LDRA Software Technology

Technical White Paper

Implementing ISO 26262 second edition with the LDRA tool suite®

Cost effective software certification from ASIL A to ASIL D

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.

Contents

Background ... 3
Automotive Safety Integrity Levels (ASILs) ... 3
Changes to ISO 26262 second edition ... 4
Security in the context of ISO 26262 ... 4
ISO 26262 Second Edition process objectives ... 5
Automating ISO 26262 processes ... 7
Technical safety concept (part 4, clause 6) ... 7
Specification of software safety requirements (part 6, clause 6) ... 8
Software architectural design (part 6, clause 7) ... 8
- Reverse engineering ... 10
- Model based development ... 10

Software unit design and implementation (part 6, clause 8) ... 10
- Coding guidelines ... 10
- Software architectural design and unit implementation ... 12

Software unit verification (part 6, clause 9) and software integration and testing (part 6, clause 10) ... 14
- Structural coverage metrics ... 17
- Software test and model based development ... 19

Bidirectional traceability (parts 4 and 6) ... 20
Confidence in the use of software tools (part 8) ... 23
Conclusions ... 24
Works Cited ... 25

Background

There is an ever-widening range of automotive electrical and/or electronic (E/E/PE) systems such as adaptive driver assistance systems, anti-lock braking systems, steering and airbags. Their increasing levels of integration and connectivity provide almost as many challenges as their proliferation, with non-critical systems such as entertainment systems sharing the same communications infrastructure as steering, braking and control systems. The net result is a necessity for exacting functional safety development processes, from requirements specification, design, implementation, integration, verification, validation, and through to configuration.

ISO 26262 "Road vehicles – Functional safety" was updated in 2018[1], having first been published in 2011[2] in response to this 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 61508[3]. The resulting ISO 26262 has become the dominant automotive functional safety standard, and its requirements and processes are becoming increasingly familiar throughout the industry.

Many of the practices adopted by IEC 61508 and its derivatives can be traced to the commercial and defence avionics industries. The RTCA Inc. series standards such as DO-178B/C[4], DO-254, DO-278A and their supplements, have proven that adherence to a structured set of best practices results in large scale reliable systems that protect public safety. Although ISO 26262 even in its original form was a relatively recent innovation, this history is significant because the establishment of functional safety standards elsewhere gave rise to a sophisticated industry providing support for their effective application. Consequently, the automotive industry benefits from established and proven tools and techniques predating ISO 26262 itself.

This document describes the key software development and verification process activities of the standard, and uses LDRA's tool suite to show how automation can assist in proving compliance in a cost-effective manner. Extracts from ISO 26262 second edition are shown in *italics*[5].

Automotive Safety Integrity Levels (ASILs)

Like DO-178B/C and IEC 61508 before it, ISO 26262 specifies a number of hazard classifications levels – in this case, 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 significantly 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 at the item level, where an item is defined as a "*system or combination of systems, to which ISO 26262 is applied, that implements a function or part of a function at the vehicle level*". 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 supports the decomposition of Functional Safety Requirements (FSRs) in a process often known as "ASIL decomposition" (Figure 1), which can help to reduce cost and effort.

[1] https://www.iso.org/standard/68383.html
[2] https://www.iso.org/news/2012/01/Ref1499.html
[3] IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems
[4] RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification, Prepared by SC-205, December 13, 2011
[5] Extracts from ISO 26262:2018, Copyright © The British Standards Institution 2018. All rights acknowledged.

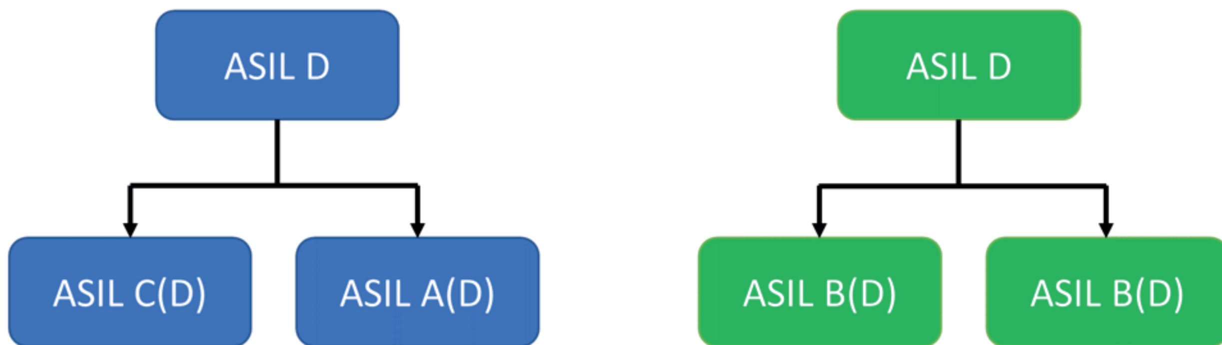

Figure 1: Decomposition of the different ASIL ratings throughout the item can occur over different systems, elements and components

Decomposition of the different ASIL ratings throughout the item can occur over different systems, elements and components, working down through the systems, subsystems, software, and hardware. ASIL decomposition is typically performed manually and must result in redundant safety requirements allocated to design elements of sufficient technical independence.

Changes to ISO 26262 second edition

The 2018 revision to the ISO 26262 standard reflects industry feedback and updates based on advances in technology since the standard was originally published. Reconstructed to provide more detailed objectives and extensions to the overall vocabulary, notable additions to the standard include:

- objective oriented confirmation measures
- management of safety anomalies
- references to cyber-security
- updated target values for hardware architecture metrics
- evaluation of hardware elements
- additional guidance on dependent failure analysis
- guidance on fault tolerance, safety-related special characteristics, and software tools
- guidance for model-based development and software safety analysis

In addition, two completely new parts were added to the standard: ISO 26262 Part 11 which relates to Semiconductors[6] and ISO 26262 Part 12 which relates to Motorcycles[7].

ISO 26262 second edition therefore consists of 12 parts with three focused on product development: system level (Part 4)[8], hardware level (Part 5)[9], and software level (Part 6)[10]. ISO 26262 Part 6 provides detailed industry specific guidelines for the production of all software for automotive systems and equipment, whether it is safety critical or not.

Security in the context of ISO 26262

Most embedded software used in the automotive sector has not taken security requirements into account, simply because security and connectivity have not really been on the agenda. Automotive embedded applications have tended to be static, fixed function, device specific implementations. Isolation has been a sufficient guarantee of security for many years, and practices and processes have relied on that status.

Even as communication within the car became increasingly sophisticated, while ever the vehicle itself remained an isolated entity, demonstrations of the ability to infiltrate safety critical systems from more benign applications remained something of an academic point[11]. After all, it has always been possible to damage a car's electrical systems with a pair of well-aimed wire cutters.

[6] ISO 26262-11:2018 Road vehicles -- Functional safety – Part 11: Guidelines on application of ISO 26262 to semiconductors
[7] ISO 26262-12:2018 Road vehicles -- Functional safety – Part 12: Adaptation of ISO 26262 for motorcycles
[8] ISO 26262-4:2018 Road vehicles -- Functional safety -- Part 4: Product development at the system level
[9] ISO 26262-5:2018 Road vehicles -- Functional safety -- Part 5: Product development at the hardware level
[10] ISO 26262-6:2018 Road vehicles -- Functional safety -- Part 6: Product development at the software level
[11] http://www.autosec.org/pubs/cars-oakland2010.pdf -- Experimental Security Analysis of a Modern Automobile, Karl Koscher, Stephen Checkoway et.al.

The key element is therefore the connection to the outside world. That changes things dramatically because it makes remote access possible while requiring no physical modification to the car's systems, most famously demonstrated in Miller and Valasek's work "Remote Exploitation of an Unaltered Passenger Vehicle"[12].

Perhaps because of this traditional isolation of automotive systems, ISO 26262 first edition made no specific mention of security; indeed, in the field of safety critical embedded software, security concerns have generally been perceived to be a separate domain from the core business of functional safety. Yet if hackers have the potential to remotely control steering, braking, and engine control systems, then security vulnerabilities clearly put safety at risk. In this situation safety and security are indistinguishable.

In acknowledgement of that fact, in ISO 26262 second edition channels of communication between functional safety and cybersecurity have been identified at both the functional safety management level and product development at the system level. Such an approach provides a useful interface to the recommendations outlined in the current SAE J3061 Cybersecurity Guidebook for Cyber-Physical Vehicle Systems"[13] "Cybersecurity" and the proposed ISO/SAE 21434 "Road vehicles – Cybersecurity engineering[14]". ISO 26262-2:2018 Annex E "Guidance on potential interaction of functional safety with cybersecurity" further discusses "*the possible interactions between the activities of functional safety and cybersecurity*".

As for any other safety related risk, as soon as there is potential for security vulnerabilities to threaten safety, ISO 26262 demands safety goals and requirements to deal with them. It requires that the safety goals be classified with appropriate ASILs for their criticality, designed with due reference to their classification, and developed and verified to show compliance with the system's safety requirements. In short, the action to be taken to deal with each safety-threatening security issue needs to be proportionate to the risk (and hence ASIL).

ISO 26262 second edition process objectives

A key element of part 4 is the practice of allocating technical safety requirements in the system design specification, and developing that design further to derive an item integration and testing plan. It applies to all aspects of the system including software, with the explicit subdivision of hardware and software development practices being dealt with further down the "V" model.

Part 6 refers more specifically to the development of the software aspects of the product. It is concerned with:

- general topics for the product development at the software level
- specification of software safety requirements
- software architectural design
- software unit design and implementation
- software unit verification
- software integration and testing
- testing of the embedded software

Figure 2 is an extract from the standard, shown here with parts 4 and 6 highlighted for context.

[12] http://illmatics.com/Remote%20Car%20Hacking.pdf -- Remote Exploitation of an Unaltered Passenger Vehicle, Dr. Charlie Miller & Chris Valasek, August 2015
[13] https://webstore.ansi.org/standards/sae/sae30612016j3061 -- Cybersecurity Guidebook For Cyber-Physical Vehicle Systems
[14] https://www.iso.org/standard/70918.html -- Road Vehicles -- Cybersecurity engineering

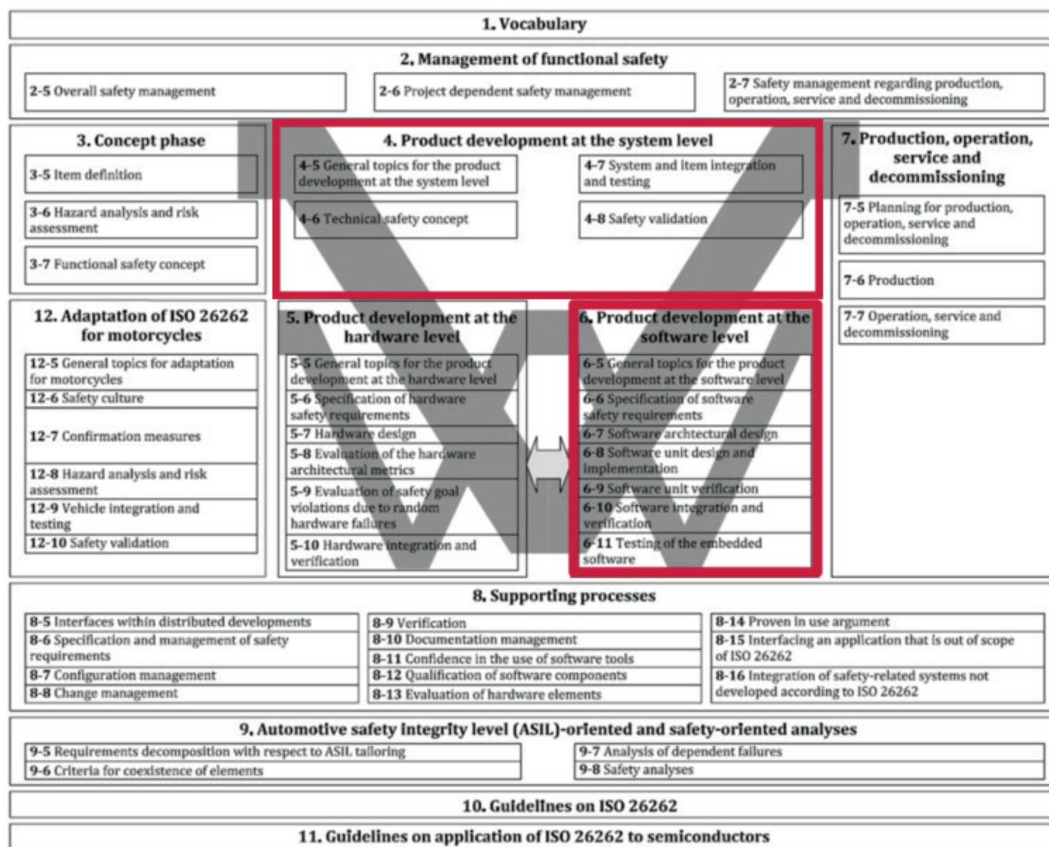

Figure 2: Software development is primarily referenced in parts 4 and 6, shown here in the context of ISO 26262 second edition as a whole.

These different elements are not intended to be viewed as isolated silos; indeed, the standard insists that safety requirements are traceable to architectural design, that architectural design to unit design and implementation, and so on. ISO 26262 also requires that this traceability is bidirectional so that (for instance) there are no safety requirements not covered in the architectural design, and no design elements not demanded by the architecture. This approach reduces the risk of failure by ensuring that not only is all required functionality present and proven, but also that there is no redundant code or “feature creep”.

Once software safety requirements and architecture are defined, the software units can be designed and then implemented in accordance with that design. The developing organization is required to establish coding rules appropriate to the circumstances of the project, and the source code for the implemented units must be validated to ensure that those coding rules are adhered to. The software units must then be dynamically tested (executed) to demonstrate that they fulfil the software unit design specification and do not include unspecified functionality. Structural coverage analysis is then required to determine which code structures and component interfaces have not been exercised during execution of these requirements-based test procedures. Unexecuted portions of code require further analysis resulting in addition or modifications of test cases, changes to inadequate requirements, removal of dead or deactivated code, or unintended functionality.

Variations in low level implementation details such as endianness and the sizes of data and address words can result in differing behaviour between test and target environments. Part 6 requires that the test environment shall correspond as closely as possible to the target environment.

Although it provides extensive guidelines relating to the use of software tools, ISO 26262 does not require that they are used. However, for all but very trivial applications, attempting to meet the standard without some level of automation would be a disproportionately labour intensive task. Tools are available to assist with almost all aspects of Part 6 not only to automate the tests themselves, but also to provide documentary evidence (or “artefacts”) relating to their successful completion.

Automating ISO 26262 processes

Figure 3 illustrates how each key stage in the ISO 26262 software development lifecycle can be automated.

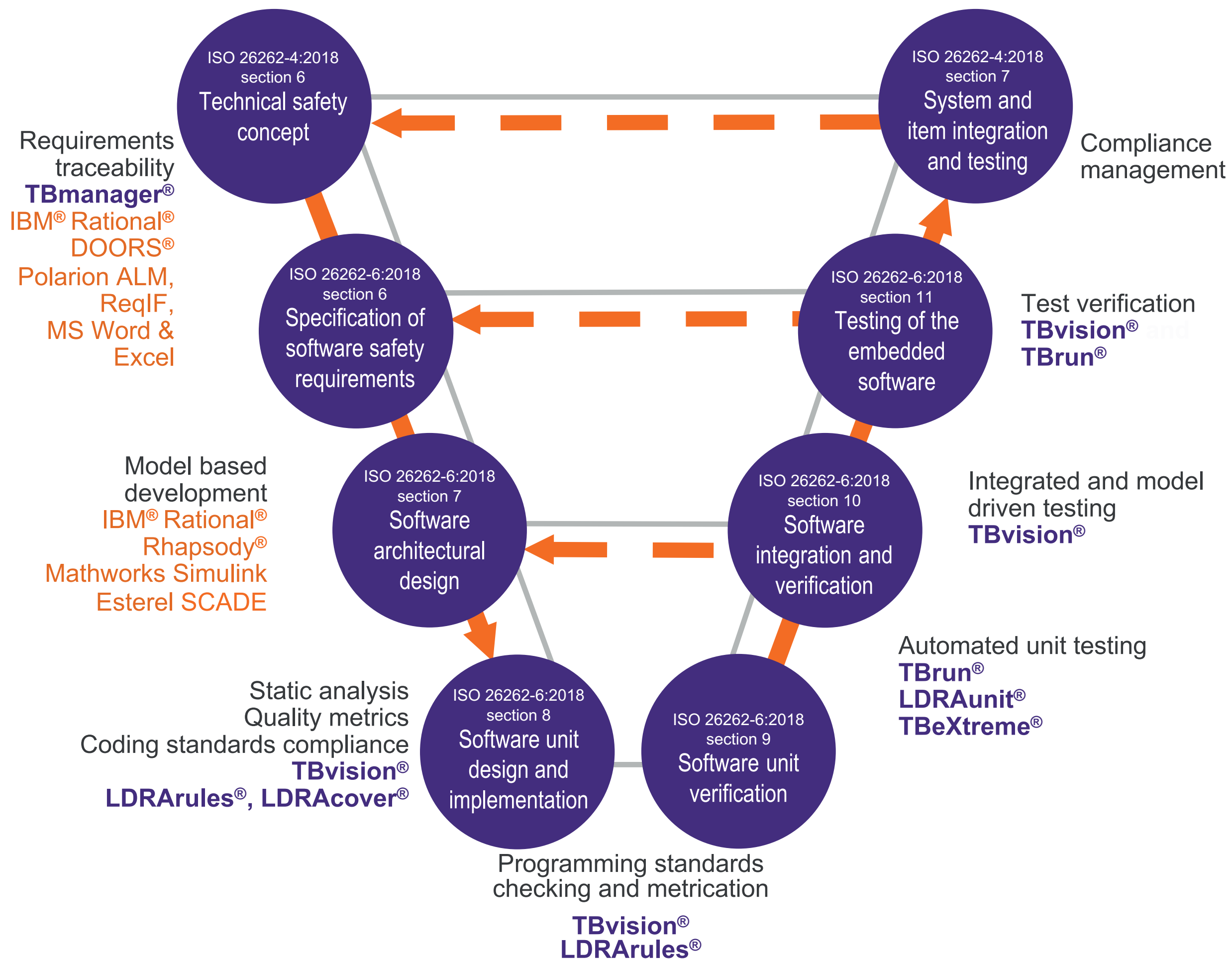

Figure 3: Mapping the capabilities of an automated tool chain to the ISO 26262 second edition software development process guidelines

Software tools effectively leverage the experience and expertise of their vendors in this and other industries to ease the path to certification.

Each element of the V diagram is expanded in the following clauses, and relevant tables from the ISO 26262 second edition standard expanded and referenced. Note that each of the tables has a clear association with a particular element of the V-Model shown in Figure 3.

Technical safety concept (part 4, clause 6)

"The technical safety concept is an aggregation of the technical safety requirements and the corresponding system architectural design, that provides rationale as to why the system architectural design is suitable to fulfil safety requirements, resulting from activities described in ISO 26262-3 (with consideration of non-safety requirements) and design constraints."

Although part 6 is the primary document for software development, it needs to be considered in the context of the system design for the product as a whole which is the domain of part 4. In order to develop a system architectural design, functional safety requirements, technical safety requirements, and non-safety-related requirements are implemented. Clearly those work products include software considerations, but in this system design sub-phase, safety-related and non-safety-related requirements are handled within one development process whether they impact software, hardware, or both in the form of the hardware-software interface.

The products of this design phase potentially include CAD drawings, spreadsheets, textual documents and many other artefacts, and clearly a variety of tools can be involved in their production. The management of the status of each of those elements and maintaining traceability between them and subsequent phases can cause a project management headache.

The ideal tools for requirements management depends largely on the scale of the development. If there are few developers in a local office, a simple spreadsheet or Microsoft Word document may suffice. Bigger projects, perhaps with contributors in geographically diverse locations, are likely to benefit from an Application Lifecycle Management (ALM) tool such as IBM Rational DOORS[15], Siemens PLM Polarion ALM[16], or other ALM tools supporting standard Requirements Interchange Format[17].

Part 4 also requires adherence to the principle of bidirectional traceability. That requirement is common to all phases of development, and its automation is discussed later in this document when other related principles have been introduced.

Specification of software safety requirements (part 6, clause 6)

The objectives of the software safety requirements sub-phase are concerned with:

- the specification or refinement of the software safety requirements
- the definition of safety-related functionalities and properties of the software required for the implementation
- the refinement of the requirements of the hardware-software interface
- the verification that the software safety requirements and the hardware-software interface requirements are suitable for software development and are consistent with the technical safety concept and the system architectural design specification

The technical safety requirements are refined and allocated to hardware and software during the technical concept phase as described in part 4, clause 8. The specification of the software safety requirements considers constraints of the hardware and the impact of these constraints on the software, but primarily focuses on the specification of software safety requirements to support the subsequent design phases.

This sub-phase is essentially the interface between the product-wide system design of parts 4 and 6. It details the process of evolution of lower level requirements as they relate specifically to the software system. That implies a continued leveraging of the requirements management tools selected for the project, as discussed in relation to the System Design sub-phase.

Software architectural design (part 6, clause 7)

The objectives of the software architectural design sub-phase are concerned with:

- the development of a software architectural design that satisfies established requirements with the required ASIL
- support for the implementation and verification of the software

There are many tools available for the generation of the software architectural design, with graphical representation of that design an increasingly popular approach. Appropriate tools include those exemplified by IBM Rational Rhapsody[18], MathWorks Simulink[19] and Esterel SCADE[20]. Figure 4 is a modified version of Table 4 reproduced from part 6, showing how the software architectural design is to be verified both at design time and as the design is implemented as development progresses.

[15] http://www-03.ibm.com/software/products/en/ratidoor
[16] https://polarion.plm.automation.siemens.com/
[17] http://www.omg.org/spec/ReqIF/
[18] http://www-03.ibm.com/software/products/en/ratirhapfami/
[19] https://www.mathworks.com/products/simulink.html
[20] http://www.esterel-technologies.com/products/scade-suite/®

Topics		ASIL			
		A	B	C	D
1a	Walkthrough of the design	++	+	o	o
1b	Inspection of the design	+	++	++	++
1c	Simulation of the dynamic parts of the design	+	+	+	+
1d	Prototype generation	o	o	+	+
1e	Formal verification	o	o	+	+
1f	Control flow analysis	+	+	++	++
1g	Data flow analysis	+	+	++	++
1h	Scheduling analysis	+	+	++	++

"++" 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 4: Mapping the capabilities of the LDRA tool suite to "Table 4:Methods for the verification of the software architectural design" specified by ISO 26262-6:2018[21]

Static analysis tools have a part to play in the verification of the design in the form of control and data flow analysis of the code generated in accordance with it. As shown in Figure 5, the tools derive the relationship between some or all of the code components and represent it graphically such that it can then be compared with the intended design.

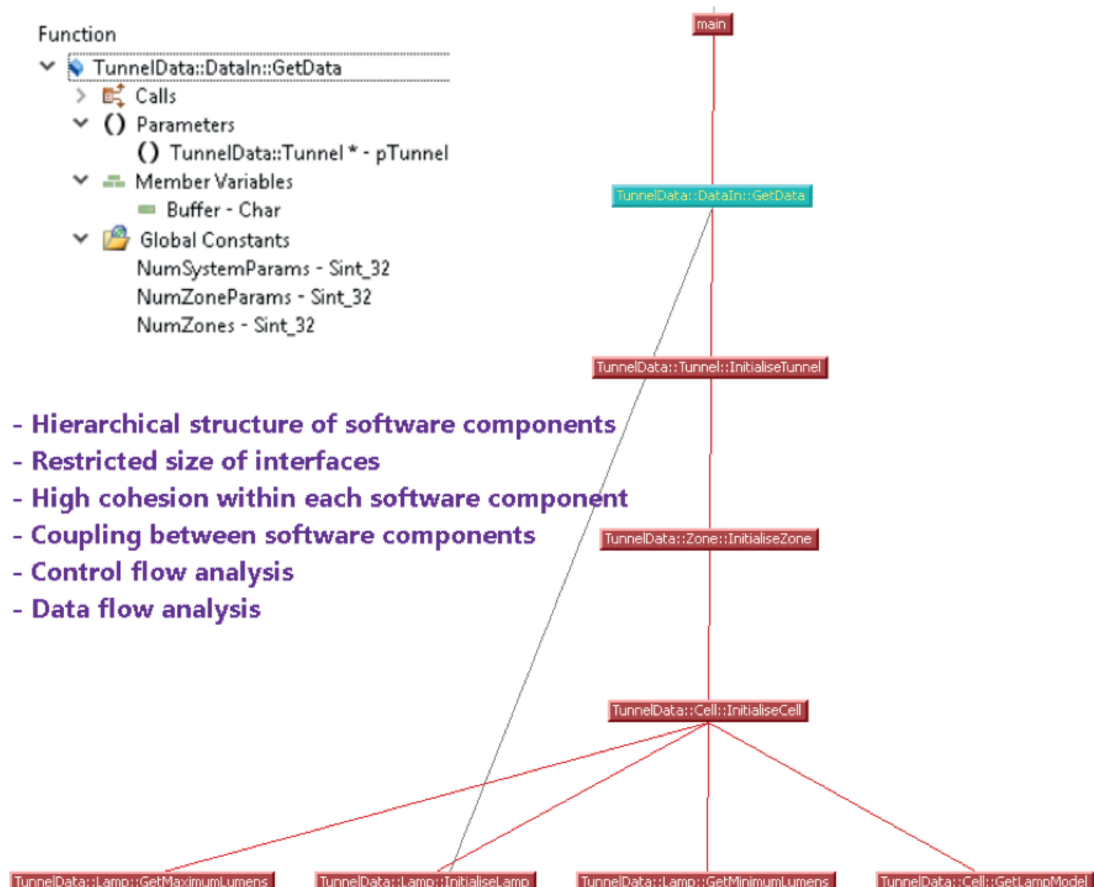

Figure 5: Diagrammatic representations of control and data flow generated from source code by the LDRA tool suite aid verification of software architectural design

[21] Based on table 4 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

LDRA

Reverse engineering

Much of the flow of the ISO 26262 processes is based on an overriding presumption that the project either begins with nothing, or is "proven in use"[22] which implies the deployment of something that has been shown to be reliable after many runtime hours. However, it is entirely possible that there is a need to adapt a long established application which would void that "proven in use" argument, as would the integration of third-party source code into an existing design. What if source code exists, but design documentation doesn't?

The second edition acknowledges this dilemma with the addition of two new clauses in part 8, namely clause 15 "Interfacing an application that is out of scope of ISO 26262" and clause 16 "Integration of safety-related systems not developed according to ISO 26262".

In such circumstances a graphical representation of the code (Figure 5) can help to establish the architecture of the existing system, such that the additions to it can be designed and proven in accordance with ISO 26262 principles.

Model based development

The LDRA tool suite can be integrated with several different model based development tools, one such example being MathWorks Simulink. 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 integration itself is primarily leveraged during software unit testing, and software integration and testing. The topic of model based development is therefore revisited later in this document in relation to those sub-phases.

Software unit design and implementation (part 6, clause 8)

The objectives of the software unit design and implementation subphase are focused upon:

- the development of a software unit design in accordance with the established architectural design, design criteria and requirements
- the implementation of those software units of evidence that software requirements are met

Coding guidelines

There are several aspects of software unit design and implementation to be considered, starting with a definition of guidelines associated with how the selected programming language is to be used. Figure 6 is a modified version of Table 1 reproduced from part 6. Table 1 in the standard shows the coding and modelling guidelines to be enforced during implementation, and it is superimposed here with an illustration of where the LDRA tool suite can confirm compliance, or can assist with the confirmation of compliance.

[22] ISO 26262-8:2018 Road vehicles -- Functional safety -- Part 8: Supporting processes

	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	+	+	++	++
1e	Use of well-trusted design principles	+	+	++	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++
1i	Concurrency aspects	+	+	+	+

"++" 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.
Supported by the LDRA tool suite
Verified by the LDRA tool suite

Figure 6: Mapping the capabilities of the LDRA tool suite to "Table 1: Topics to be covered by modelling and coding guidelines" specified by ISO 26262-6:2018[23]

All of these guidelines can be justified on the basis that they make the resulting code more reliable, less prone to error, easier to test, and/or easier to maintain. For example:

- Low complexity (1a, Figure 6) is important because complex code is less easy to read and maintain, and hence more susceptible to error. For low complexity to be enforced, it needs to be quantified and a "pass/fail" criteria established.
- Language subsets (1b, Figure 6) such as MISRA C restrict the use of a programming language to those elements known to be least susceptible to causing problems.
- Use of style guides (1g, Figure 6) ensures that the code has a consistent appearance no matter who has written it. That makes it much easier to maintain or modify, which in turn makes it less prone to error.

The traditional approach to enforcing adherence to such guidelines would be to use peer code reviews. These may well still have a place in the development process – they can be very useful as an aid to learning between team members, for example – but automating the more tedious checks is far more efficient and less prone to error (Figure 7).

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

[23] Based on table 1 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

Part 6 highlights the MISRA language subsets, but only as an example – not as an instruction that they should be used. There are many different sets of coding rules available (see the sidebar), and even supposing a particular set is chosen as a basis it is entirely permissible to manipulate, adjust and add to it to make it more appropriate for a particular application. Clearly, if a tool is to be useful in such circumstances then it too must be able to accommodate these adjustments.

Software architectural design and unit implementation

If the coding guidelines of part 6 are analogous to the "bricks" of the software application, then its principles for software architectural design define where the "walls" should be built, and its unit implementation guidelines define how those "walls" should be built.

Establishing appropriate project guidelines for coding, architectural design and unit implementation are clearly three discrete tasks but just like a bricklayer building a wall, software developers responsible for implementing the design need to be mindful of them all concurrently. Figure 8 and Figure 9 illustrate how the architectural design and unit implementation principles required by part 6 can usually be checked by means of static analysis, just like coding guidelines.

Language subsets

There are many language subsets (or "coding standards") each with differing attributes but nevertheless with strong similarities, especially when referencing the same language. The most popular standards include:

C

MISRA C:1998
MISRA C:2004
MISRA C:2012
CERT C
CWE

C++

MISRA C++:2008
JSF++ AV
HIC++
CERT C++

Java

CWE
CERT J

Figure 8 is a modified version of Table 3 reproduced from part 6, and it is superimposed here with an illustration of where the LDRA tool suite can assist with the confirmation of compliance.

Principles		ASIL			
		A	B	C	D
1a	Appropriate hierarchical structure	++	++	++	++
1b	Restricted size of software components	++	++	++	++
1c	Restricted size of interfaces	+	+	+	++
1d	Strong cohesion within each software component	+	++	++	++
1e	Loose coupling between software components	+	++	++	++
1f	Appropriate scheduling properties	++	++	++	++
1g	Restricted use of interrupts	+	+	+	++
1h	Appropriate spatial isolation of the software components	+	+	+	++
1i	Appropriate management of shared resources	++	++	++	++

"++" 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.
Supported by the LDRA tool suite

Figure 8: Mapping the capabilities of the LDRA tool suite to "Table 3: Principles for software architectural design" specified by ISO 26262-6:2018[24]

As for the coding guidelines before them, all of these principles are founded on the notion that they make the resulting code more reliable, less prone to error, easier to test and/or easier to maintain.

[24] Based on table 3 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

For example:

- Restricted size of software components (1b, Figure 8) is important not least because large, rambling functions are difficult to read, maintain, and test – and hence more susceptible to error.
- Restricted size of interfaces (1c, Figure 8) is important for similar reasons.
- Strong cohesion within each software component (1d, Figure 8) refers to the level of strength and unity with which different components of a software program are inter-related with each other. 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 is a modified version of Table 6 reproduced from part 6, and it is superimposed here with an illustration of where compliance can be confirmed automatically.

Principles		ASIL			
		A	B	C	D
1a	One entry and one exit point in subprograms and functions	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names	+	+	++	++
1e	Avoid global variables or else justify their usage	+	+	++	++
1f	Restricted use of pointers	o	+	+	++
1g	No implicit type conversions	+	++	++	++
1h	No hidden data flow or control flow	+	++	++	++
1i	No unconditional jumps	++	++	++	++
1j	No recursions	+	+	++	++

"++" 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.
Verified by the LDRA tool suite

Figure 9: Mapping the capabilities of the LDRA tool suite to "Table 6: Design principles for software unit design and implementation" specified by ISO 26262-6:2018[25]

Once again, all of these principles are 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:

- Initialization of variables (1c, Figure 9) references an example of "undefined behaviour" in the C and C++ languages in that different compiler vendors can treat uninitialized variables differently. Some might initialize them to zero, but if developers rely on that and their code is ported to different hardware later in life then it may not behave as expected.
- Avoid global variables or else justify their usage (1e, Figure 9). The problem with global variables is that since every function has access to them, it becomes increasingly hard to figure out which functions actually read and write to them making testing and maintenance difficult. Static analysis can help, but it is much cleaner to avoid them.
- No recursion (1j, Figure 9). Recursive functions are difficult to understand, and it is frequently impossible to predict their likely resource usage.

In general, a fully integrated tool suite can ensure that the good practices required by the standard are adhered to whether they are coding rules, design principles, or principles for software architectural design.

For example, metrics can be generated to ensure that software component size, complexity, cohesion, and coupling are measured and controlled. Complexity metrics are generated as a product of interface analysis, cohesion evaluated through data object analysis, and coupling through data and control coupling analysis (Figure 10).

[25] Based on table 6 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

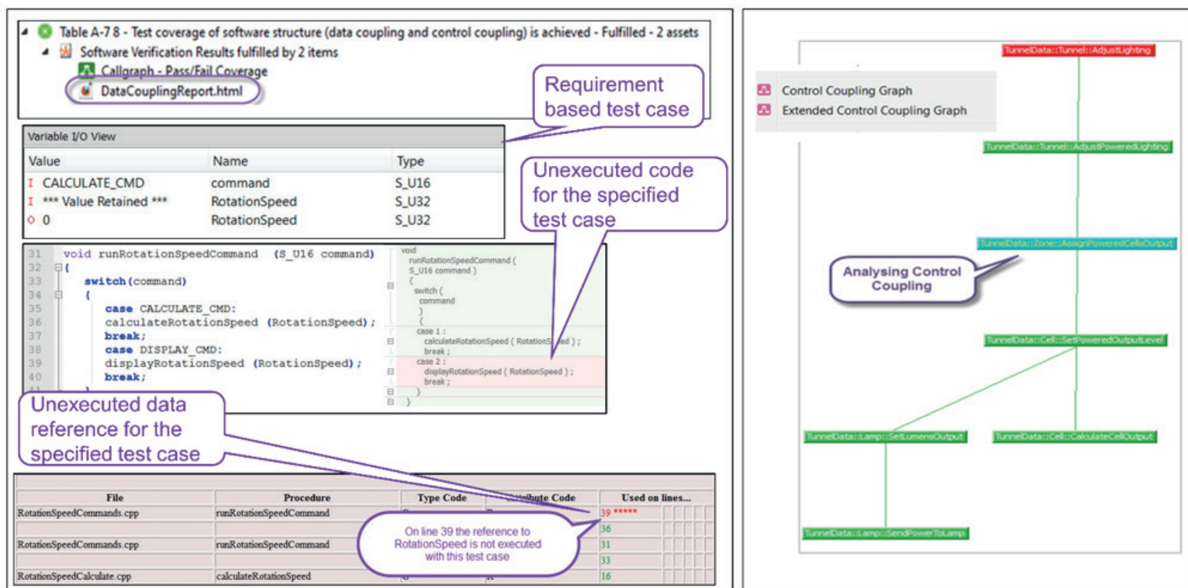

Figure 10: Output from control and data coupling analysis as represented in the LDRA tool suite

The ongoing application of static analysis throughout the code implementation phase provides support and tutelage to the development team. In practice, for developers who are newcomers to ISO 26262, the use of the tool often progresses from pointing out where violations have occurred, to a point where it provides confirmation that there are none.

Software unit verification (part 6, clause 9) and software integration and testing (part 6, clause 10)

The objectives of the software unit verification subphase are focused upon:

- the provision of evidence that software requirements are met
- the verification that defined safety measures are properly implemented
- the provision of evidence that unit design is correctly implemented in accordance with the allotted ASIL
- the provision of evidence that there are no undesirable properties or functionalities

The objectives of the software integration and testing subphase are focused upon:

- the definition and implementation of software elements through to system completion
- the verification that defined safety measures are properly implemented
- the provision of evidence of correct implementation of the software architectural design
- the provision of evidence that there are no undesirable properties or functionalities

The techniques discussed so far in relation to compliance with part 6 have largely focused on static analysis – that is, an automated "inspection" of the source code. Just as static analysis techniques embraced verification of adherence to the part 6 guidelines for coding, architectural design and unit implementation, the dynamic analysis techniques (which involve the execution of some or all of the code) are helpful in both software unit testing, and software integration testing.

Tables 7, 8, 10, and 11 in part 6, list techniques and metrics for performing unit and integration tests, for which a primary function is to ensure that the expected functionality and software interfaces are verified at the unit and integration levels. Software unit and integration tests need to be executed on target hardware and if the developed unit or integrated software is "safety-related", then test results should comply with safety requirements. Fault injection and resource tests help further ensure robustness and resilience. For organizations that apply model-based development, back-to-back testing at the model and code level is recommended.

Figure 11 and Figure 12 are modified versions of Table 7 and 8 respectively, each reproduced from part 6. They are superimposed here with illustrations of where compliance can be confirmed automatically.

Methods		ASIL			
		A	B	C	D
1a	Walk-through	++	+	o	o
1b	Pair-programming	+	+	+	+
1c	Inspection	+	++	++	++
1d	Semi-formal verification	+	+	++	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis	+	+	++	++
1g	Data flow analysis	+	+	++	++
1h	Static code analysis	++	++	++	++
1i	Static code analysis based on abstract interpretation	+	+	+	+
1j	Requirement-based test	++	++	++	++
1k	Interface test	++	++	++	++
1l	Fault injection test	+	+	+	++
1m	Resource usage evaluation	+	+	+	++
1n	Back-to-back test between model and code, if applicable	+	+	++	++

"++" 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.
Supported by the LDRA tool suite
Verified by the LDRA tool suite

Figure 11: Mapping the capabilities of the LDRA tool suite to "Table 7: Methods for the verification of software unit design and implementation" specified by ISO 26262-6:2018[26]

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	+	o	o
1b	Generation and analysis of equivalence classes	+	+	+	+
1c	Analysis of boundary values	+	++	++	++
1d	Error guessing based on knowledge or experience	+	+	++	++

"++" 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.
Supported by the LDRA tool suite
Verified by the LDRA tool suite

Figure 12: Mapping the capabilities of the LDRA tool suite to "Table 8: Methods for deriving test cases for software unit testing" specified by ISO 26262-6:2018[27]

Each developed software unit needs to be tested with reference to the software unit design specification. Test procedures then need to be authored, reviewed, and executed to ensure the software unit does not contain any undesired functionality. Unit tests can then be executed on the target hardware or simulated environment based on the verification plan and verification specification. Once the test procedures are executed, actual outputs are captured and compared with the expected results. Pass/Fail results are then reported and software safety requirements are verified accordingly.

Figure 13 and Figure 14 are modified versions of Tables 10 and 11 respectively, reproduced from part 6. They are superimposed here with illustrations of where compliance can be confirmed automatically.

[26] Based on table 7 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged
[27] Based on table 8 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

Methods		A	B	C	D
		ASIL			
1a	Requirement-based test	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test	+	+	++	++
1d	Resource usage test	++	++	++	++
1e	Back-to-back test between code and model, if applicable	+	+	++	++
1f	Verification of the control flow and data flow	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Static code analysis based on abstract interpretation	++	++	++	++

"++" 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.
Supported by the LDRA tool suite
Verified by the LDRA tool suite

Figure 13: Mapping the capabilities of the LDRA tool suite to "Table 10: Methods for verification of software integration" specified by ISO 26262-6:2018[28]

Methods		A	B	C	D
		ASIL			
1a	Analysis of requirements	++	+	o	o
1b	Generation and analysis of equivalence classes	+	+	+	+
1c	Analysis of boundary values	+	++	++	++
1d	Error guessing based on knowledge or experience	+	+	++	++

"++" 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.
Supported by the LDRA tool suite
Verified by the LDRA tool suite

Figure 14: Mapping the capabilities of the LDRA tool suite to "Table 11: Methods for deriving test cases for software integration testing" specified by ISO 26262-6:2018[29]

Integration testing is designed to ensure that when the units are working together in accordance with the software architectural design, they meet the related specified requirements. In practice, these integration tests typically involve the verification of safety and non-safety related software functions.

It is desirable for all dynamic testing to use environments that correspond closely to the target environment and hence test dependencies between hardware and software. However, that is not always practical. One alternative approach involves developing the tests in a simulated environment and then, once proven, re-running them on the target. Part 6 provides specific guidance on simulation vs target testing:

"*If the software unit testing is not carried out in the target environment, the differences in the source and object code, and the differences between the test environment and the target environment, shall be analysed in order to specify additional tests in the target environment during the subsequent test phases.*"

[28] Based on table 10 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged
[29] Based on table 11 from ISO 26262-6:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

Should changes become necessary – perhaps as a result of a failed dynamic test, or in response to a requirement change - then all impacted unit and integration tests would need to be re-run (regression tested). These regression tests can be automated and systematically re-applied as development progresses to ensure that new functionality does not compromise any that is established and proven.

As for the static analysis of product code, ISO 26262 does not require that any of these tests deploy software test tools. However, once again, such tools are capable of making the test process far more efficient, especially where the project is substantial.

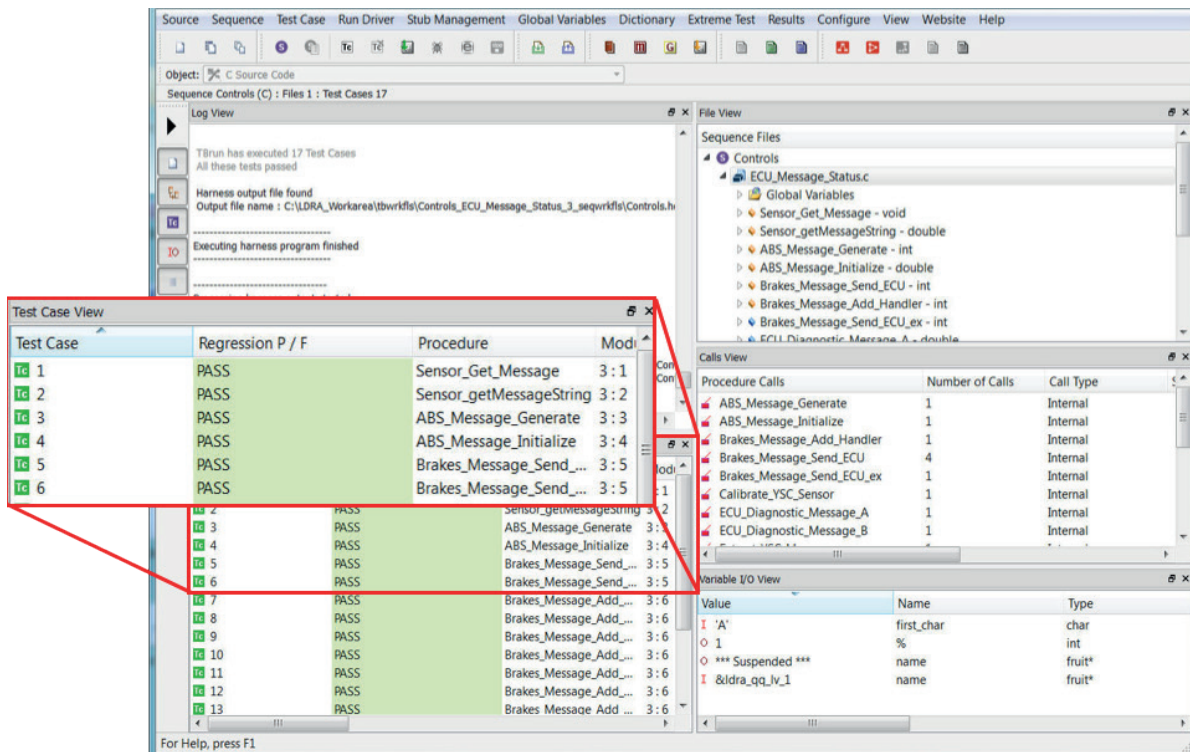

Figure 15: Performing requirement based unit-testing using the LDRA tool suite

The example in Figure 15 shows how the software interface is exposed at the function scope allowing the user to enter inputs and expected outputs. The tool suite then generates a test harness, which is compiled and executed on the target hardware. Actual outputs are captured, along with structural coverage data, and then compared with the expected outputs specified in the test cases.

The inputs and expected outputs defined in the test case are typically derived from requirements Figure 11, 1j) to ensure intended functionality is verified. Various other forms of tests including negative tests, fault injection (Figure 11, 1l), and robustness tests use the same mechanism.

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

The analysis of boundary values can be automated using an "extreme test" facility (Figure 12, 1c) to automatically generate a series of unit test cases. The same facility also provide for the definition of equivalence boundary values such as minimum value, value below lower partition value, lower partition value, upper partition value and value above upper partition boundary. Features like automated stub management, global variable declarations, and exception handling help to complete a comprehensive unit test facility.

Structural coverage metrics

In addition to showing that the software functions correctly, dynamic analysis is also used to generate structural coverage metrics. In tandem with the coverage of requirements at the software unit level, these metrics provide the necessary data to "*reveal shortcomings in requirements-based test cases, inadequacies in requirements, dead code, deactivated code or unintended functionality.*"

Statement, branch and MC/DC coverage can be generated by both the unit test and system activity. Figure 16 is a modified version of Table 9 reproduced from part 6, and it is superimposed here with an illustration of where compliance can be aided by automated test tools.

Topics		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++
"++" 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. Verified by the LDRA tool suite					

Figure 16: Mapping the capabilities of the LDRA tool suite to "Table 9: Structural coverage metrics at the software unit level" specified by ISO 26262-6:2018[30]

These packages can also operate in tandem, so that (for instance) coverage can be generated for most of the source code through a dynamic system test and that can be complemented using unit tests to exercise constructs inaccessible during normal system operation, such as defensive constructs (Figure 17).

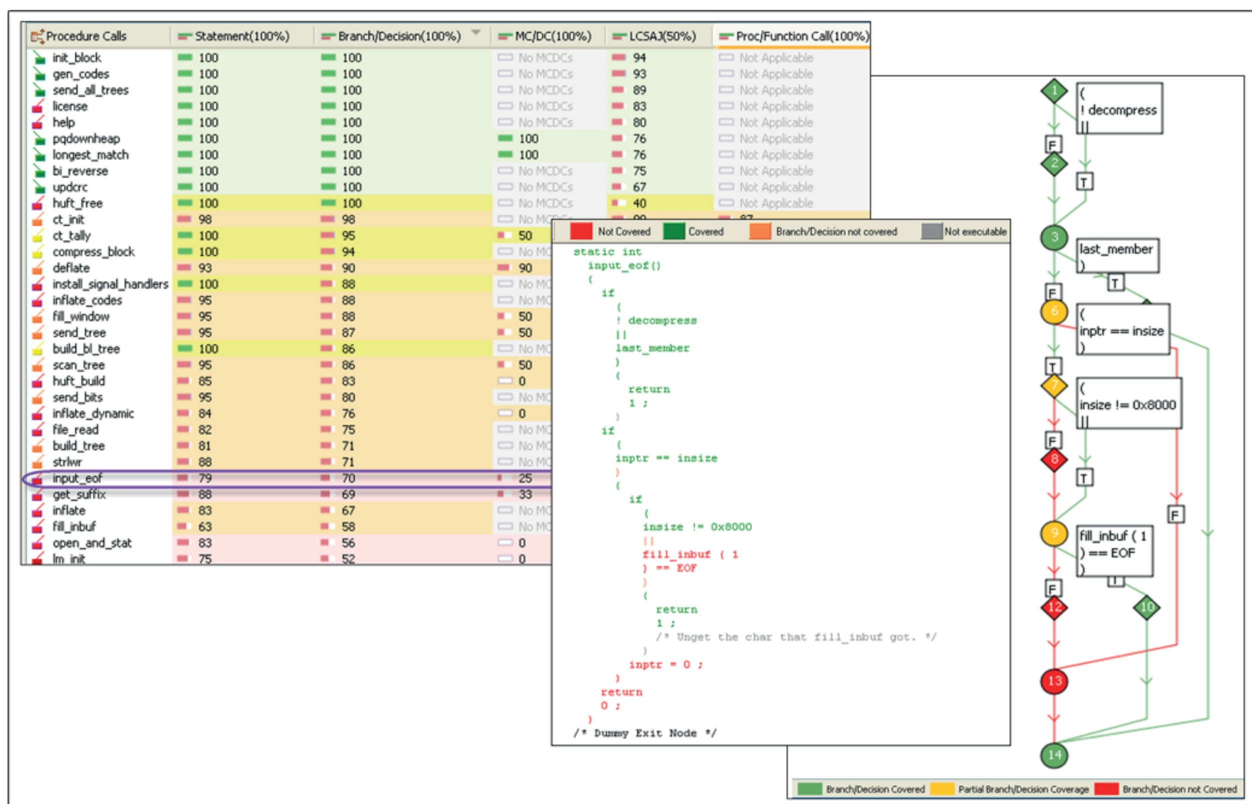

Figure 17: Examples of representations of structural coverage within the LDRA tool suite

The generation of functional and call coverage data can also be automated by executing tests on the target. Figure 18 is a modified version of Table 12 reproduced from part 6, which relates to integration testing. It is superimposed here with an illustration of where compliance can be confirmed automatically.

Coverage data also forms one part of the mechanism that allows control coupling analysis to be supported by the LDRA tool suite, as discussed earlier (Figure 10). Possible issues are highlighted during static analysis, and warning messages raised during dynamic analysis if the actual linkage is different from the predicted linkage. Graphical representations are then explored to resolve inconsistencies.

[30] Based on table 9 from ISO 26262-6:2011, Copyright © The British Standards Institution 2018. All rights acknowledged

Topics		ASIL			
		A	B	C	D
1a	Function coverage	+	+	++	++
1b	Call coverage	+	+	++	++

"++" 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.
Verified by the LDRA tool suite

Figure 18: Mapping the capabilities of the LDRA tool suite to the "Table 12: Structural coverage metrics at the software architectural level" specified by ISO 26262-6:2018[31]

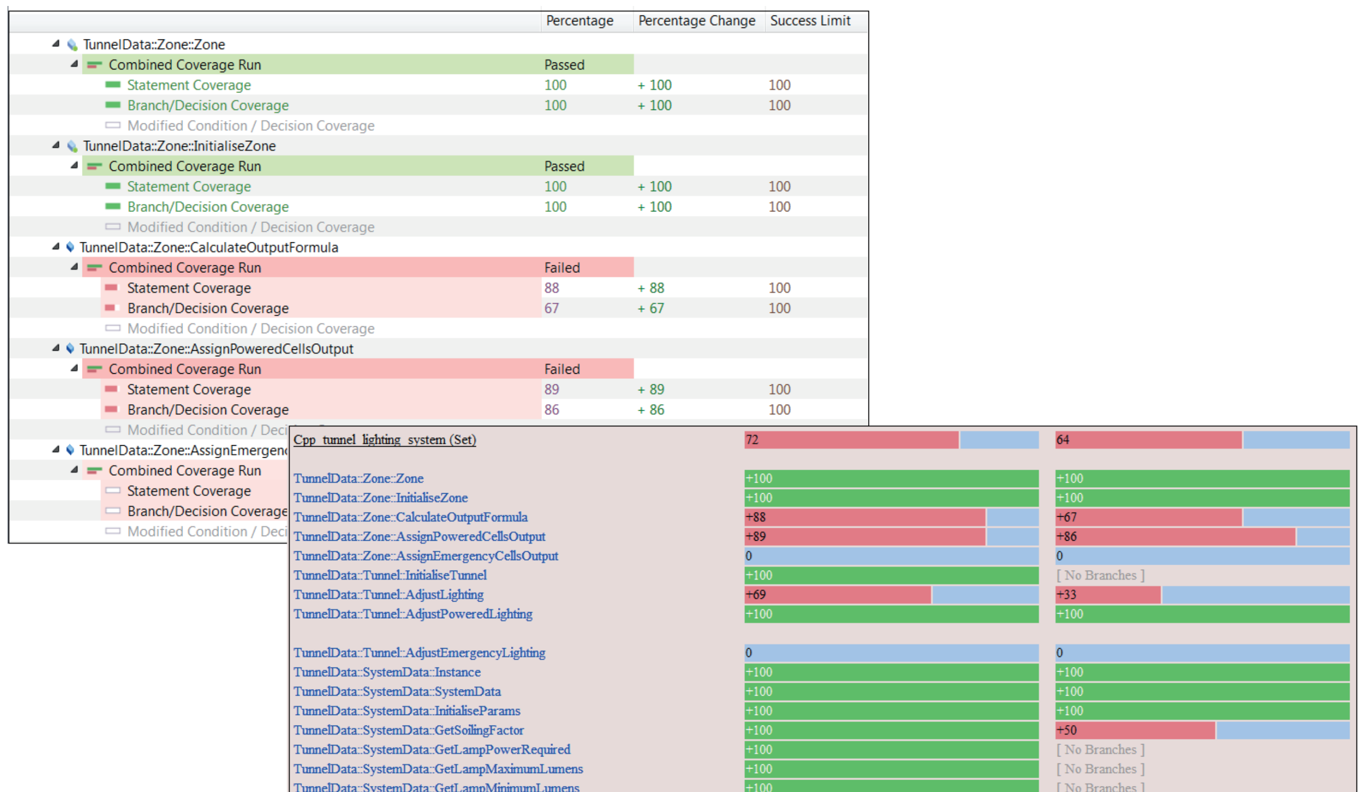

Figure 19: Examples of representations of function and call coverage within the LDRA tool suite

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, MathWorks Simulink, and ANSYS Scade. 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.

Model based development offers many advantages to developers of automotive software and many modelling tools include integrated model and auto-generated code testing features. However, ISO 26262 suggests that "*In comparison to a traditional development process where lifecycle data are separated, a stronger coalescence of the phases ... may occur. The potential benefits of this approach ... are appealing, but this approach may also introduce issues causing systematic faults*". An automated approach to testing that is integrated with the modelling tool and yet independent from it, helps to offset those concerns.

[31] Based on table 12 from ISO 26262-6:2018, © The British Standards Institution 2018 . All rights acknowledged

Figure 20 illustrates one example of how an integration with Simulink can be deployed using an approach appropriate for use with "Back-to-back" testing (Figure 11, 1n and Figure 13, 1e). 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.

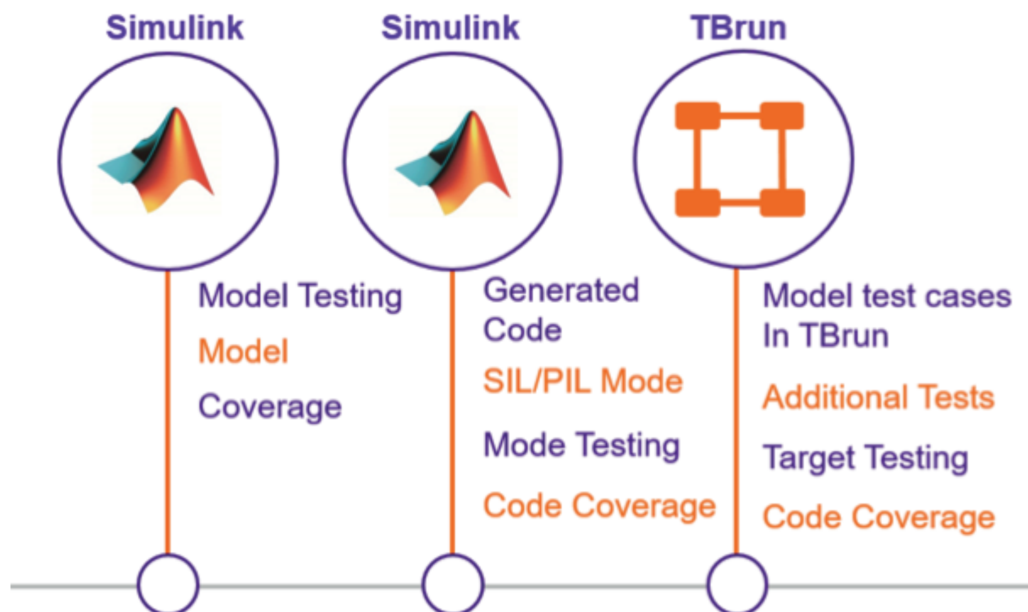

Figure 20: Generating structural coverage data of auto generated code with MathWorks Simulink and the LDRA tool suite

The generated source code can be analysed statically to ensure compliance with an appropriate coding standard, such as MISRA C:2012 Appendix E[32]. 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.

Bidirectional traceability (parts 4 and 6)

Bidirectional traceability is referenced throughout ISO 26262. It is explicitly mentioned in part 6 paragraph 7.4.2:

"*This implies bi-directional traceability between the software architectural design and the software safety requirements.*"

More generally though, its presence is implied by the need for each development phase to accurately reflect the one before it, such as in part 6 paragraph 9.1, which requires that all requirements are fulfilled with no additional undesirable properties:

"c) to provide evidence that the implemented software unit complies with the unit design and fulfils the allocated software requirements with the required ASIL; and
d) to provide sufficient evidence that the software unit contains neither undesired functionalities nor undesired properties regarding functional safety."

[32] https://www.misra.org.uk/tabid/72/Default.aspx

In theory, bidirectional traceability is easy to achieve. 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 unfortunately, challenges do arise during the course of a project.

Consider what happens if an integration test fails.

- It may fail because there is a contradiction in the requirements. If that is the case, the requirements will need to change. But what other parts of the software are affected by that?
- It may fail because there is a coding error. If that is the case, then it will need to be corrected. But what other software units were dependent on the modified code? What if they were dependent on an incorrect output?
- It may fail because the requirements have an incorrect specification for the test parameters. That means a requirement change. But have there been unit tests which are compromised because these parameters were wrong?

These scenarios can quickly lead to situations where the traceability between the products of software development falls down. While it is possible to perform the tasks of maintaining it manually, automation is likely to help a great deal.

Software unit design can take many forms and can leverage natural language or model based approaches. Either way, these design elements need to be bidirectionally 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 (Figure 21).

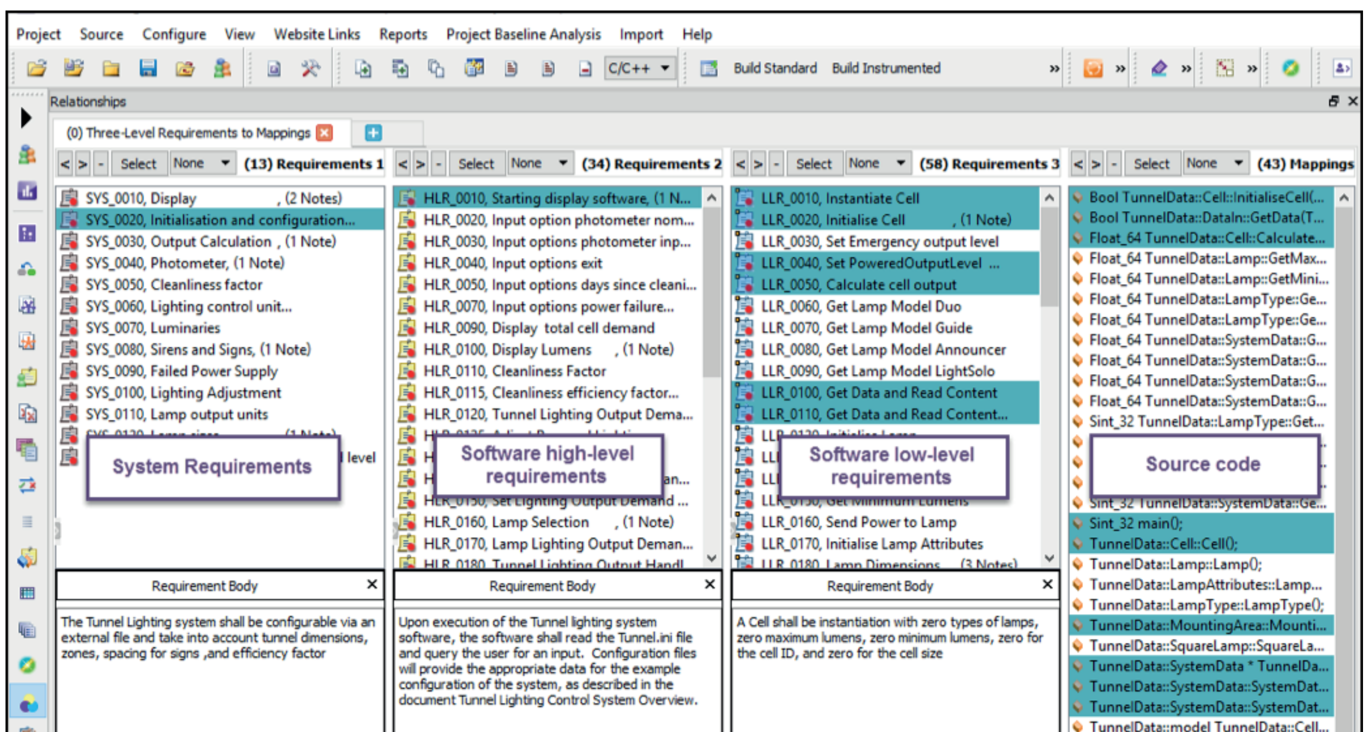

Figure 21: Traceability in the LDRA tool suite. The example detailed design is linked upstream to software safety requirements and downstream to software units.

Figure 22 shows how to establish a traceability policy can be established between requirements and tests cases of different scopes, which allows test coverage to be assessed. Test cases are reviewed and developed based on requirements, and gaps in requirements coverage can be quickly assessed and filled.

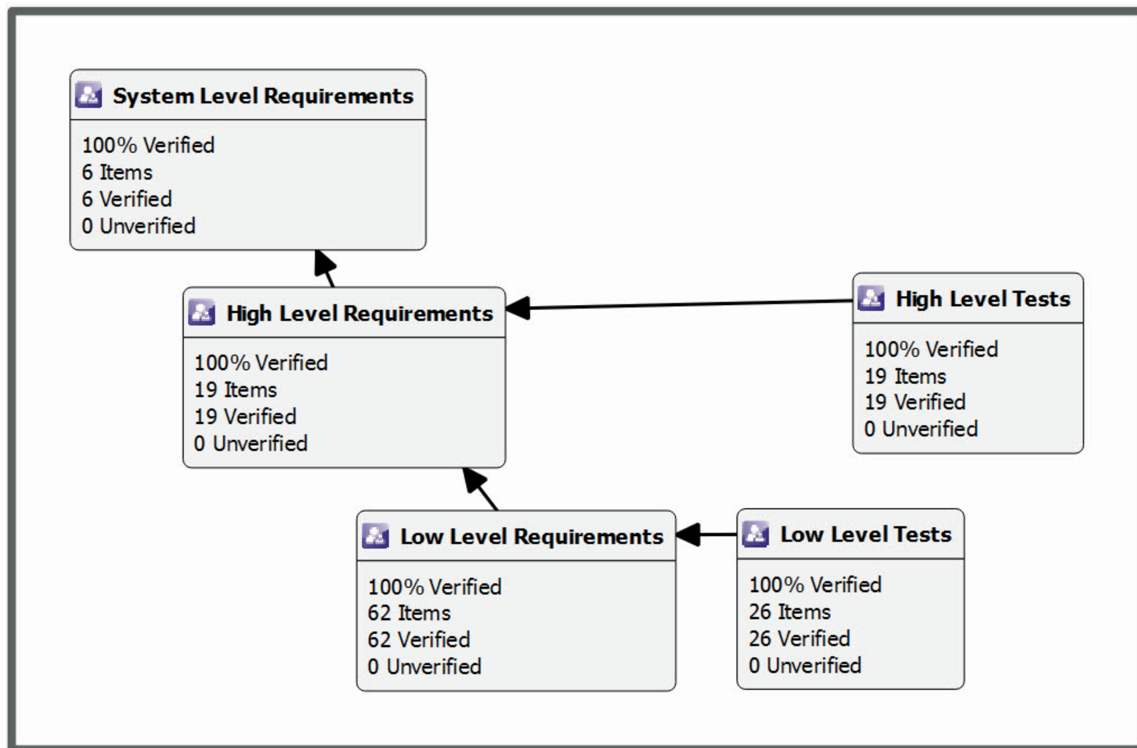

Figure 22: Performing requirement based testing. Test cases are linked to requirements and executed within the LDRA tool suite

Conversely, bidirectional analysis can also determine test cases that are not linked to requirements, and prompt for their correction. Additionally, it provides a mechanism for impact analysis with reference to changes in requirements, tests, or code, and identifies where regression testing needs to be targeted in order to rework overhead. It also provides a mechanism for the generation of artefacts such as traceability matrices to present evidence of compliance to.

LDRA TBmanager Test Case Traceability Matrix - High Level Tests -> High Level Requirements

Project	C:\Ldra\Demos\Tunnel_5wc\trunk\DO178\Tunnel_5.tbp	Date	09/07/16 14:32:21	Version	9.5.7

Summary Show/Hide

High Level Requirements Items Covered by High Level Tests Items: 33/34 (97%)

High Level Tests Items Covering High Level Requirements Items: 33/34 (97%)

High Level Requirements Traceability Table Show/Hide

Number/Name	Text	Covered	Number/Name	Text
High Level Requirements			High Level Tests	
HLR_0090, Display total cell demand	In the nominal power state, after entering a nominal range photometer input or nominal days since cleaning, the software shall display Total Cell demand and lumens per metre	Yes	TCI_0070	The tunnel software output stream will provide the total cell demand and lumens per meter.
HLR_0231, Lamps maximum output	A lamp shall provide an output of 120lm/W when used at the maximum output more text	Yes	TCI_0250	The Tunnel software output produced to drive maximum lm/W levels will generate output levels no greater than 120 lm/w
HLR_0125, Adjust Powered Lighting	Given the photometer input lighting shall be calculated across all zones	Yes	TCI_0120	The Tunnel software output produced from photometer inputs will show calculations for all zones
HLR_0340, System Data Initialisation , (1 Note)	All software data shall be stored for management, tracking, and reporting	Yes	TCI_0320	The Tunnel software output produced from photometer inputs, given a set of input configuration files, verified against expected output files will verify that data management capabilities of the software are being met

Figure 23: Extract from an LDRA tool suite traceability matrix (high-level requirements to tests cases)

Figure 23 shows a traceability matrix between high-level requirements and functional test cases. In this example, there is one requirement which has no associated test case, and so the objective of test coverage of high-level requirements is unfulfilled. A similarly transparent user experience is available for all facets of test coverage analysis. By combining them with unit test capabilities, it offers an accessible mechanism for ensuring traceability throughout the development lifecycle, and particularly during software unit and integration testing.

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; ultimately resulting in the addition or modifications of test cases, changes to requirements, and/or the removal of dead code. An iterative "review, correct and analyse" sequence is typically needed to ensure that design specifications are satisfied.

Confidence in the use of software tools (part 8)

This ISO 26262 supporting process defines a mechanism to provide evidence that the software tool chain can be relied upon. The required level of confidence in a software tool depends upon the circumstances of its deployment, with particular reference to:

- the possibility that the malfunctioning software tool and its corresponding erroneous output can introduce or fail to detect errors in a safety-related item or element being developed, and
- the confidence in preventing or detecting such errors in its corresponding output.

The LDRA tool suite has been qualified for use with ISO 26262 up to ASIL D, which removes considerable user overhead in providing evidence of that confidence (Figure 24).

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

CERTIFICATE

No. Z10 084753 0006 Rev. 00

Holder of Certificate: LDRA Ltd.
Portside, Monks Ferry
Wirral, Merseyside CH41 5LH
UNITED KINGDOM

Factory(ies): 084753

Certification Mark:

Product: Software Tool for Safety Related Development

Model(s): LDRA tool suite
LDRArules
LDRAcover
LDRAunit
LDRAlite

Parameters: The certified tools fulfills the requirements for support tools classified T2 according to IEC 61508-3 and EN 50128. The tools are qualified to be used in safety-related software development according to IEC 61508, EN 50128 and ISO 26262. It is suitably validated for use in safety-related development according to IEC 62304. The test report is a mandatory part of this certificate.

Tested according to: EN 50128:2011
IEC 61508-3:2010
IEC 62304:2006
IEC 62304:2006/AMD1:2015
ISO 26262-8:2018

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. See also notes overleaf.

Test report no.: LW85043C

Valid until: 2024-08-15

Date, 2019-08-16 (Peter Weiß)

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

A4 / 07.17

SGS TÜV SAAR

CERTIFICATE NO FS/71/220/19/0403 PAGE 1/1
ZERTIFIKAT NR.: SEITE(N)

LICENCE HOLDER GENEHMIGUNGSINHABER	MANUFACTURING PLANT FERTIGUNGSSTÄTTE
LDRA LTD Portside, Monks Ferry, Wirral, CH41 5LH, United Kingdom	LDRA LTD Portside, Monks Ferry, Wirral, CH41 5LH, United Kingdom

PROJECT NO/-ID PROJEKT-NR/-ID	LICENSED TEST MARK GENEHMIGTES PRÜFZEICHEN	CERT. REPORT NO. ZERTIFIKATSBERICHT NR.
M15F-AU04	SGS TÜV SAAR FUNKTIONALE SICHERHEIT GEPRÜFT FUNCTIONAL SAFETY APPROVED	M15F0005

Tested according to (Geprüft nach): IEC 61508:2010; clause 7.4.4
ISO 26262:2018; clause 11.4.9
EN 50128:2011
IEC 60880:2006
IEC 62304:2015

Certified product(s) (Zertifizierte(s) Produkt(e)): LDRA tool suite
LDRAunit
LDRArules
LDRAcover
LDRAlite

Model(s) (Modell(e)): Version 9.7.4
Version 9.7.5
Version 9.7.6
Version 9.7.7

Technical Data and Parameter (Technische Daten und Parameter): Suitable for development of safety-related software acc. to:
- ISO 26262 up to ASIL D,
- IEC 61508 up to SIL 4, class T2 tool
- EN 50128 up to SIL 4, class T2 tool
- IEC 62304 up to SW safety Class C
- IEC 60880 for category A functions

Specific Requirements (Spezifische Anforderungen): Any changes to the design, components or processing may require repetition of some parts of the pre-qualification to retain the certification. The certificate report is an integral part of this certificate.

Certification Body
for Functional Safety
SGS-TÜV Saar GmbH
Zertifizierungsstelle für Funktionale Sicherheit

Munich, 05.07.2019
Gudrun Neumann

The test mark regulation is an integral part of this certificate.
Die Prüf- und Zertifizierordnung ist integraler Bestandteil des Zertifikates.

SGS-TÜV Saar GmbH, Hofmannstr. 50, 81379 München, Germany

Website www.sgs-tuv-saar.com E-mail fs@sgs.com

CERTIFICATE

Figure 24: Evidence of TUV approved qualification of the LDRA tool suite

It is classified as a TI2 (Tool Impact - 2) category tool because although a verification tool can fail to detect existing errors in source code, unlike (say) an auto code generator it cannot introduce errors into an application.

The TI2 classification dictates that the organization using the tool suite in the development of a compliant application, is required to estimate the likelihood of them detecting any error introduced by the tool (Tool Error Detection), with categories ranging from TD1 (high confidence) to TD3. Evidence of the reasoning behind that assessment is required.

		Tool Error Detection		
		TD1	TD2	TD3
Tool Impact	T1	TCL1	TCL1	TCL1
	T2	TCL1	TCL2	TCL3

Figure 25: Mapping the characteristics of the LDRA tool suite to "Table 3: Determination of the tool confidence level" from ISO 26262-8:2018[33]

In turn, the TCL (Tool Confidence Level) is then derived from a look-up table as shown in Figure 25. Depending on the user's assessment of the application, the resulting TCL will therefore be either TCL1 or TCL2 for the LDRA tool suite.

In all cases except where the tool suite is assigned TCL2 and the product is designated ASIL D, the existence of a TUV certificate (Figure 24) is sufficient to establish confidence in the tool.

	Methods	**ASIL**			
		A	**B**	**C**	**D**
1a	Increased confidence from use in accordance with 11.4.7	++	++	++	+
1b	Evaluation of the tool development process in accordance with 11.4.8	++	++	++	+
1c	Validation of the software tool in accordance with 11.4.9	+	+	+	++
1d	Development in accordance with a safety standard	+	+	+	++

"++" The method is highly recommended for this ASIL.
"+" The method is recommended for this ASIL.
Applying TCL2 designation

Figure 26: Assessing the need for the qualification the LDRA tool suite when it is designated TCL2 from "Table 5: Qualification of software tools classified TCL2" from ISO 26262-8:2018[34]

Otherwise, the tool is required to be subjected to a validation process (Figure 26). showing that the tool is capable of analysing sample software in the target environment. In this example case, a Tool Qualification Support Package (TQSP) is available from LDRA to provide that sample software.

Conclusions

The explosion in the quantity and complexity of automotive software is well documented. ISO 26262 in its latest second edition form, fine tunes the sound foundations established by the first edition version and extends it, not only to include other types of vehicle, but also to acknowledge the increasing impact of cybersecurity on the development of automotive software.

There is clearly an incentive to minimise the ASIL applied to each element of a system, because of the reduced cost involved in achieving it. However, the whole principle of the assignment of ASILs for various automotive systems implies an assumption of separation, such that the most critical systems on a vehicle cannot be compromised by less critical functionality elsewhere. For a connected vehicle to be considered safe and compliant with the principles of ISO 26262, it is therefore imperative that attack surfaces are minimized, and that separation between systems is optimized.

With that assurance in place, the principle of ASIL decomposition can help minimize the overheads associated with functionally safe development that even now are relatively new to the automotive sector. Tools such as those provided by LDRA are well proven in other safety critical sectors, making them not only ideally placed to further optimize the route to compliance, but also better established, than the standard itself!

[33] Based on table 3 from ISO 26262-8:2018, Copyright © 2018 IEC, Geneva, Switzerland. All rights acknowledged
[34] Based on table 5 from ISO 26262-8:2018, Copyright © The British Standards Institution 2018. All rights acknowledged

Works Cited

"MISRA C:2012 - Guidelines for use of the C language in critical systems" ISBN 978-906400-11-8 (PDF), March 2013

"RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification", Prepared by SC-205, December 13, 2011

"IEC 61508:1998 & 2000, Functional safety of electrical/electronic/programmable electronic safety-related systems" IEC, Geneva, Switzerland

"ISO 26262:2011 Road vehicles – Functional safety". The British Standards Institution

"ISO 26262-4:2018 Road vehicles – Functional safety – Part 4: Product development at the system level". The British Standards Institution

ISO 26262-5:2018 Road vehicles – Functional safety – Part 5: Product development at the hardware level". The British Standards Institution

"ISO 26262-6:2018 Road vehicles – Functional safety – Part 6: Product development at the software level". The British Standards Institution

"ISO 26262-8:2018 Road vehicles – Functional safety – Part 8: Supporting Processes". The British Standards Institution 2018

"ISO 26262-11:2018 Road vehicles – Functional safety – Part 11: Guidelines on application of ISO 26262 to semiconductors." The British Standards Institution

"ISO 26262-12:2018 Road vehicles – Functional safety – Part 12: Adaptation of ISO 26262 for motorcycles." The British Standards Institution

"ISO/SAE CD 21434 Road Vehicles – Cybersecurity engineering", ISO/SAE, under development
https://www.iso.org/standard/70918.html

"Experimental Security Analysis of a Modern Automobile", Karl Koscher, Stephen Checkoway et.al. From 2010 IEEE Symposium on Security and Privacy
http://www.autosec.org/

"Remote Exploitation of an Unaltered Passenger Vehicle", Dr. Charlie Miller & Chris Valasek, August 2015
http://illmatics.com/Remote%20Car%20Hacking.pdf

"SAE J3061-2016 Cybersecurity Guidebook For Cyber-Physical Vehicle Systems", Society of Automotive Engineers, 2016
https://webstore.ansi.org/standards/sae/sae30612016j3061

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, 3rd Floor, 12th Main Lewisville Texas 75056
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit B-3, Third floor Tower B, Golden Enclave
HAL Airport Road Bengaluru 560017
Tel: +91 80 4080 8707
e-mail: india@ldra.com

Implementing ISO 26262 2nd ed. with the LDRA tool suite® v2.0 08/19