LDRA
Software Technology

Technical Briefing

AUTOSAR Classic and AUTOSAR Adaptive: Your Platforms for Success

www.ldra.com

* Registration required to download the document

Background

This document focuses on how an integrated and comprehensive set of tools such as the LDRA tool suite® can help ease the development path of AUTOSAR applications for the Classic and Adaptive Platforms.

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." The Classic Platform[1] is AUTOSAR's established solution for embedded systems with hard real-time and safety constraints, first published in 2005.

The Adaptive Platform[2] is AUTOSAR's solution for high-performance computing ECUs to build safety-related systems for use cases such as highly automated and autonomous driving.

The Foundation[3] standard underpins both platforms, enforcing interoperability between them by defining requirements and technical specifications common to both, such as communications protocols.

Adherence to either AUTOSAR platform standard does not in itself imply compliance with the functional safety standard "ISO 26262 Road vehicles — Functional safety,"[4] and for that reason the AUTOSAR and ISO standards need to be considered concurrently. 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 1 shows how LDRA and other complementary tools can be used throughout the development lifecycle.

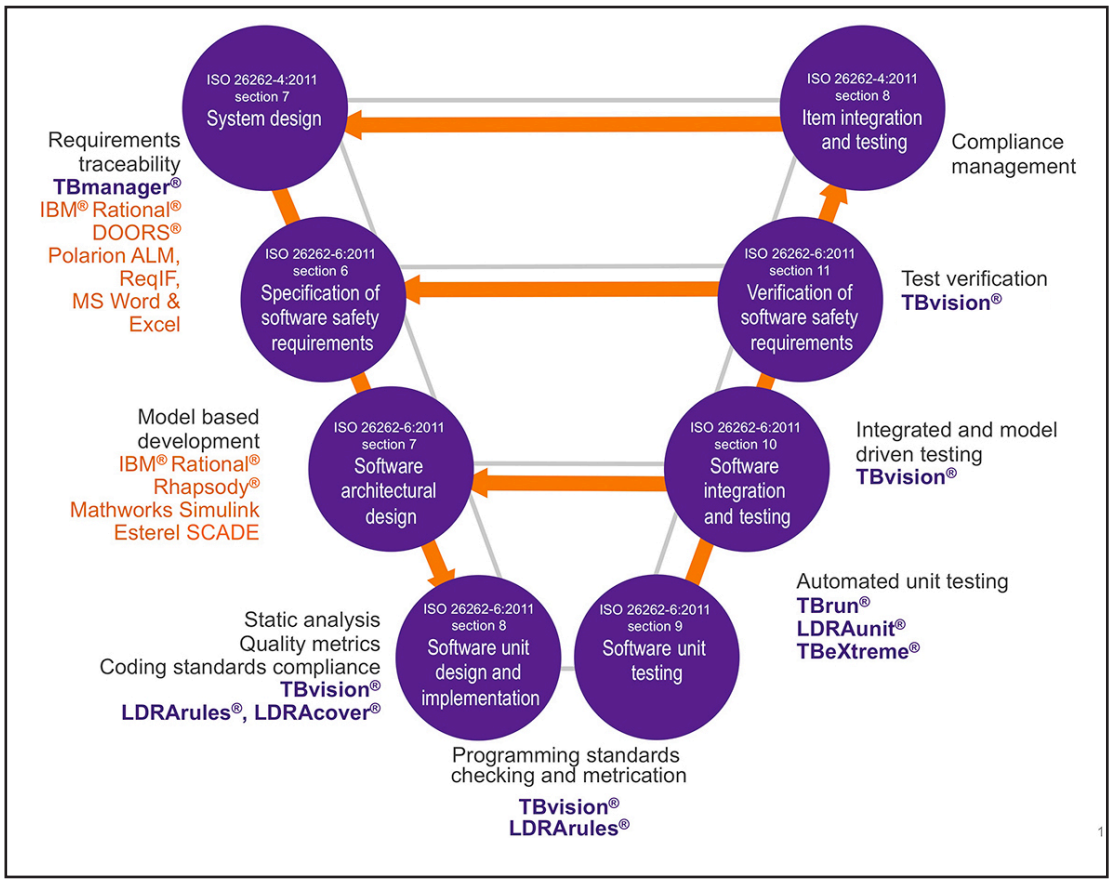

Figure 1: Software-development V-model with cross-references to ISO 26262 and standard development tools

[1] AUTOSAR Classic Platform https://www.autosar.org/standards/classic-platform/
[2] AUTOSAR Adaptive Platform https://www.autosar.org/standards/adaptive-platform/
[3] AUTOSAR Foundation https://www.autosar.org/standards/adaptive-platform/
[4] International standard ISO 26262 Road vehicles — Functional Safety

AUTOSAR Classic Platform

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.

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.

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 process[5].

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

With reference to the ISO 26262 V-model (Figure 1) and to the elements of it specifically impacted by AUTOSAR, it is useful to consider some elements of 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 (SAE J3061 Section 8.6.5)

The illustration in Figure 2 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.

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 2: Mapping the capabilities of the LDRA tool suite to “Table 6: Methods for the verification of the software architectural design” specified by ISO 26262-6:2011[6]

These guidelines combine to make the resulting code more reliable, less prone to error, easier to test, and/or easier to maintain. Figure 3 shows just one example of how language subset violations can be presented.

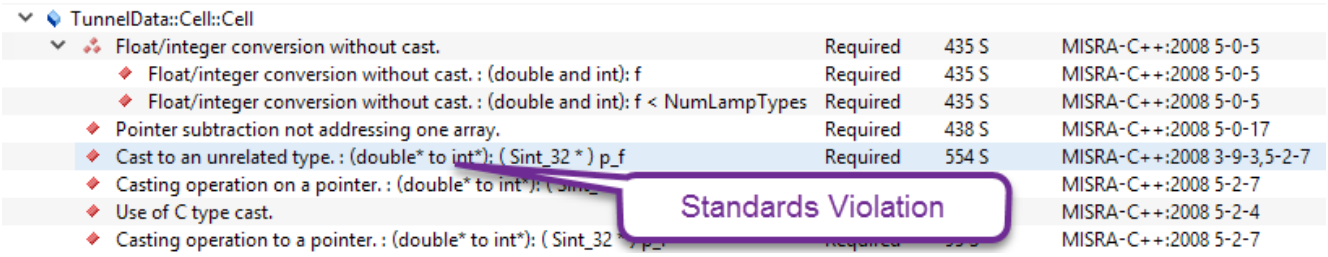

Figure 3: Highlighting violated coding guidelines in the LDRA tool suite

[5] An introduction to AUTOSAR https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf
[6] Based on table 6 from ISO 26262-6:2011, Copyright© 2015 IEC, Geneva, Switzerland. All rights acknowledged

ISO 26262-6:2011 references the MISRA[7] 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. 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.

LDRA's static analysis tools can also 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.

More generally, they 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.

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.

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 at the unit and integration levels. 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.

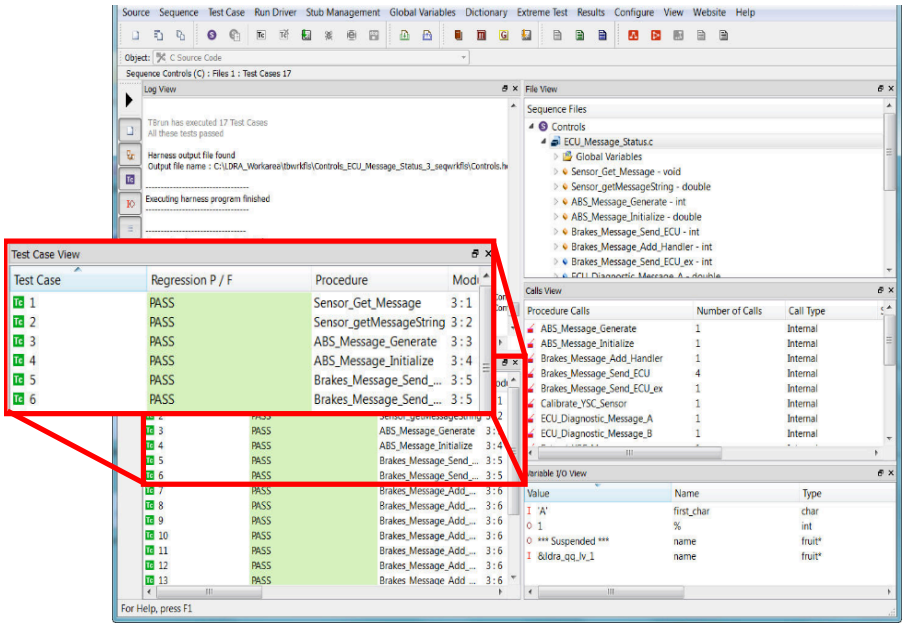

Figure 4: Performing requirement based unit-testing using the LDRA tool suite

The example in Figure 4 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.

[7] MISRA – The Motor Industry Software Reliability Association https://www.misra.org.uk/

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 can be re-run (regression tested), automatically re-applying tests to ensure that the changes do not compromise any established functionality.

Structural Coverage Metrics

In addition to showing that the software functions correctly, dynamic analysis is used to generate structural coverage metrics. In conjunction with demonstrating 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 5).

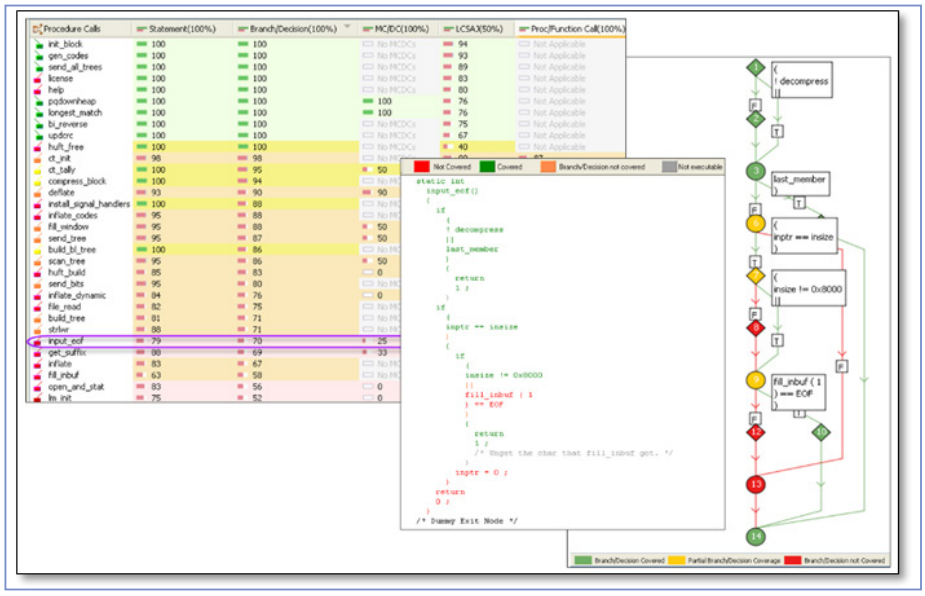

Figure 5: Examples of representations of structural coverage within the LDRA tool suite

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 Rhapsody[8], MathWorks Simulink[9] and Esterel SCADE[10]. 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.

The AUTOSAR Adaptive Platform

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.'"[11]

[8] http://www-03.ibm.com/software/products/en/ratirhapfami/
[9] https://www.mathworks.com/products/simulink.html
[10] http://www.esterel-technologies.com/products/scade-suite/
[11] http://www.ai-online.com/Adv/Previous/show_issue.php?id=6881#sthash.Cvo3ABXn.oxLSMTMG.dpbs

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 6 illustrates its layered architecture.

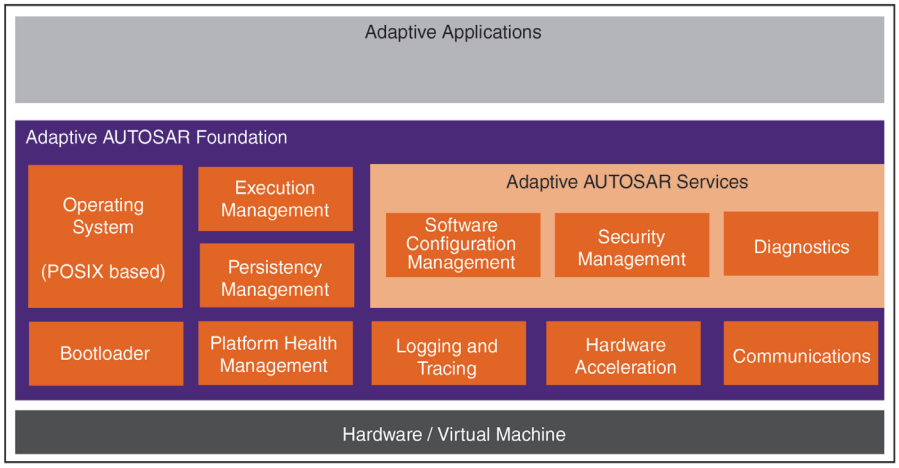

Figure 6: Adaptive AUTOSAR Architecture

Adaptive applications communicate using ara::com middleware, a new AUTOSAR standard designed to complement the Adaptive Platform and provide a mechanism for communications between applications. Ara::com[12] 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 QNX Neutrino[13] and Lynx LynxOS[14], there is a multitude of POSIX conformant offerings from such as Green Hills INTEGRITY[15] and Wind River VxWorks, and Linux Standard Base compliant options from Linux providers including Automotive Grade Linux[16].

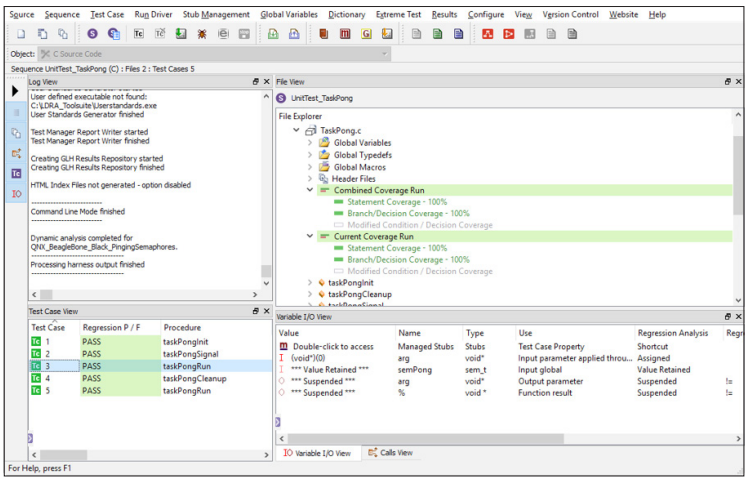

Figure 7: Support for QNX Momentics from the LDRA tool suite

[12] https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_EXP_ARAComAPI.pdf
[13] QNX Neutrino http://blackberry.qnx.com/en/products/neutrino-rtos/index
[14] Lynx Software Technologies LynxOS RTOS http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/
[15] Green Hills INTEGRITY RTOS https://www.ghs.com/products/rtos/integrity.html
[16] Automotive Grade Linux https://www.automotivelinux.org/

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 7).

AUTOSAR C++ Coding Guidelines

The AUTOSAR document "Guidelines for the use of the C++14 language in critical and safety-related systems"[17] (hereafter AUTOSAR C++) was designed as an addendum to MISRA C++:2008 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.

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++[18] , SEI CERT C++[19], and the C++ Core Guidelines[20].

There is no such traceability matrix provided in connection with ISO 26262. However, it does clearly provide a basis for fulfilling the requirement of that document to use a language subset.

AUTOSAR C++ Coding Rules Classification

Because AUTOSAR C++ are an addendum to MISRA C++, the MISRA format is retained for the complementary AUTOSAR C++ rules. Both 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).

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 8). This capability is significant for developers looking to comply with ISO 26262, where AUTOSAR C++ is the language subset of choice.

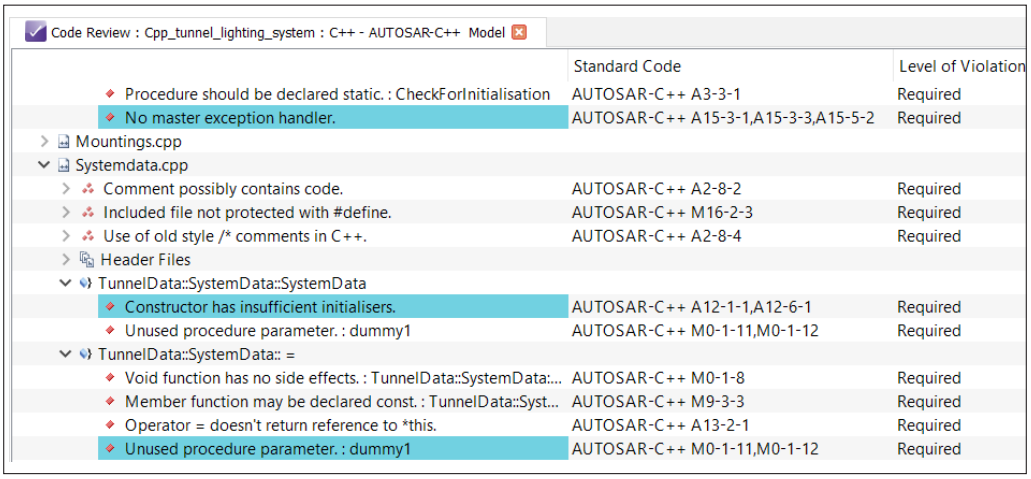

Figure 8: Violations of AUTOSAR C++ as shown in the LDRA tool suite

[17] 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

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

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

[20] Bjarne Stroustrup, Herb Sutter, C++ Core Guidelines, 2017.

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 9: AUTOSAR C++ coding compliance in the 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 problem . 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 10 illustrates example code which violates this rule, and shows how it can be identified automatically.

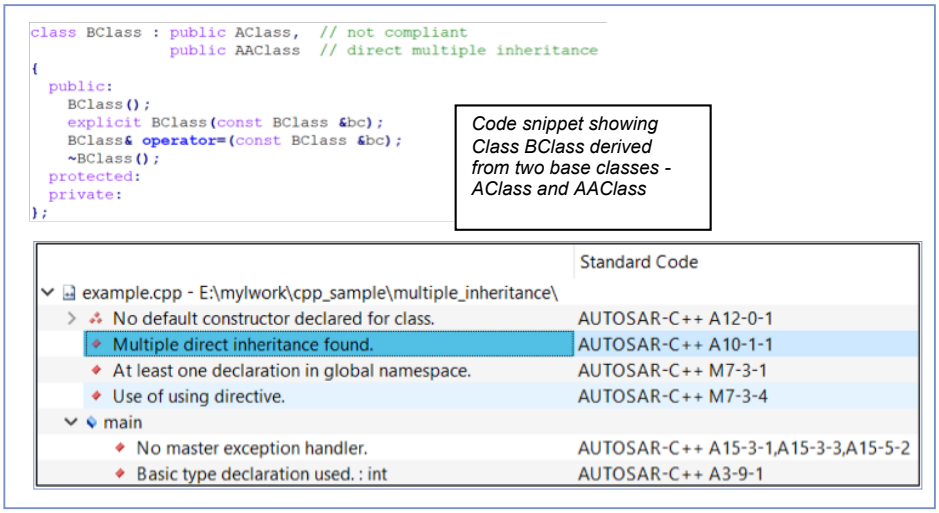

Figure 10: Violation of AUTOSAR rule A10-0-1 as identified by the LDRA tool suite

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

"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.
Figure 11 illustrates example code which violates this rule, and shows how it can be identified automatically.

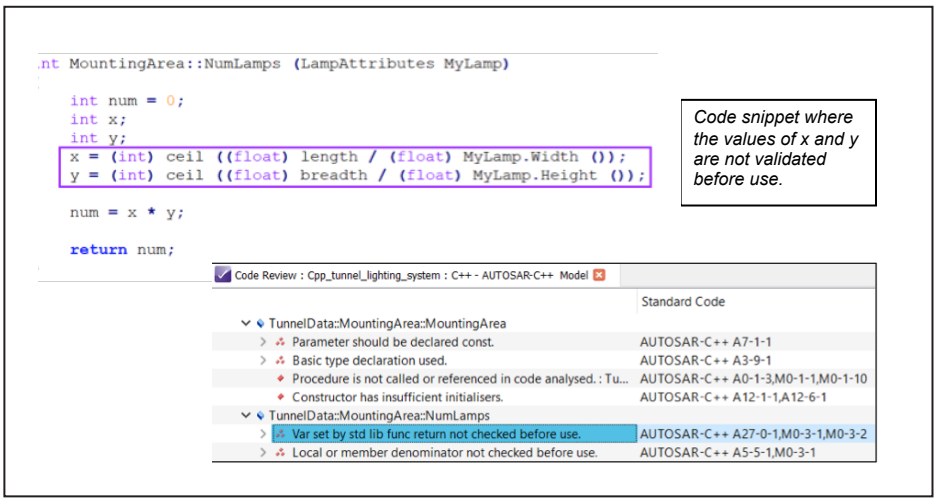

Figure 11: Violation of AUTOSAR rule A27-0-1 as identified by the LDRA tool suite

[21] https://medium.freecodecamp.org/multiple-inheritance-in-c-and-the-diamond-problem-7c12a9ddbbec

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 12 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.

```
void f(int);
void f(char*);

void g()
{
    f(NULL); //calls f(int)
}
```

```
void f(int);
void f(char*);

void g()
{
    f(nullptr); //calls f(char*)
}
```

Figure 12: Demonstrating how the use of NULL as the null pointer constant can be misleading

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 (Figure 13).

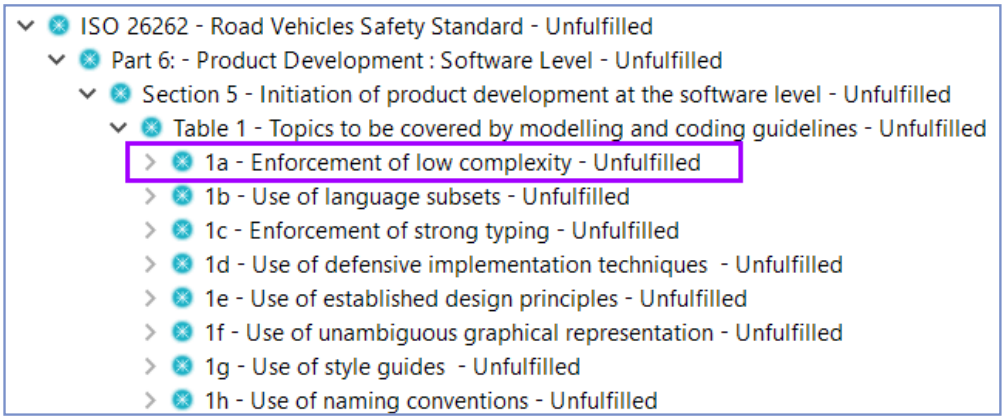

Figure 13: The LDRA TBmanager component of the LDRA tool suite automates the monitoring of the fulfilment of objectives, including the "Enforcement of low complexity" illustrated here

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

Important though static and dynamic analysis are, there is much more to ISO 26262 compliance with or without AUTOSAR. Bi-directional traceability runs as a principle throughout ISO 26262:2011, with each development phase required to accurately reflect the one before it.

Automated requirements traceability tools are used to establish between requirements and tests cases of different scopes, which allows test coverage to be assessed (Figure 14). 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.

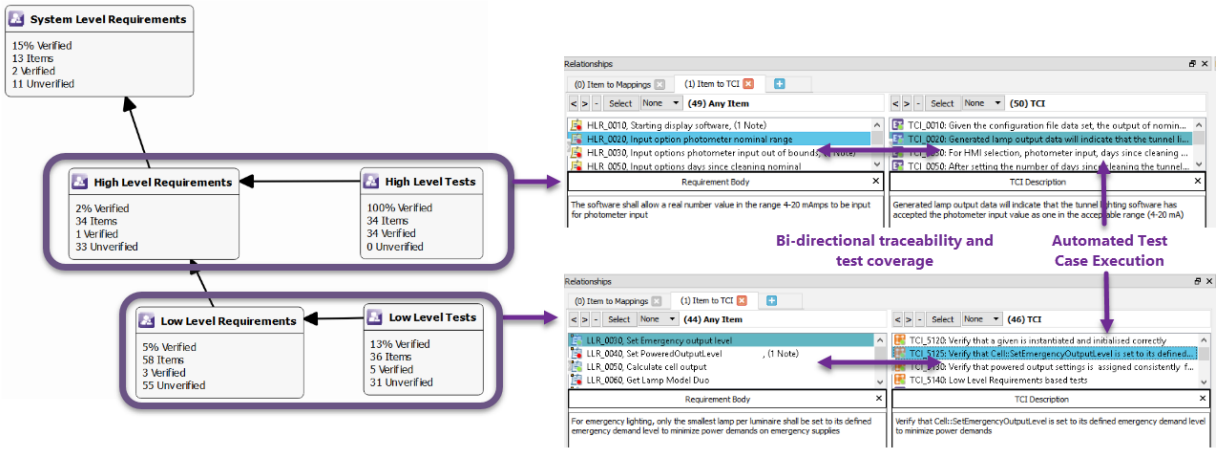

Figure 14: Performing requirement based testing. Test cases are linked to requirements and executed within the LDRA tool suite.

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.

www.ldra.com

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