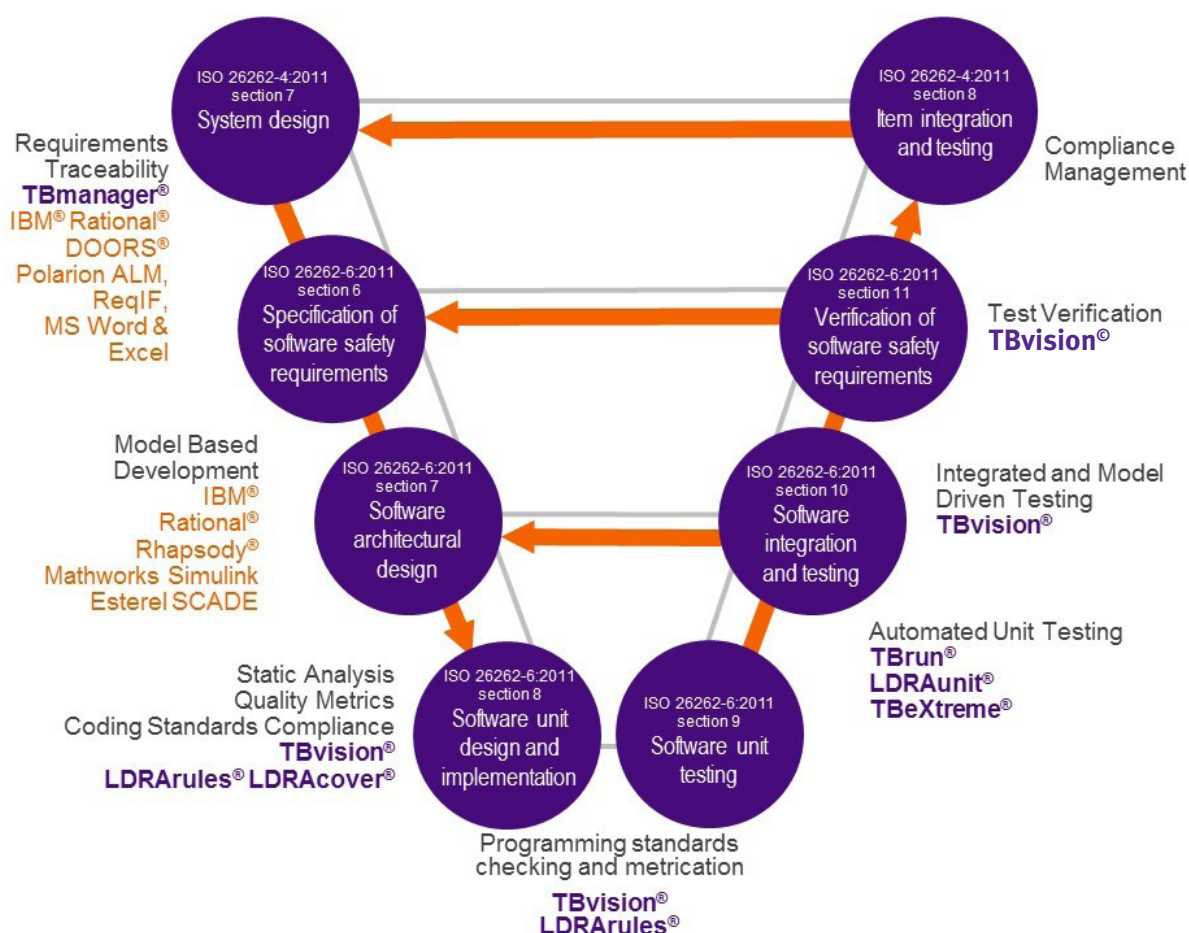


Figure 2: Software-development V-model with cross-references to ISO 26262 and standard development tools

AUTOSAR Classic Platform: The Premise

Traditionally, software development for automotive applications was completed in isolation by OEMs or their suppliers. This led to a lot of duplicated effort amongst players in the automotive sector, prolonged development cycles, and a lack of commercial flexibility especially when updates to either hardware or software were required.

⁶ IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems

⁷ ISO 26262-4:2011 Road vehicles -- Functional safety -- Part 4: Product development at the system level

⁸ ISO 26262-5:2011 Road vehicles -- Functional safety -- Part 5: Product development at the hardware level

⁹ ISO 26262-6:2011 Road vehicles -- Functional safety -- Part 6: Product development at the software level

The AUTOSAR Classic Platform looks to address those deficiencies by providing a clearly defined layer of abstraction between the hardware, providing a common software foundation irrespective of the chosen microcontroller (Figure 3).

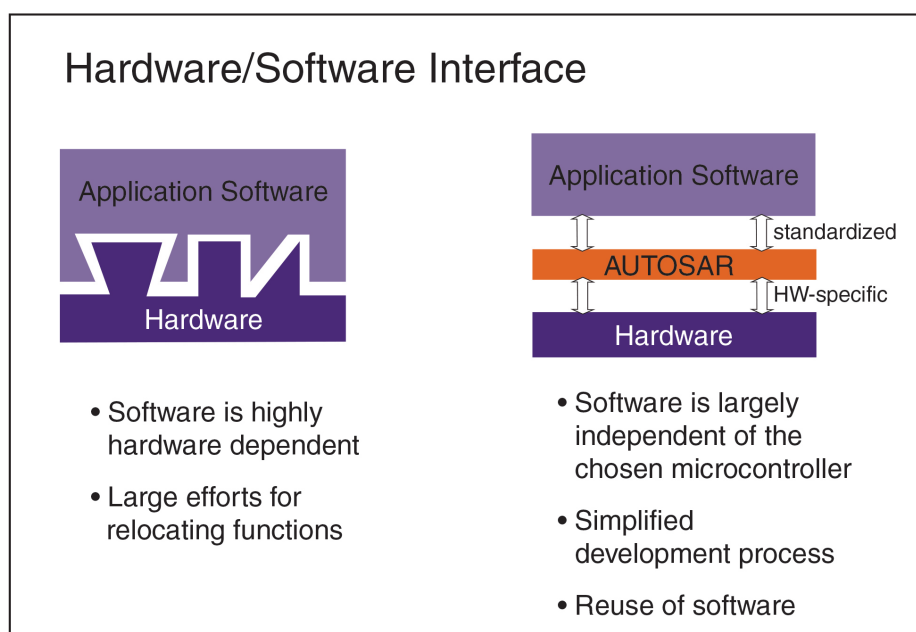


Figure 3: The basic premise of the AUTOSAR Classic Platform¹⁰

Working with the AUTOSAR Classic Platform

AUTOSAR Classic Platform Application Development

Consider an example system consisting of 3 interconnecting software units (Figure 4). Each software component encapsulates a part of the application, and the components communicate with each other over a hardware independent “Virtual Function Bus.”

Each ECU can host one or more software units, and each hosts a runtime environment to implement the Virtual Functional Bus and the AUTOSAR Basic Software.

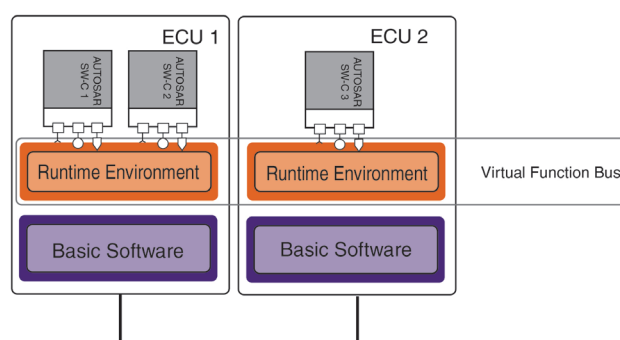


Figure 4: Physical allocation of software units on ECUs

This BASIC software provides service functions to the application developer such as operating system services, network communication and management services, memory services, and diagnostic services. In this way, the hardware environment is abstracted away from the software units, and so is largely transparent to the application developers.

¹⁰ AUTOSAR – A First Glance <https://www.youtube.com/watch?v=F27jtKkxbAo>

The BASIC software is subdivided into layers (Figure 5):

- The Microcontroller Abstraction Layer (MCAL), providing drivers for access to memory, communication and input/output (I/O) of the microcontroller;
- The ECU Abstraction Layer (ECUAL), providing uniform access to all functionalities of an ECU such as communication, memory or I/O, irrespective of where these functionalities are implemented physically;
- The Service Layer, providing both the operating system for the application layer, and background services such as network services, memory management and bus communication;
- The Runtime Environment (RTE), abstracting the application layer from the basic software, controlling the runtime behaviour of the application layer, and implementing the data exchange.

In the application layer, the application functionality of the ECU is implemented in the form of individual AUTOSAR software components (SWCs).

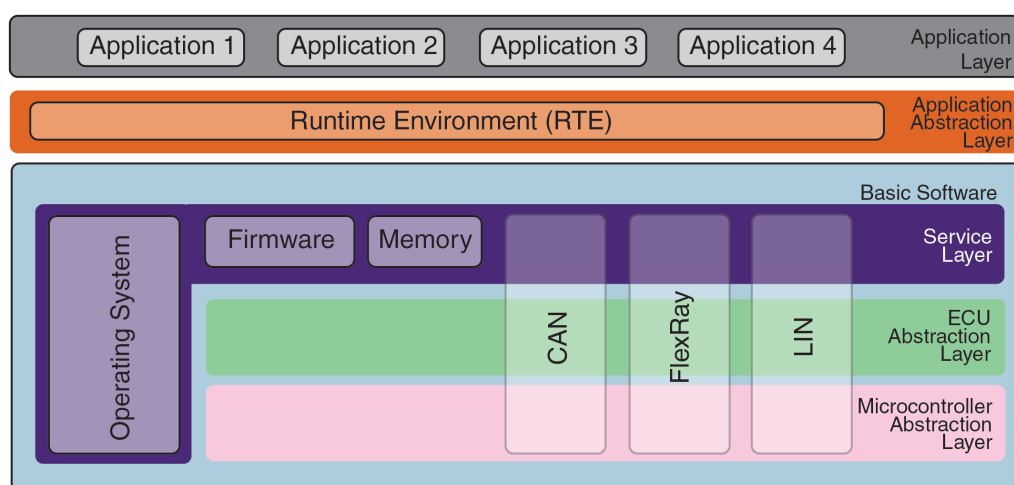


Figure 5: The AUTOSAR Classic Platform Architecture

In practice, this architecture provides a level of abstraction such that these AUTOSAR software components (SWCs) are transferable and can be deployed to ECUs very late in the development process¹¹. AUTOSAR component dependencies are described in form of interfaces and ports with reference to the Virtual Function Bus. No internal, hidden dependencies are permitted.

The net result is that any components implementing the same logic and providing the same public communication interfaces to the remaining system should be exchangeable. All such components are said to be of the same Component Type, and each Component type is defined in accordance with the Software Component Template¹² - itself a document over 800 pages long.

AUTOSAR Classic Platform, ISO 26262, and the LDRA tool suite

With reference to the ISO 26262 V-model (Figure 2) and to the elements of it specifically impacted by AUTOSAR, it is useful to consider the development lifecycle starting from a completed Software Architectural Design, focusing on the subsequent work relating to Software Unit Design and Implementation, Software Unit Testing, and Software Integration and Testing.

Software Unit Design and Implementation (ISO 26262-6:2011 section 8)

The illustration in Figure 6 is a typical example of a table from ISO 26262-6:2011. It shows the coding and modelling guidelines to be enforced during implementation, superimposed with an indication of where compliance can be confirmed using automated tools.

¹¹ An introduction to AUTOSAR https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf

¹² AUTOSAR Software Component Template https://www.autosar.org/fileadmin/user_upload/standards/classic/4-1/AUTOSAR_TPS_SoftwareComponentTemplate.pdf

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity	++✓	++✓	++✓	++✓
1b	Use of language subsets	++✓	++✓	++✓	++✓
1c	Enforcement of strong typing	++✓	++✓	++✓	++✓
1d	Use of defensive implementation techniques	O	+	++	++
1e	Use of established design principles	+✓	+✓	+✓	++✓
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+✓	++✓	++✓	++✓
1h	Use of naming conventions	++✓	++✓	++✓	++✓
"++" The method is highly recommended for this ASIL. "+" The method is recommended for this ASIL. "O" The method has no recommendation for or against its usage for this ASIL. ✓ Satisfied by the LDRA tool suite					

Figure 6: Mapping the capabilities of the LDRA tool suite to “Table 6: Methods for the verification of the software architectural design” specified by ISO 26262-6:2011¹³

These guidelines combine to make the resulting code more reliable, less prone to error, easier to test, and/or easier to maintain. Peer reviews represent a traditional approach to enforcing adherence to such guidelines, and while they still have an important part to play, automating the more tedious checks using tools is far more efficient, less prone to error, repeatable, and demonstrable. Figure 7 shows just one example of how language subset violations can be presented.

▼ TunnelData::Cell::Cell				
▼ ❖ Float/integer conversion without cast.	Required	435 S	MISRA-C++:2008 5-0-5	
❖ Float/integer conversion without cast. : (double and int): f	Required	435 S	MISRA-C++:2008 5-0-5	
❖ Float/integer conversion without cast. : (double and int): f < NumLampTypes	Required	435 S	MISRA-C++:2008 5-0-5	
❖ Pointer subtraction not addressing one array.	Required	438 S	MISRA-C++:2008 5-0-17	
❖ Cast to an unrelated type. : (double* to int*): (Sint_32 *) p_f	Required	554 S	MISRA-C++:2008 3-9-3,5-2-7	
❖ Casting operation on a pointer. : (double* to int*): (Sint_32 *) p_f			MISRA-C++:2008 5-2-7	
❖ Use of C type cast.			MISRA-C++:2008 5-2-4	
❖ Casting operation to a pointer. : (double* to int*): (Sint_32 *) p_f	Required	554 S	MISRA-C++:2008 5-2-7	

Standards Violation

Standards Violation

Figure 7: Highlighting violated coding guidelines in the LDRA tool suite

ISO 26262-6:2011 references the MISRA¹⁴ language subset as an example of what might be used. It is entirely permissible to use an in-house set or to manipulate, adjust and add to such a standard set, to make it more appropriate for a particular application (Figure 8).

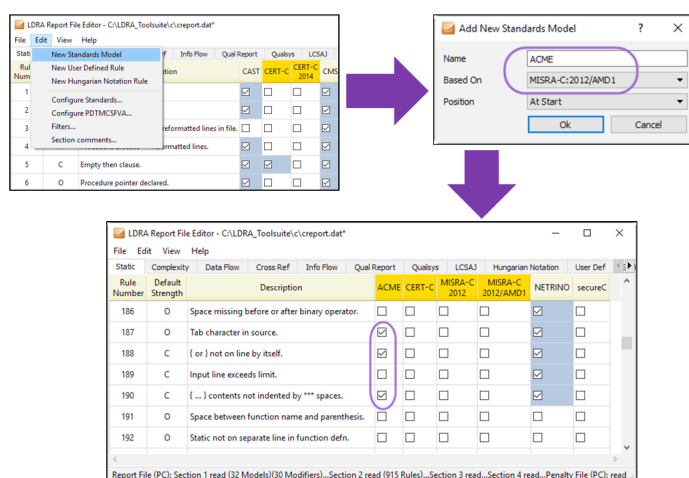


Figure 8: Basing a new standards model on an existing one, using the LDRA tool suite

¹³ Based on table 6 from ISO 26262-6:2011, Copyright© 2015 IEC, Geneva, Switzerland. All rights acknowledged

¹⁴ MISRA – The Motor Industry Software Reliability Association <https://www.misra.org.uk/>

Anyone working with both the AUTOSAR Classic Platform and MISRA standards has a second challenge to overcome, in that the standard format of AUTOSAR system calls are not MISRA compliant. The LDRA tool suite has been configured to recognise AUTOSAR system calls, and not to highlight them as MISRA related violations.

Software Architectural Design and Unit Implementation

Establishing appropriate project guidelines for coding, architectural design and unit implementation are clearly three discrete tasks but software developers responsible for implementing the design need to be mindful of them all concurrently.

These guidelines are also founded on the notion that they make the resulting code more reliable, less prone to error, easier to test and/or easier to maintain. For example, architectural guidelines include:

- **Restricted size of software components** and **Restricted size of interfaces** are recommended not least because large, rambling functions are difficult to read, maintain, and test – and hence more susceptible to error.
- **High cohesion within each software component.** High cohesion results from the close linking between the modules of a software program, which in turn impacts on how rapidly it can perform the different tasks assigned to it.

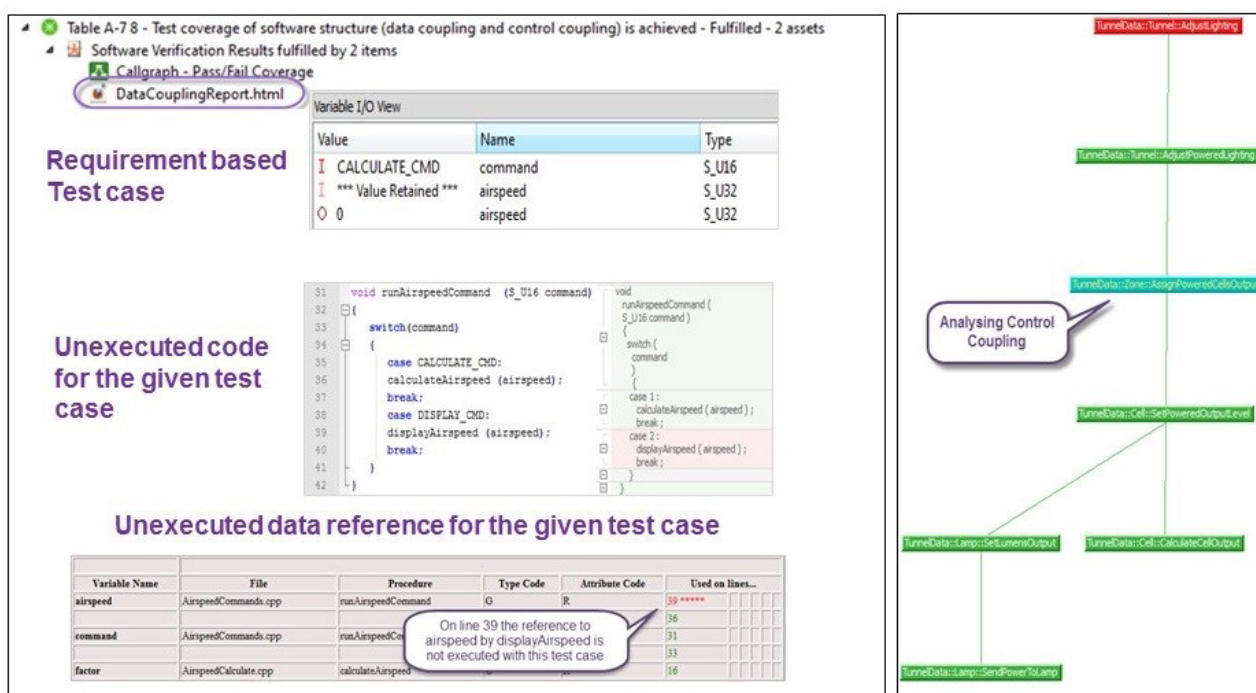


Figure 9: Output from control and data coupling analysis as represented in the LDRA tool suite

Static analysis tools can provide metrics to ensure compliance with the standard such as complexity metrics as a product of interface analysis, cohesion metrics evaluated through data object analysis, and coupling metrics via data and control coupling analysis (Figure 9).

More generally, static analysis can help to ensure that the good practices required by ISO 26262:2011 are adhered to whether they are coding rules, design principles, or principles for software architectural design.

In practice, for developers who are newcomers to ISO 26262, the role of such a tool often evolves from a mechanism for highlighting violations, to a means to confirm that there are none.

Software Unit Testing (ISO 26262-6:2011 section 9) and Software Integration and Testing (ISO 26262-6:2011 section 10)

Just as static analysis techniques (an automated “inspection” of the source code) are applicable across the sub-phases of coding, architectural design and unit implementation, dynamic analysis techniques (involving the execution of some or all of the code) are applicable to unit, integration and system testing. Unit testing is designed to focus on particular software procedures or functions in isolation, whereas integration testing ensures that safety and functional requirements are met when units are working together in accordance with the software architectural design.

ISO 26262-6:2011 tables list techniques and metrics for performing unit and integration tests on target hardware to ensure that the safety and functional requirements are met and software interfaces are verified. Fault injection and resource tests further prove robustness and resilience and, where applicable, back-to-back testing of model and code helps to prove the correct interpretation of the design. Artefacts associated with these techniques provide both reference for their management, and evidence of their completion. They include the software unit design specification, test procedures, verification plan and verification specification. On completing each test procedure, pass/fail results are reported and compliance with requirements verified appropriately.

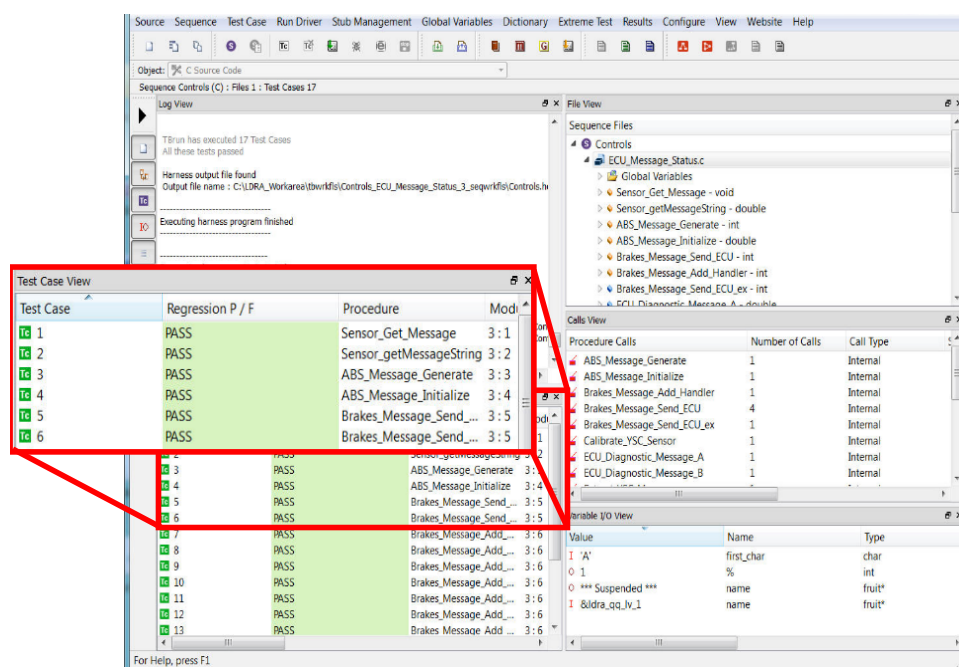


Figure 10: Performing requirement based unit-testing using the LDRA tool suite

The example in Figure 10 shows how the software interface is exposed at the function scope allowing the user to enter inputs and expected outputs to form the basis of a test harness. The harness is then compiled and executed on the target hardware, and actual and expected outputs compared.

Unit tests become integration tests as units are introduced as part of a call tree, rather than being “stubbed.” Exactly the same test data can be used to validate the code in both cases.

Equivalence boundary values such as minimum value, value below lower partition value, lower partition value, upper partition value and value above upper partition boundary can be analysed by automatically generating a series of unit test cases, complete with associated input data.

Should changes become necessary – perhaps as a result of a failed test, or in response to a requirement change from a customer - then all impacted unit and integration tests would need to be re-run (regression tested), automatically re-applying those tests through the tool to ensure that the changes do not compromise any established functionality.

ISO 26262:2011 does not require that any of the tests it promotes deploy software test tools. However, just as for static analysis, dynamic analysis tools help to make the test process far more efficient, especially for substantial projects.

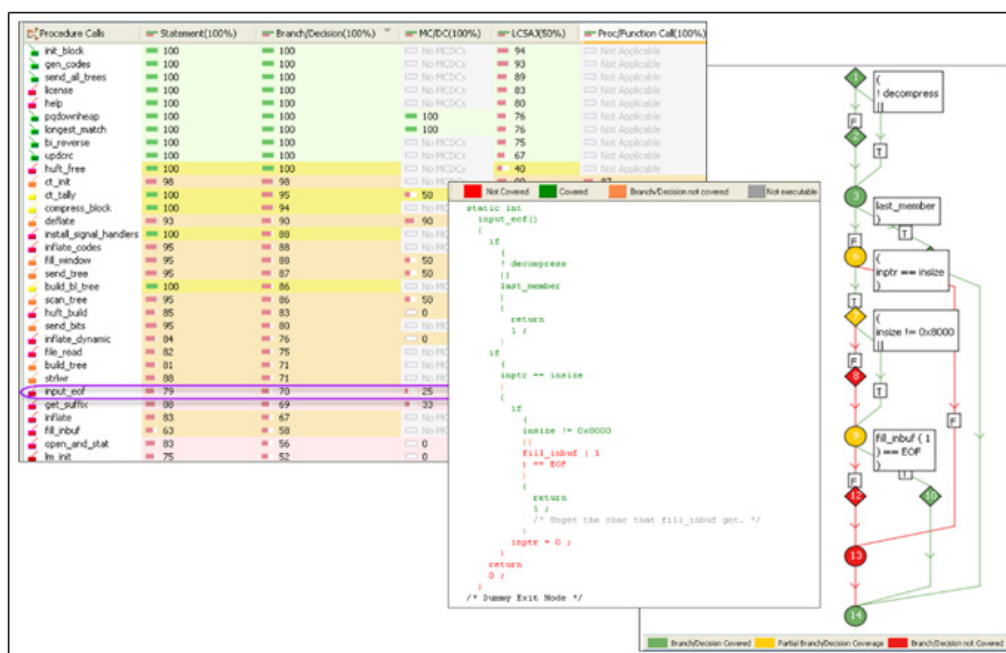


Figure 11: Examples of representations of structural coverage within the LDRA tool suite

Structural Coverage Metrics

In addition to showing that the software functions correctly, dynamic analysis is used to generate structural coverage metrics. In conjunction with the coverage of requirements at the software unit level, these metrics provide the necessary data to evaluate the completeness of test cases and to demonstrate that there is no unintended functionality (Figure 11).

Metrics recommended by ISO 26262:2011 include functional, call, statement, branch and MC/DC coverage. Unit and system test facilities can operate in tandem, so that (for instance) coverage data can be generated for most of the source code through a dynamic system test, and then be complemented using unit tests to exercise such as defensive constructs which are inaccessible during normal system operation.

Software Test and Model Based Development

These static and dynamic facilities can be integrated with several different model based development tools, such as IBM Rational Rhapsody¹⁵, MathWorks Simulink¹⁶ and Esterel SCAD¹⁷. The development phase itself involves the creation of the model in the usual way, with the integration becoming more pertinent once source code has been auto generated from that model.

Figure 12 illustrates how an integration with Simulink can be deployed using an approach appropriate for use with “Back-to-back” testing. Design models are developed with Simulink and verified with Simulink tests. Then, code is generated from Simulink, instrumented by the LDRA tool suite, and then executed in Software In the Loop (SIL or host), or Processor In the Loop (PIL or target) mode. Structural coverage is then collected and structural coverage reports can be generated at the source code level by Simulink and by source code dynamic analysis in tandem.

¹⁵ <http://www-03.ibm.com/software/products/en/ratirhapfam/>

¹⁶ <https://www.mathworks.com/products/simulink.html>

¹⁷ <http://www.esterel-technologies.com/products/scade-suite/>

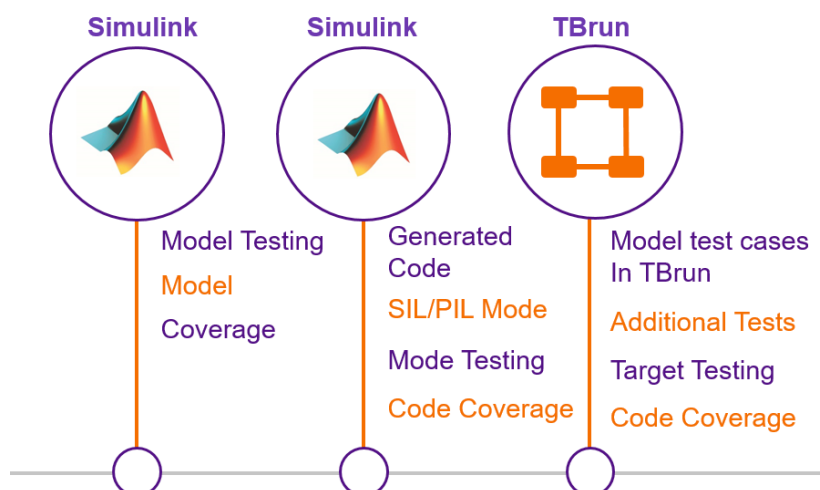


Figure 12: Generating structural coverage data of auto generated code with MathWorks Simulink and the LDRA tool suite

Several other tests are available using an integration such as this. The generated source code can be analysed statically to ensure compliance with an appropriate coding standard, such as MISRA C:2012 Appendix E¹⁸. Additional dynamic testing can be performed at the source level from within the LDRA tool suite. Requirements based tests can be created to verify functionality and collate structural coverage. Test data can also be imported from Simulink and used to migrate test data to the LDRA tool suite for efficiency.

Real time embedded systems based on auto generated code usually also include some level of conventionally written code. Software for board support packages, interrupt handlers, drivers, and other lower-level code is typically hand coded. Legacy code is almost always part of deployed systems. These portions of the system can be verified using traditional methods using the LDRA tool suite alongside auto-generated code.

The AUTOSAR Adaptive Platform: Adapting to a Changing World

However successful the AUTOSAR Classic Platform continues to be, the advent of the connected car has revealed its limitations. Simon Fürst, now the General Manager of Learning, Reasoning and Knowledge Representation with the BMW Group and a former AUTOSAR spokesperson says that the “AUTOSAR Classic Platform...remains the system of choice with regard to control units from the classical E/E domains of the automobile.”

“This platform continues to be the preferred solution for applications that have limited demands on hardware resources while fulfilling safety requirements and providing hard real - time capabilities.” he continues. “Upcoming new functionalities will be realized more efficiently by a software platform being designed for their special requirements. On the one hand we see, for example, there is a demand for number-crunching algorithms and high interconnectivity while on the other hand we have a focus on dependability of applications and systems, or new technologies such as ‘update over the air.’”¹⁹

Working with the AUTOSAR Adaptive Platform

Of course, any applications written for the AUTOSAR Adaptive Platform are likely to also require ISO 26262 compliance, and so all of the supporting tools outlined in Figure 2 and many of the testing processes discussed with reference to the Classic Platform application are equally applicable here.

There are, however, some key elements in the AUTOSAR Adaptive Platform to highlight from the perspective of the application developer. Figure 13 illustrates its layered architecture.

¹⁸ <https://www.misra.org.uk/tabid/72/Default.aspx>

¹⁹ http://www.ai-online.com/Adv/Previous/show_issue.php?id=6881#sthash.Cvo3ABXn.0xLSMTMG.dpbs

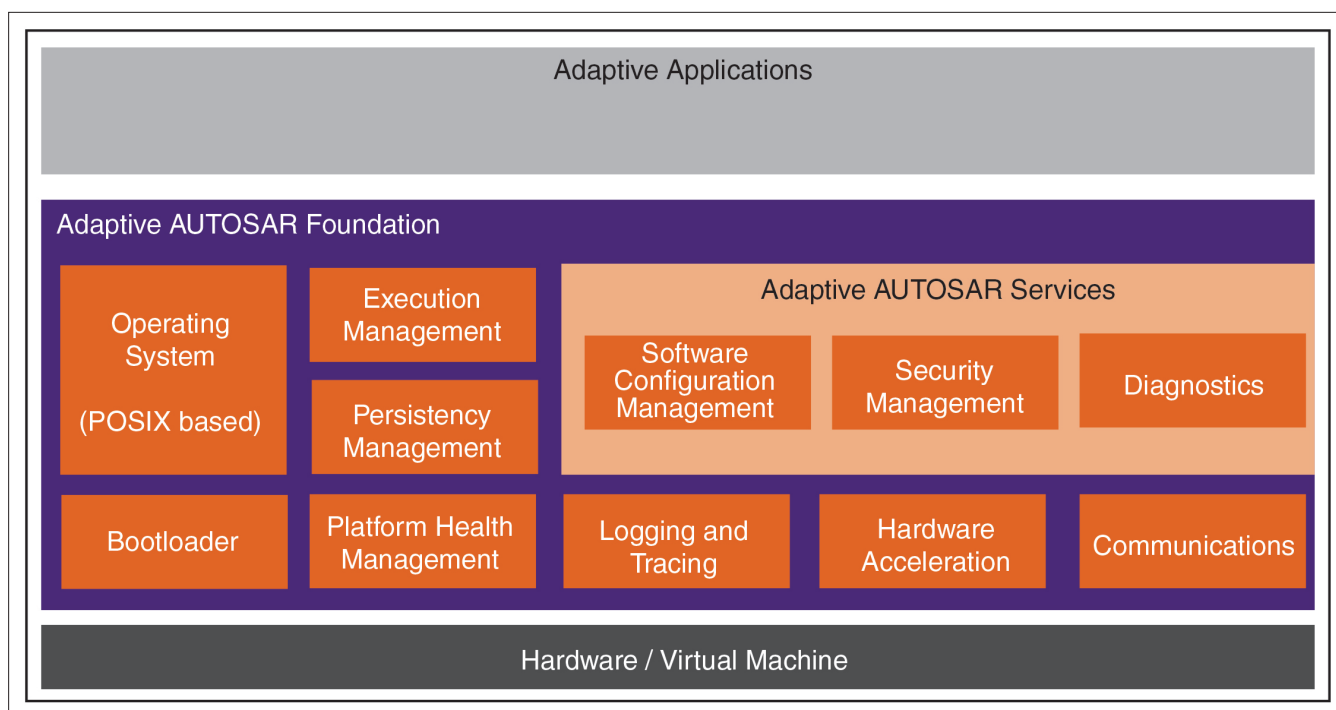


Figure 13: Adaptive AUTOSAR Architecture

Communications middleware takes the form of `ara::com`²⁰, a new AUTOSAR standard designed to complement the Adaptive Platform and provide a mechanism for communications between applications. `ara::com` is just one of the services provided for Adaptive Applications to call upon, and was created because it was felt that not all of AUTOSAR's key requirements were met by existing solutions.

Unlike AUTOSAR Classic applications, Adaptive applications do not consist of a number of source files compiled together to create a monolithic executable. Instead, they are separately executable, single or multi-threaded processes. Adaptive applications are configured on target by means of Manifest files, rather than at compilation time. The C++ language is preferred to C, with all the advantages of Object Orientation. And unlike the bespoke AUTOSAR Classic Operating System, Adaptive AUTOSAR uses any one of a number of POSIX PSE 51 compliant OS.

It is useful to reflect on two of these differences as compared with the Classic Platform development lifecycle.

POSIX Compliance

In addition to native POSIX conformant RTOSs such as QNX Neutrino²¹ and Lynx LynxOS²², there is a multitude of POSIX conformant offerings from such as Green Hills INTEGRITY²³ and Wind River VxWorks, and Linux Standard Base compliant options from Linux providers including Automotive Grade Linux²⁴.

Because all of these are well-established RTOS in their own right, there is considerable existing and proven support from partner organisations such as LDRA, providing support throughout the development lifecycle (Figure 14).

²⁰ AUTOSAR Explanation of `ara::comm` API

https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_EXP_ARAComAPI.pdf

²¹ QNX Neutrino <http://blackberry.qnx.com/en/products/neutrino-rtos/index>

²² Lynx Software Technologies LynxOS RTOS <http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/>

²³ Green Hills INTEGRITY RTOS <https://www.ghs.com/products/rtos/integrity.html>

²⁴ Automotive Grade Linux <https://www.automotivelinux.org/>

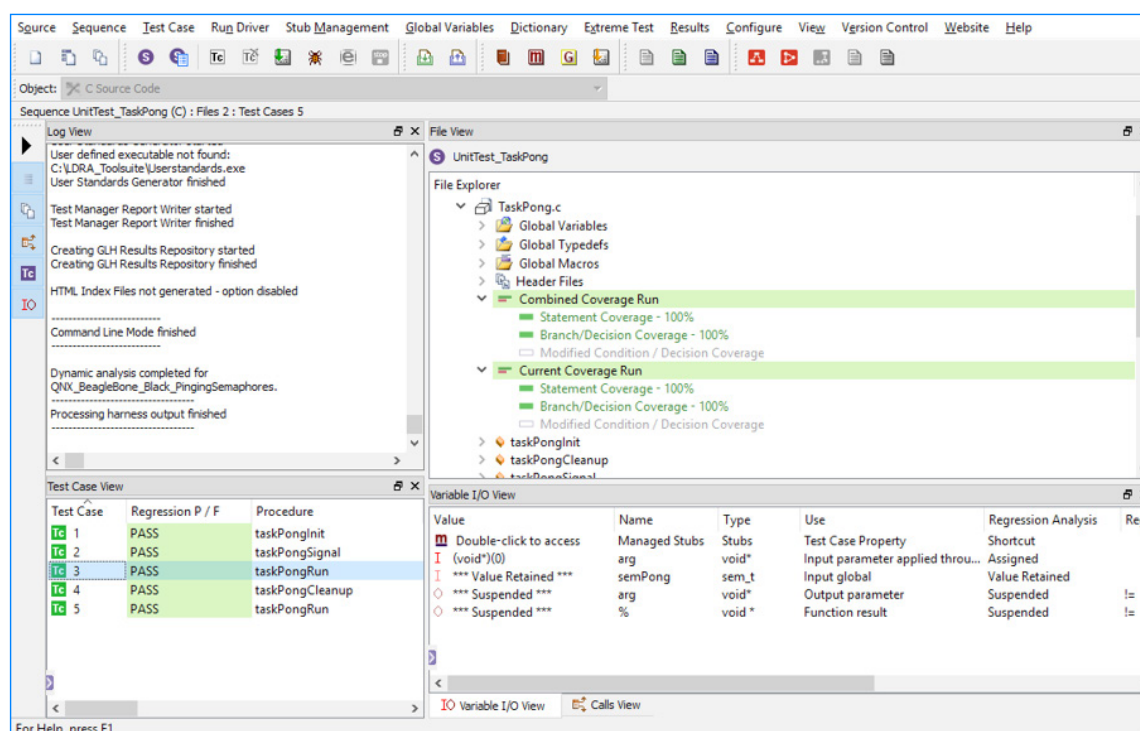


Figure 14: Support for QNX Momentics from the LDRA tool suite

AUTOSAR C++ Coding Guidelines

In April 2008, MISRA C++:2008 “Guidelines for the use of C++ language in critical systems”²⁵ (or MISRA C++) was published, to widespread approval. When the time came to establish an AUTOSAR coding standard, it was therefore sensible for the AUTOSAR group to base their work on this proven language subset.

However, MISRA C++ was based on the ISO/IEC 14882:2003²⁶ standard, often known as C++03. Following the MISRA publication, a major update to the C++ language was published in September 2011 (C++11²⁷), and a more minor revision in December 2014 (C++14²⁸) as illustrated in Figure 15.

The AUTOSAR document “Guidelines for the use of the C++14 language in critical and safety-related systems”²⁹ (hereafter AUTOSAR C++) was therefore designed as an addendum to MISRA C++ to address language features introduced more recently. These include lambda expressions, override specifiers, smart pointers, variable templates, and variadic templates. For the earlier language features, AUTOSAR C++ references many MISRA C++ rules without modification and without reproduction. Consequently, developers applying AUTOSAR C++ must also reference the MISRA C++ guidelines.

Although AUTOSAR C++ is clearly primarily focused on the automotive sector, it could be applied to other embedded application sectors.

²⁵ “MISRA C++ Guidelines for the Use of the C++ Language in Critical Systems”, ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF), June 2008. <https://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-cpp>

²⁶ ISO/IEC 14882:2003, “The C++ Standard Incorporating Technical Corrigendum 1, International Organization for Standardization, 2003.

²⁷ ISO/IEC 14882:2011 Standard C++ Foundation, “C++11 Overview”, <https://isocpp.org/wiki/faq/cpp11/>

²⁸ ISO/IEC 14882:2014 Standard C++ Foundation, “C++14 Overview”, <https://isocpp.org/wiki/faq/cpp14/>

²⁹ AUTOSAR Document ID 838, “Guidelines for the use of the C++14 language in critical and safety-related systems” https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf

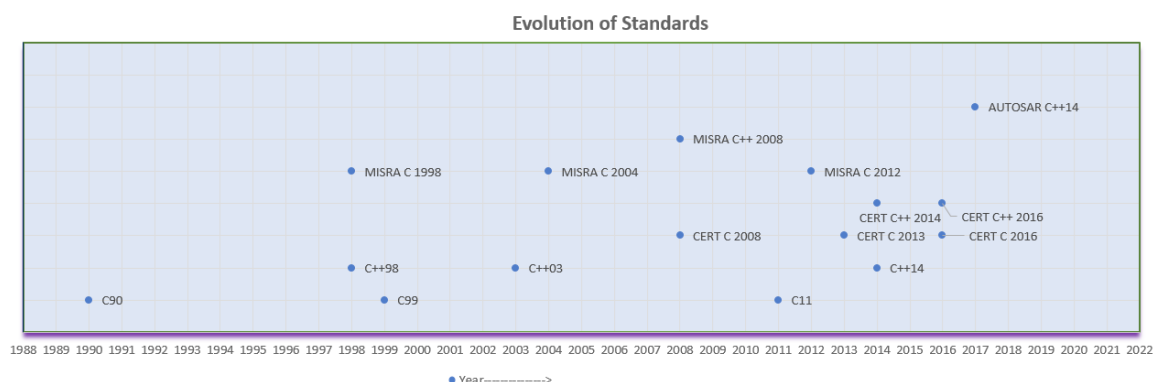


Figure 15: Timeline showing C and C++ Coding Standards releases

AUTOSAR C++ and Other Standards

In addition to its clear alignment with MISRA C++, the AUTOSAR standard includes traceability matrices showing its relationship to other C++ standards including JSF AV C++³⁰, SEI CERT C++³¹, and the C++ Core Guidelines³².

There is no such traceability matrix provided in connection with the automotive functional safety standard ISO 26262³³. However, it does clearly provide a basis for fulfilling the requirement of that document to use a language subset (Figure 16).

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity ^a	++	++	++	++
1b	Use of language subsets ^b	++	++	++	++
1c	Enforcement of strong typing ^c	++	++	++	++
1d	Use of defensive implementation techniques	O	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

^a An appropriate compromise of this topic with other methods in this part of ISO 26262 may be required

^b The objectives of method 1^b are

- Exclusion of ambiguously defined language constructs which may be interpreted differently by different modelers, programmers, code generators or compilers.
- Exclusion of language constructs which from experience easily lead to mistakes for example assignments in conditions or identical naming of local and global variables.
- Exclusion of language constructs which could result in unhandled run-time errors.

^c The objective of method 1^c is to impose principles of strong typing where these are not inherent in the language

Figure 16: Table 1 from ISO 26262: “Topics to be covered by modelling and coding guidelines”

AUTOSAR C++ Coding Rules Classification

Because AUTOSAR C++ is an addendum to MISRA C++, the MISRA format is retained for the complementary AUTOSAR C++ rules. Both standards take a common approach for rule classification by obligation levels, enforcement, and allocation. These classes give a clear indication on issues such as whether a rule can be automatically checked by static analysis, and where in the tool chain it is to be applied (with compiler warnings applied to the toolchain, for example).

³⁰ Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Document Number 2RDU00001 Rev C, Lockheed Martin Corporation, 2005.

³¹ Software Engineering Institute CERT C++ Coding Standard, Software Engineering Institute Division at Carnegie Mellon University, 2016.

³² Bjarne Stroustrup, Herb Sutter, C++ Core Guidelines, 2017.

³³ International standard ISO 26262 Road vehicles — Functional safety

Figure 17 illustrates the three categories of rules classification, complete with examples of how they are applied.

Category 1: Obligation Level			
Required		Advisory	
Rules must be followed to comply If not followed, then formal deviation is required		If Rules are not followed, then formal deviation is not required	
Category 2: Enforcement by Static Analysis			
Automated	Partially Automated	Non-automated	
Rules can be checked by Static Analyser	Rules can be partially checked by Static Analyser, but manual inspection is also required	Static Analyser cannot provide any help here or can provide limited help. Rules need to be tested by other means also, for example, manual review of the code.	
Example Rules			
Rule A10-1-1 Class shall not be derived from more than one base class which is not an interface class.	Rule A15-5-2 Program shall not be abruptly terminated	Rule A1-1-2 A warning level of the compilation process shall be set in compliance with project policies	
Category 3: Allocated Target			
Implementation	Verification	Toolchain	Infrastructure
These rules are implemented in code, software design	Rules are applied to the verification activity like manual code review, testing	Some rules can be applied to the toolchain like compiler, linker etc.	Rules can be applied to OS running on Target system.
Example Rules			
Rule M0-1-9 There shall be no dead code.	Rule A15-0-6 An analysis shall be performed to analyse the failure modes of exception handling.	Rule A1-1-2 A warning level of the compilation process shall be set in compliance with project policies.	Rule A0-4-1 The floating-point implementation shall comply with IEEE 754 standard.

Figure 17: Rule classifications common to AUTOSAR C++ and MISRA C++

The LDRA tool suite and AUTOSAR C++14

LDRA has a long standing tradition of supporting language subsets, and boasts significant representation on the MISRA C++ committee responsible for the standard on which AUTOSAR C++ is based. Support for AUTOSAR C++ in the LDRA static analysis tools is therefore a logical extension to existing support for subsets including MISRA C:2016 with Amendment 1, MISRA C++:2008, CERT C, and CERT C++ (Figure 18). This capability is significant for developers looking to comply with ISO 26262, where AUTOSAR C++ is the language subset of choice.

Code Review : Cpp_tunnel_lighting_system : C++ - AUTOSAR-C++ Model		
Procedure should be declared static. : CheckForInitialisation	Standard Code	Level of Violation
No master exception handler.	AUTOSAR-C++ A3-3-1	Required
	AUTOSAR-C++ A15-3-1,A15-3-3,A15-5-2	Required
Mountings.cpp		
Systemdata.cpp		
Comment possibly contains code.	AUTOSAR-C++ A2-8-2	Required
Included file not protected with #define.	AUTOSAR-C++ M16-2-3	Required
Use of old style /* comments in C++.	AUTOSAR-C++ A2-8-4	Required
Header Files		
TunnelData::SystemData::SystemData		
Constructor has insufficient initialisers.	AUTOSAR-C++ A12-1-1,A12-6-1	Required
Unused procedure parameter. : dummy1	AUTOSAR-C++ M0-1-11,M0-1-12	Required
TunnelData::SystemData::=		
Void function has no side effects. : TunnelData::SystemData::...	AUTOSAR-C++ M0-1-8	Required
Member function may be declared const. : TunnelData::Syst...	AUTOSAR-C++ M9-3-3	Required
Operator = doesn't return reference to *this.	AUTOSAR-C++ A13-2-1	Required
Unused procedure parameter. : dummy1	AUTOSAR-C++ M0-1-11,M0-1-12	Required

Figure 18: Violations of AUTOSAR C++ as shown in the LDRA tool suite

Including those rules that simply cite the MISRA C++ equivalents, AUTOSAR C++ has 337 rules out of which 308 are statically checkable. Of those, 78% are implemented in the LDRA tools.

Classification	Enhanced Enforcement	Fully Implemented	Partially Implemented	Not yet Implemented	Not statically Checkable	Total
Required	22	168	39	56	23	308
Advisory	1	8	1	11	2	23
Document	0	1	1	0	4	6
Total	23	177	41	67	29	337

Figure 19: AUTOSAR C++ coding compliance in LDRA tool suite

Example AUTOSAR C++ Rules and Violations

Rule A10-1-1: Addressing the “Diamond Problem”

“Class shall not be derived from more than one base class which is not an interface class.”

This rule addresses a C++ problem associated with multiple inheritance, commonly known as the diamond problem³⁴. This describes a situation where it becomes ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements that feature.

Figure 20 illustrates example code which violates this rule, and shows how it can be identified automatically.

```

class BClass : public AClass, // not compliant
              public AAClass // direct multiple inheritance
{
public:
    BClass();
    explicit BClass(const BClass &bc);
    BClass& operator=(const BClass &bc);
    ~BClass();
protected:
private:
};

```

Code snippet showing Class BClass derived from two base classes - AClass and AAClass

	Standard Code
example.cpp - E:\my\work\cpp_sample\multiple_inheritance\	
> No default constructor declared for class.	AUTOSAR-C++ A12-0-1
> Multiple direct inheritance found.	AUTOSAR-C++ A10-1-1
> At least one declaration in global namespace.	AUTOSAR-C++ M7-3-1
> Use of using directive.	AUTOSAR-C++ M7-3-4
main	
> No master exception handler.	AUTOSAR-C++ A15-3-1,A15-3-3,A15-5-2
> Basic type declaration used. : int	AUTOSAR-C++ A3-9-1

Figure 20: Violation of AUTOSAR rule A10-1-1 as identified by the LDRA tool suite

Rule A27-0-1: AUTOSAR C++ and Security

“Rule A27-0-1 Inputs from independent components shall be validated.”

Although security is not explicitly mentioned in AUTOSAR C++, this is just one example of many rules within the standard that are pertinent to it. The validation of external input is a vital defensive strategy, because external input is the attacker’s entry point. If unchecked, rogue data input has the potential to corrupt memory, run arbitrary code and take control of the system.

³⁴ <https://medium.freecodecamp.org/multiple-inheritance-in-c-and-the-diamond-problem-7c12a9ddbdec>

Figure 21 illustrates example code that violates this rule, and shows how it can be identified automatically.

The figure displays a C++ code snippet and a screenshot of the LDRA tool suite's Code Review window. The code snippet is as follows:

```
int MountingArea::NumLamps (LampAttributes MyLamp)
{
    int num = 0;
    int x;
    int y;
    x = (int) ceil ((float) length / (float) MyLamp.Width ());
    y = (int) ceil ((float) breadth / (float) MyLamp.Height ());

    num = x * y;

    return num;
}
```

A box highlights the lines where `x` and `y` are assigned values from `ceil` functions. A text box to the right states: "Code snippet where the values of x and y are not validated before use."

The screenshot of the Code Review window shows the following violations:

Violation	Rule
Parameter should be declared const.	AUTOSAR-C++ A7-1-1
Basic type declaration used.	AUTOSAR-C++ A3-9-1
Procedure is not called or referenced in code analysed. : Tu...	AUTOSAR-C++ A0-1-3,M0-1-1,M0-1-10
Constructor has insufficient initialisers.	AUTOSAR-C++ A12-1-1,A12-6-1
Var set by std lib func return not checked before use.	AUTOSAR-C++ A27-0-1,M0-3-1,M0-3-2
Local or member denominator not checked before use.	AUTOSAR-C++ A5-5-1,M0-3-1

Figure 21: Violation of AUTOSAR rule A27-0-1 as identified by the LDRA tool suite

Rule M8-5-2: Braced Initialization

"Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures."

Braced (or list) initialization was introduced into C++11, and hence is also a feature of C++14. It is a less error-prone initialization mechanism than that advocated in earlier versions.

The traditional approach to initialization involves the use of a '=' operator as shown in Figure 22:

```
int varint =2; // 2 will be stored in varint variable, no error
int myvar =2.7; // result in loss of information, as 2.7 will become 2 when stored in variable myvar
```

Figure 22: Traditional initialization

This approach is potentially misleading. In the example, the developer clearly intended that 2.7 should be stored in `myvar`, and the code will compile with no issue. However, because `myvar` is an integer type variable, it will actually be assigned a value of 2.

Figure 23 illustrates the equivalent use of braced initialization. In this case, the attempt to initialize `myvar` to 2.7 will cause a compilation error.

```
int varint {2}; // recommended method
int myvar {2.7}; // compilation error as narrowing is happening in this line
double mydouble {2.7}; // correct method
```

Figure 23: Braced (or list) initialization

AUTOSAR C++ Rule M8-5-2 encourages the use of braced initialization to address a commonplace issue resulting from the non-initialization of all elements in an array or structure. Figure 24 shows two examples relating to integer and character arrays.

```

INT_32 my_array[3] = { 1, 2 };           /* Not Compliant */
CHAR char_10[10] = "Hello";             /* Not Compliant */

```

Figure 24: partially initialized array, violation of Rule M8-5-2

Rule A4-10-1: Use of *nullptr*

“Only *nullptr* literal shall be used as the null-pointer-constant”

Before C++11, NULL was traditionally used to represent both a null pointer and an integer value of 0. Figure 25 shows a function (f) that is overloaded to take both int and char*. If it were called with NULL (left) to represent a null pointer then it would call the version overloaded for int. Using *nullptr* instead (right) results in the desired behaviour.

<pre> void f(int); void f(char*); void g() { f(NULL); //calls f(int) } </pre>	<pre> void f(int); void f(char*); void g() { f(nullptr); //calls f(char*) } </pre>
--	---

Figure 25: Demonstrating how the use of NULL as the null pointer constant can be misleading

Rule A7-2-3: Use of Scoped Enumeration

“Enumerations shall be declared as *scoped enum classes*.”

Enum classes were introduced in C++11. Unlike enumerators declared as plain enum, where enumerators are declared as enum classes, their enumerator names are local to the enum and their values do not implicitly convert to other types (such as another enum or int).

This is more restrictive, but means that unexpected outcomes are much less likely.

AUTOSAR C++ and ISO 26262

ISO 26262:2011 specifies a number of hazard classifications levels, known as ASILs (Automotive Safety Integrity Levels). Development process checks and safety measures are specified to avoid an unreasonable residual risk proportionate to the ASIL. ASILs range from A to D, where ASIL D represents the most hazardous and hence demanding level so that the overhead involved in producing a safety critical ASIL D system (e. g. automatic braking) is greater than that required to produce an ASIL A system with few safety implications (e. g. the in-car entertainment system).

ASILs are assigned as properties of each individual safety function, not as a property of the whole system or system component. The assigned ASIL for a safety function in a safety-related system is dictated by the properties of associated hazardous events, and is influenced by three of its attributes:

- Frequency of the situation (or “exposure”)
- Impact of possible damage (or “severity”)
- Controllability

ISO 26262:2011-6 Section 5 addresses both coding guidelines (including the use of language subsets) and the enforcement of low complexity, as illustrated in Figure 26.

ISO:26262-6 Section 5	Coding Guidelines	Applies to:
1a	Enforcement of Low Complexity	All ASILs
1b	Use of Language Subsets	All ASILs
1c	Enforcement of Strong Typing	All ASILs
1g	Use of Style Guides	ASILs B, C and D
1h	Use of Naming Conventions	All ASILs
ISO:26262-6 Section 7	Software Design	Applies to:
1a	Enforcement of Low Complexity	All ASILs
1b	Use of Language Subsets	All ASILs

Figure 26: Coding Guidelines and Software Component Size in ISO 26262-6:2011

There are many metrics available to aid in the management of code quality, and many organizations will nominate a favoured few of these for their projects (Figure 27).

	Value	Lower Limit	Upper Limit
▼ TunnelData::Cell::SetPoweredOutputLevel			
> Clarity	90% Metrics Successful		
> Maintainability	25% Metrics Successful		
> Cyclomatic Complexity	10	1	10
> Knots	16 : (Fail)	0	5
> Essential Cyclomatic Complexity	5 : (Fail)	1	3
> Essential Knots	4 : (Fail)	0	2
> Testability	88% Metrics Successful		
> Fan Out	2	0	5
> File Fan in	0	0	5
> Procedure Exit Points	1	0	1
> Number of Loops	3	0	4
> Number of Basic Blocks	27	1	30
> Executable reformatted Lines	90	2	200
> Cyclomatic Complexity	10	1	10
> Knots	16 : (Fail)	0	5

Figure 27: The LDRA tool suite measures more than 45 quality metrics, and each has configurable lower and upper limits. This Quality Review Report from the LDRA tool suite illustrates some of those metrics.

Complexity can be quantified by using metrics such as the Cyclomatic Complexity of code, the average length of basic blocks, and depth of loop nesting. These are complemented by other code metrics designed to evaluate various aspects of code quality, including the number of unreachable lines, unreachable branches, LCSAJs³⁵, and procedure exit points.

AUTOSAR C++ has several rules addressing the use of coding metrics.

Example AUTOSAR C++ Rules and Violations Relating to Coding Metrics.

Rule A1-4-2: Use of Defined Boundaries

“All code shall comply with defined boundaries of code metrics”

The objective of this rule is to control the complexity of the code, because complex code is difficult to maintain and test as previously discussed in relation to ISO 26262 (Figure 28).

▼ ISO 26262 - Road Vehicles Safety Standard - Unfulfilled
▼ Part 6: - Product Development : Software Level - Unfulfilled
▼ Section 5 - Initiation of product development at the software level - Unfulfilled
▼ Table 1 - Topics to be covered by modelling and coding guidelines - Unfulfilled
> 1a - Enforcement of low complexity - Unfulfilled
> 1b - Use of language subsets - Unfulfilled
> 1c - Enforcement of strong typing - Unfulfilled
> 1d - Use of defensive implementation techniques - Unfulfilled
> 1e - Use of established design principles - Unfulfilled
> 1f - Use of unambiguous graphical representation - Unfulfilled
> 1g - Use of style guides - Unfulfilled
> 1h - Use of naming conventions - Unfulfilled

Figure 28: The LDRA TBmanager component of the LDRA tool suite automates the monitoring of the fulfilment of objectives, including the “Enforcement of low complexity” illustrated here

³⁵ LCSAJ Linear Code Sequence and Jump Analysis, devised by Professor Michael Hennell (Founder of LDRA)
<http://www.professionalqa.com/lcsaj-testing>

A More Complete Solution

Bi-directional Traceability (ISO 26262-4:2011 and ISO 26262-6:2011)

Important though static and dynamic analysis is, there is much more to ISO 26262 compliance with or without AUTOSAR. Bi-directional traceability runs as a principle throughout ISO26262:2011, with each development phase required to accurately reflect the one before it. In theory, if the exact sequence of the V-model is adhered to, then the requirements will never change and tests will never throw up a problem. But life's not like that.

Consider, then, what happens if there is a code change in response to a failed integration test, perhaps because the requirements are inconsistent or there is a coding error. What other software units were dependent on the modified code?

Such scenarios can quickly lead to situations where the traceability between the products of software development falls down. Once again, while it is possible to maintaining traceability manually, automation helps a great deal.

Software unit design elements need to be bi-directionally traceable to both software safety requirements and the software architecture. The software units must then be implemented as specified and then be traceable to their design specification.

Automated requirements traceability tools are used to establish between requirements and tests cases of different scopes, which allows test coverage to be assessed (Figure 29). The impact of failed test cases can be assessed and addressed, as can the impact in requirements changes and gaps in requirements coverage. And artefacts such as traceability matrices can be automatically generated to present evidence of compliance to ISO 26262:2011.



Figure 29: Performing requirement based testing. Test cases are linked to requirements and executed within the LDRA tool suite.

In practise, initial structural coverage is usually accrued as part of this holistic process from the execution of functional tests on instrumented code leaving unexecuted portions of code which require further analysis. That ultimately results in the addition or modification of test cases, changes to requirements, and/or the removal of dead code. Typically, an iterative sequence of review, correct and analyse ensures that design specifications are satisfied.

During the development of a traditional, isolated system, that is clearly useful enough. But connectivity demands the ability to respond to vulnerabilities identified in the field. Each newly discovered vulnerability implies a changed or new requirement, and one to which an immediate response is needed – even though the system itself may not have been touched by development engineers for quite some time. In such circumstances, being able to isolate what is needed and automatically test only the impacted code becomes something much more significant.

Connectivity changes the notion of the development process ending when a product is launched, or even when its production is ended. Whenever a new vulnerability is discovered in the field, there is a resulting change of requirement to cater for it, coupled with the additional pressure of knowing that in such circumstances, a speedy response to requirements change has the potential to both save lives and enhance reputations. Such an obligation shines a whole new light on automated requirements traceability.

Conclusions

Standards vary in size, scope and detail, and it would be easy to think of AUTOSAR as just another standard. In fact, it is so huge that it is hard to imagine any one individual having detailed knowledge of it all.

The Adaptive Platform does not supersede the Classic Platform. It extends AUTOSAR's capabilities, with the net result that the challenge of getting a sufficient grasp on the different elements of AUTOSAR as a whole just became considerably bigger.

Any tools that have a proven track record across a range of safety critical industries in the fields of static analysis, dynamic analysis, unit test, and requirements and objectives traceability are surely beneficial. But applying a disparate collection of tools to the development cycle carries with it the potential to add needless complexity to an already challenging situation. Perhaps, then, the most significant attribute of all is the capability to integrate such a toolchain into one seamless whole.

Works Cited

An introduction to AUTOSAR

https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf

ANSYS Esterel Scade Suite

<http://www.esterel-technologies.com/products/scade-suite/>

Automotive Grade Linux

<https://www.automotivelinux.org/>

Automotive Industries – March 2016 - New AUTOSAR adaptive platform on its way

http://www.ai-online.com/Adv/Previous/show_issue.php?id=6881#sthash.Cvo3ABXn.oxLSMTMG.dpbs

AUTOSAR – A First Glance

<https://www.youtube.com/watch?v=F27jtKkxbAo>

AUTOSAR Adaptive Platform

<https://www.autosar.org/standards/adaptive-platform/>

AUTOSAR Classic Platform

<https://www.autosar.org/standards/classic-platform/>

AUTOSAR Document ID 838, “Guidelines for the use of the C++14 language in critical and safety-related systems”

https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf

AUTOSAR Explanation of ara:comm API

https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_EXP_ARAComAPI.pdf

AUTOSAR Foundation

<https://www.autosar.org/standards/adaptive-platform/>

AUTOSAR Software Component Template

https://www.autosar.org/fileadmin/user_upload/standards/classic/41/AUTOSAR_TPS_SoftwareComponentTemplate.pdf

Bjarne Stroustrup, Herb Sutter, C++ Core Guidelines, 2017.

Embedded.com - Electronic Blogs Accelerating development with model-based design

<https://www.embedded.com/electronics-blogs/say-what-/4440209/Accelerating-development-with-model-based-design>

freeCodeCamp - Onur Tuna – Multiple Inheritance in C++ and the Diamond Problem

<https://medium.freecodecamp.org/multiple-inheritance-in-c-and-the-diamond-problem-7c12a9ddbbec>

Green Hills INTEGRITY RTOS

<https://www.ghs.com/products/rtos/integrity.html>

IBM Rational Rhapsody family

<http://www-03.ibm.com/software/products/en/ratirhapfami/>

Works Cited

ISO 26262:2011 Road vehicles — Functional safety

ISO 26262:2011 Road vehicles — Functional safety — Part 4: Product development at the system level

ISO 26262:2011 Road vehicles — Functional safety — Part 5: Product development at the hardware level

ISO 26262:2011 Road vehicles — Functional safety — Part 6: Product development at the software level

ISO/IEC 14882:2003, The C++ Standard Incorporating Technical Corrigendum 1, International Organization for Standardization, 2003.

ISO/IEC 14882:2011 Standard C++ Foundation, “C++11 Overview”

<https://isocpp.org/wiki/faq/cpp11/>

ISO/IEC 14882:2014 Standard C++ Foundation, “C++14 Overview”

<https://isocpp.org/wiki/faq/cpp14/>

IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems

Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Document Number 2RDU00001 Rev C, Lockheed Martin Corporation, 2005.

LCSAJ Linear Code Sequence and Jump Analysis

<http://www.professionalqa.com/lcsaj-testing>

Lynx Software Technologies LynxOS RTOS

<http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/>

MathWorks Simulink

<https://www.mathworks.com/products/simulink.html>

MISRA C++ Guidelines for the Use of the C++ Language in Critical Systems, ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF), June 2008.

<https://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-cpp>

MISRA – The Motor Industry Software Reliability Association

<https://www.misra.org.uk/>

QNX Neutrino

<http://blackberry.qnx.com/en/products/neutrino-rtos/index>

Software Engineering Institute CERT C++ Coding Standard, Software Engineering Institute Division at Carnegie Mellon University, 2016.



www.ldra.com
LDRA
LDRA UK & Worldwide
 Portside, Monks Ferry,
 Wirral, CH41 5LH
 Tel: +44 (0)151 649 9300
 e-mail: info@ldra.com

LDRA Technology Inc.
 2540 King Arthur Blvd, Suite 228
 Lewisville, Texas 75056
 United States
 Tel: +1 (855) 855 5372
 e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
 Unit No B-3, 3rd floor Tower B,
 Golden Enclave, HAL Airport Road
 Bengaluru
 560017
 India
 Tel: +91 80 4080 8707
 e-mail: india@ldra.com