Technical White Paper

Implementing IEC 61508:2010 with the LDRA tool suite®

Working with the programmable electronic components sector to achieve functional safety

www.ldra.com

Contents

Introduction

With recent advances in automation, software is no longer a bit-part contributor to electro-mechanical systems but is now the underlying technology providing functional safety for products in many market segments. The requirement for software functional safety has therefore become a critical topic in industrial automation, transportation, nuclear energy generation and other markets. IEC 61508:2010 "*Functional safety of electrical/electronic/programmable electronic safety-related systems*" is widely accepted as a reference standard. It also forms the basis for sector-specific standards.

The IEC 61508 standard describes a risk-based approach for determining the SIL (Safety Integrity Level) of safety instrumented functions. If computer system technology is to be effectively and safely exploited, it is essential that the available guidance on these safety related aspects is adequate to allow the correct decisions to be made.

It is recognised that there is a great variety of applications using Electrical/Electronic/Programmable Electronic (E/E/PE) safety-related systems in a range of application sectors, covering a wide range of complexity, hazard and risk potentials. The required safety measures for each particular application will be dependent on many factors specific to it. The generic nature of IEC 61508 makes it an ideal "blank canvas" for the seamless integration of these application dependent factors and hence the derivation of industry and sector specific standards.

In many situations, safety is achieved by a number of systems which rely on many engineering disciplines, including mechanical, hydraulic, pneumatic, electrical, electronic, and programmable electronic technologies. Any safety strategy must therefore consider not only all the elements within an individual system (for example sensors, controlling devices and actuators) but also all the safety-related subsystems which contribute to the safety-related system as a whole. Therefore, although IEC 61508 is concerned with E/E/PE safety-related systems, it also serves as a framework within which safety-related systems based on other technologies may additionally be considered.

This white paper describes the key software development and verification process requirements of the IEC 61508 standard and how automated tools such as the LDRA tool suite® and its component parts can assist with meeting them. In general, it is laid out to reflect the flow of the V-model referenced by the standard, but IEC 61508 uses sets of annexed tables to identify particular techniques to be applied. These techniques often apply to different stages of the lifecycle, which makes them difficult to integrate into the narrative illustrated by the "*V*" model especially when they are sub-referenced. For that reason, descriptions of the annex B tables are held in an appendix.

Safety Integrity Levels

Embedded software developers will be primarily concerned with part 3 of IEC 61508:2010[1], "*Software Requirements*". However, the level of effort required to complete each objective in the standard is dependent on the Safety Integrity Level (or "*SIL*") of the safety functions implemented by the system. The derivation of the SIL is covered in more detail in part 5[2] of the standard, "*Examples of methods for the determination of safety integrity levels*".

Annex A of that standard discusses the concept of "*Necessary risk reduction*"[3] and describes it as being "*the reduction in risk that has to be achieved to meet the tolerable risk for a specific situation*". Tolerable risk is dependent on such as the severity of injury, the number of people exposed to danger, and the frequency and duration of that exposure. It is derived by taking due consideration of inputs such as legal requirements, safety authority guidelines, and discussions with interested parties.

The standard goes on to define Safety Integrity as "... *the probability of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time*"[4] and subdivides it into "*Hardware Safety Integrity*" and "*Systematic Safety Integrity*". The latter is the primary concern for software applications.

[1] IEC 61508:2010-3, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software Requirements

[2] IEC 61508:2010-5, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 5: Examples of methods for the determination of safety integrity levels

[3] IEC 61508:2010-5, Annex A, Section A.2, "Necessary risk reduction"

[4] IEC 61508:2010-5, Annex A, Section A.4 - Safety Integrity

The SIL assigned to each safety function therefore depends the probability of failure, which can be derived in several different ways. The higher the probability of failure, the higher the SIL (from SIL 1 to SIL 4), and the more demanding the overheads on software development to make the risk acceptable.

SIL	Low demand mode: average probability of failure on demand	Nigh demend or contiuous mode: probability of dangerous failure per hour
1	$\geq 10^{-2}$ to $< 10^{-1}$	$\geq 10^{-6}$ to $< 10^{-5}$
2	$\geq 10^{-3}$ to $< 10^{-2}$	$\geq 10^{-7}$ to $< 10^{-6}$
3	$\geq 10^{-4}$ to $< 10^{-3}$	$\geq 10^{-8}$ to $< 10^{-7}$ (1 dangerous failure in 1140 years)
4	$\geq 10^{-5}$ to $< 10^{-4}$	$\geq 10^{-9}$ to $< 10^{-8}$

Figure 1: Deriving the SIL of a safety function from the probability of failure

The Software Development Lifecycle

According to the introduction, IEC 61508 "*… sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic (E/E/PE) elements that are used to perform safety functions.*" Figure 2 shows the V-model illustration from the standard, superimposed with an illustration of how the LDRA tool suite and other complementary tools can be applied within the process.

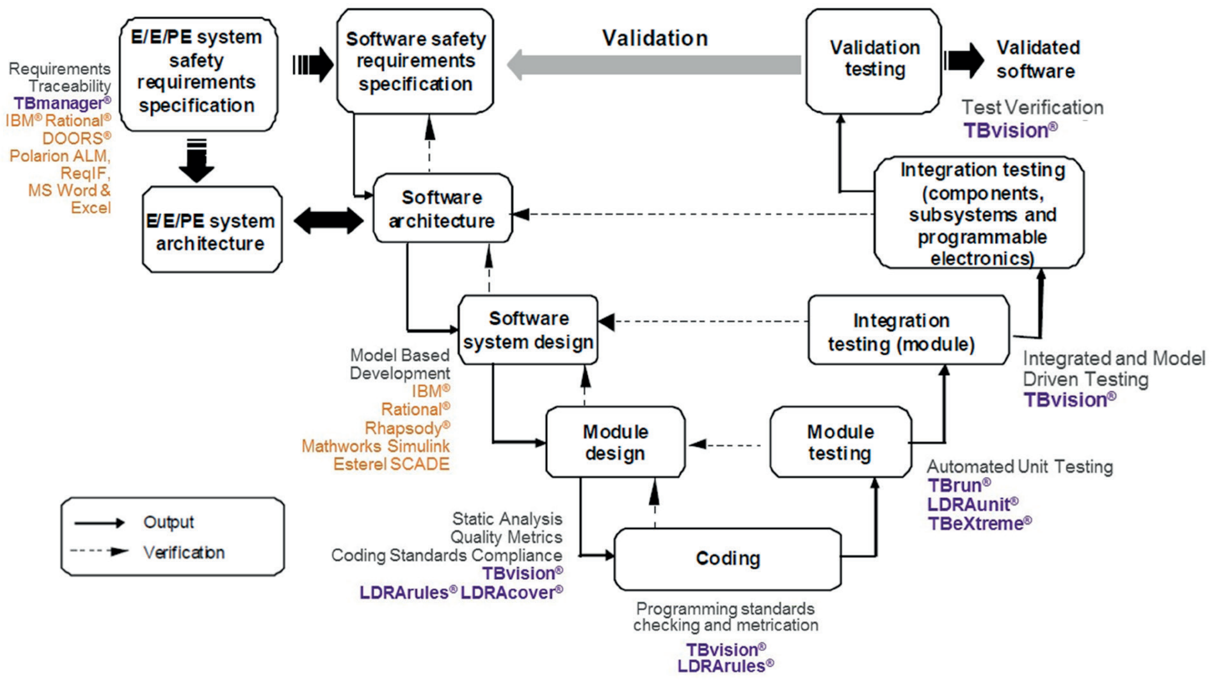


Figure 2: Mapping the capabilities of the LDRA tool suite and complementary tools to the IEC 61508:2010 development lifecycle (the V-model)[5]

IEC 61508 is not only a stand-alone standard. It also forms the basis for complete, industry-specific derivative standards such as ISO 26262[6] for the automotive industry, and is also frequently referenced piecemeal when its generic objectives are applicable to more narrowly defined sectors. One such example is the IEC 13849:2015 for control system software, which defers to the development cycle of IEC 61508 for the most critical applications it describes.

Part 3 of IEC 61508, "*Software Safety Lifecycle Requirements*", structures the development of the software in defined phases and activities to reflect and subdivide the phases illustrated in this V-model diagram. These are used as the basis for the layout of this white paper, and comprise:

[5] Based on IEC 61508:2010-3 Figure 6 – Software systematic capability and the development lifecycle (the V-model)
[6] ISO 26262:2011, Road vehicles – Functional safety

Section 7.2 Software safety requirements specification
Section 7.3 Validation plan for software aspects of system safety
Section 7.4 Software design and development
    Section 7.4.3 Requirements for software architecture design
    Section 7.4.4 Requirements for support tools, including programming languages
    Section 7.4.5 Requirements for detailed design and development – software system design
    Section 7.4.6 Requirements for code implementation
    Section 7.4.7 Requirements for software module testing
    Section 7.4.8 Requirements for software integration testing
Section 7.5 Programmable electronics integration (hardware and software)
Section 7.6 Software operation and modification procedures
Section 7.7 Software aspects of system safety validation
Section 7.8 Software modification
Section 7.9 Software verification
Section 8 Functional safety assessment

There are also 7 Annexes defined in IEC 61508:2010-3 which are referenced by the main body of the standard. Annex A is discussed in the body of this white paper, and Annex B is considered in an appendix.

IEC 61508:2010-3 Section 7.2: "*Software safety requirements specification*"

The first phase illustrated in the IEC 61508:2010 V-model concerns the definition of a software safety requirements specification. Section 7.2 highlights the objectives associated with the specification of software safety requirements. These include the derivation of requirements for the software safety functions, the software systematic capability, and the implementation of the required safety functions.

The V-model illustrates the need for each step in the process to be traceable to the next, as implied by the verification arrows during the lifecycle, and the validation step at its end. Bi-directional traceability is specified as an explicit objective in the Annex A.1 table (Figure 3) which is typical of the tables used extensively in the standard.

Achieving a format that lends itself to bi-directional traceability will help to achieve compliance with the standard. Bigger projects, perhaps with contributors in geographically diverse locations, are likely to benefit from an application lifecycle management tool such as IBM® Rational® DOORS®[7], Siemens® Polarion® PLM®[8], or more generally, similar tools offering support for standard Requirements Interchange Formats[9]. Smaller projects can cope admirably with carefully worded Microsoft® Word® or Microsoft® Excel® documents, written to facilitate links up and down the development process model.

	Technique/Measure	Ref	SIL			
			1	2	3	4
1a	Semi-formal methods	Table B.7	R	R	HR	HR
1b	Formal methods	B.2.2, C.2.4	---	R	R	HR
2	Forward traceability between the system safety requirements and the software safety requirements	C.2.11	R	R	HR	HR
3	Backward traceability between the safety requirements and the perceived safety needs	C.2.11	R	R	HR	HR
4	Computer-aided specification tools to support appropriate techniques/measures above	B.2.4	R	R	HR	HR
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 3: Copy of IEC 61508-3 Table A.1[10], with static analysis techniques supported by the LDRA static analysis tools highlighted. Note the specific requirement for bi-directional traceability.

[7] http://www-03.ibm.com/software/products/en/ratidoor
[8] https://polarion.plm.automation.siemens.com/
[9] http://www.omg.org/spec/ReqIF/
[10] IEC 61508-3 Annex A Table A.1 – Software safety requirements specification

Bi-Directional Traceability

It would be easy to dismiss the task of tracing between the development lifecycle phases as trivial, but their combined effect on project management overhead can be significant.

For instance, consider an unexpected change of requirement imposed by a customer. What is impacted? Which requirements? What elements of the code design? What code needs to be revised? And which parts of the software will require re-testing?

The most effective way to ensure that the project is not thrown off course by such eventualities is to maintain Bi-directional Traceability of Requirements[11] Figure 4). In this diagram, when requirements are managed well, traceability can be established forwards from the outline requirements, through the detailed requirements and on to the work products – and they can similarly be traced backwards. Such bi-directional traceability helps determine that all source requirements have been completely addressed and that all lower level requirements can be traced to a valid source, and that there is no spurious source code that is surplus to requirements. Requirements traceability can also cover the relationships to other entities such as intermediate and final work products, changes in design documentation, and test plans.

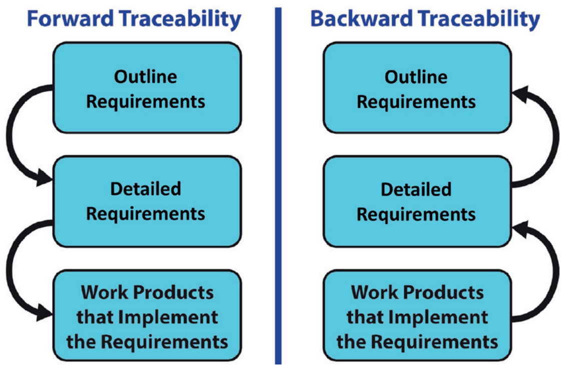


Figure 4: An Illustration of the principles of Bi-directional Traceability

Requirements rarely remain unchanged throughout the lifetime of a project, and that can turn the maintenance of a traceability matrix into an administrative nightmare. Furthermore, connected systems extend that scenario into the maintenance phase, requiring revision whenever a vulnerability is exposed.

Automating the tracing of requirements alleviates this concern by automatically maintaining the connections between the requirements, development, and testing artefacts and activities. Any changes in the associated documents or software code are automatically highlighted such that any consequential re-testing can be dealt with accordingly (Figure 5).

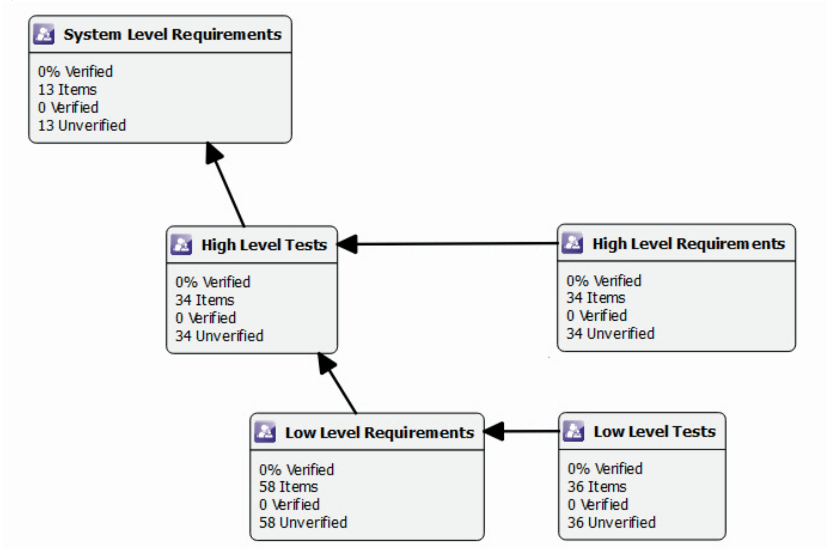


Figure 5: The Uniview graphic from the TBmanager® component of the LDRA tool suite, showing how the relationships between tests and requirements can be configured

[11] http://www.westfallteam.com/Papers/Bidirectional_Requirements_Traceability.pdf Bidirectional Requirements Traceability, Linda Westfall

Static analysis and bi-directional traceability

In general, static analysis tools provide a source of evidence that the standard's objectives have been met in the implemented code, and that the designers' vision in meeting those objectives has been realized. For example, IEC 61508:2010-7[12] Sections B.2 and C.2, "*Requirements and detailed design*" require the use of data flow diagrams and decision/truth tables to represent the system design. These lend themselves to verification by means of static analysis later in the lifecycle by means of "*as implemented*" data flow diagrams (Figure 6), LCSAJ (Linear Code Sequence and Jumps) truth tables, and MC/DC (Modified Condition/Decision Coverage) test case analyses.

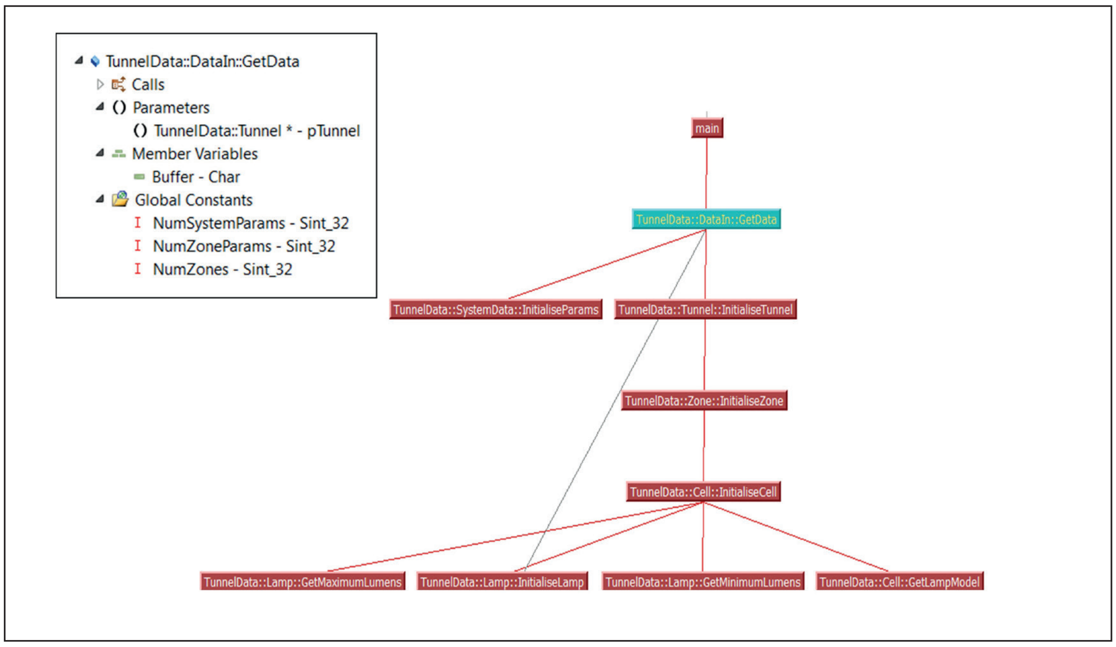

TunnelData::DataIn::GetData
Calls
Parameters
TunnelData::Tunnel * - pTunnel
Member Variables
Buffer - Char
Global Constants
NumSystemParams - Sint_32
NumZoneParams - Sint_32
NumZones - Sint_32
main
TunnelData::DataIn::GetData
TunnelData::SystemData::InitialiseParams
TunnelData::Tunnel::InitialiseTunnel
TunnelData::Zone::InitialiseZone
TunnelData::Cell::InitialiseCell
TunnelData::Lamp::GetMaximumLumens
TunnelData::Lamp::InitialiseLamp
TunnelData::Lamp::GetMinimumLumens
TunnelData::Cell::GetLampModel

Figure 6: Diagrammatic representations of control and data flow, generated from source code by the LDRA tool suite, aid verification that software architectural design has been implemented correctly

The reports generated as products of these and other static analysis techniques can be linked to a requirements traceability tool to automate bi-directional end-to-end traceability to and from the system safety requirements and the software safety requirements. Figure 7 shows a fuller list of example IEC 61508 design objectives that can be confirmed to be correctly implemented in the application code using static analysis.

IEC 61508:2010-3 Section 7.3: "*Validation plan for software aspects of system safety*"

This section of the standard is focused on the planning of when, where, how and by whom safety-related verification and validation activities are to be carried out. It requires consideration of whether these activities are to be manually or automatically implemented, but the more detailed definition of requirements for the tools themselves are not considered until later in the lifecycle.

IEC 61508:2010-3 Section 7.4.3: "*Requirements for software architecture design*"

Table A.2 in the standard is focused on this "*Requirements for software architecture design*" section. IEC 61508:2010-7 Sections B.2 "*E/E/PE system design requirements specification*", C.2 "*Requirements and detailed design*", and C.3 "*Architecture design*" specify where fault detection techniques need to be implemented as part of the software architecture, such as fault detection, error detection and failure assertion programming. These techniques are designed to highlight failures, thus providing the basis for counter-measures in order to minimize their consequences.

[12] IEC 61508:2010-7, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures

Static analysis techniques can be used to confirm that these sound design objectives are reflected in the code. Examples include Structured Programming Verification (used to identify unstructured code which may lead to erroneous behaviour of the application) and the generation of complexity metrics such as Cyclomatic Complexity and Halstead's metrics (used to help determine the software module size, software complexity and the data flow information). This confirmation of implemented objectives also reflects the need for bi-directional traceability as highlighted in IEC 61508:2010-7 Section C.2.11 "*Traceability*".

IEC 61508 reference	Assistance from LDRA tools
IEC 61508-3 "*Table A.2 – Software design and development – software architecture design - Fault detection*" and "*Error detecting codes*"	Static and dynamic analysis can be used to confirm validity of code implementing these design features
IEC 61508-3 "*Table A.2 – Failure assertion programming*"	Can be confirmed using unit and module testing, particularly negative testing
IEC 61508-3 "*Table A.2 – Modular approach*"	Can be confirmed by means of structural analysis and its associated reports and charts
IEC 61508-3 "*Table A.2 – Forward traceability between the software safety requirements specification and software architecture*"	Requirements traceability tool to manage links between requirements, objectives, personnel, source code and the other LDRA tools to maintain traceability information in real time
IEC 61508-3 "*Table A.2 – Backward traceability between the software safety requirements specification and software architecture*"	
IEC 61508-3 "*Table A.2 – Structured diagrammatic methods*"	Pertinent static analysis techniques including structured programming verification
IEC 61508-3 "Table A.2 – Semi-formal methods"	Formal methods transfer the principles of mathematical reasoning to the specification and implementation of technical systems. In practical terms, LDRA tools use static analysis of the source code with respect to various programming models (E.g. MISRA C), whilst the parser engine mathematically analyses the structure and provides analysis reports
IEC 61508-3 "*Table A.2 – Formal design and refinement methods*"	
IEC 61508-3 "*Table A.3 – Strongly typed programming language*" and "*Table A.3 – Language subset*"	LDRA tools provide code standards checking, including MISRA rules

Figure 7: Examples of IEC 61508 design objectives that can be confirmed as implemented in the application code through the application of LDRA tools

IEC 61508:2010-3 Section 7.4.4: "*Requirements for support tools, including programming languages*"

This section discusses the selection of the programming language(s) to be used and the associated tool chain for the development of that code, including verification and validation tools (section 7.4.4.2), static code analysers, test coverage monitors and configuration management tools.

IEC 61508:2010-7 Section C.4.5 "*Suitable programming languages*" recommends that "*The programming language chosen should lead to an easily verifiable code with a minimum of effort and facilitate program development, verification and maintenance.*"

Features which make verification difficult and therefore should be avoided are:

- Unconditional jumps excluding subroutine calls
- Recursion
- Pointers, heaps or any type of dynamic variables or objects
- Interrupt handling at source code level
- Multiple entries or exits of loops, blocks or subprograms
- Implicit variable initialisation or declaration
- Variant records and equivalence and
- Procedural parameters

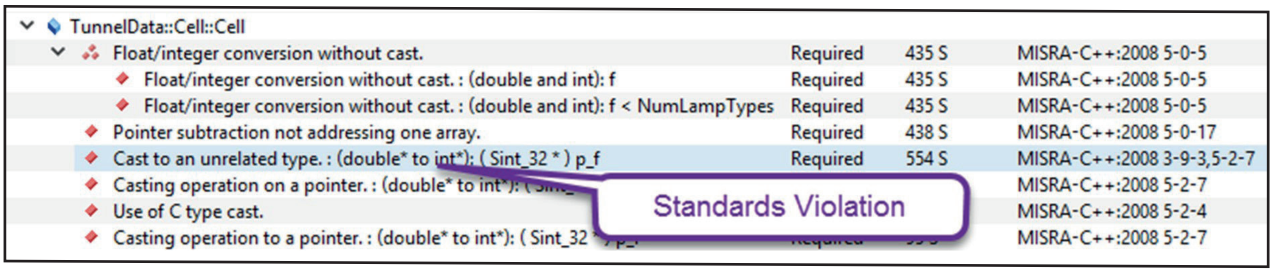
Static analysis techniques provide automated facilities to check compliance with the programming standards such as MISRA and CERT C which are designed to prevent the introduction of vulnerabilities or latent errors in source code. Such coding standards usually explicitly disallow the use of the programming features identified above, and adherence to these coding standards can be checked automatically (Figure 8).

Coding standards
There are many 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/AMD1/ADD2
SEI CERT C
CWE

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

Java
CWE
CERT J



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

Figure 8: Adherence to coding standards guidelines can be checked automatically by LDRA's static analysis tools

Table A.3 from IEC 61508-3 Annex A[13] references the need for "*certified tools and translators*". That implies detailed and thorough testing of those tools, which is a time consuming and costly process.

In most cases, the most cost effective approach is therefore to use a tool that is already approved for the applied standard by an appropriate TÜV certifying organization.

IEC 61508:2010-3 Section 7.4.5: "*Requirements for detailed design and development – software system design*"

This section of the standard specify design and coding standard enforcement measures pertinent to the source code.

Figure 9 is a reproduction of Table A.4 from Part 3 of the standard, which refers to the sections "*Requirements for detailed design and development*" (7.4.5), "*Requirements for code implementation*" (7.4.6) and C.2 of IEC 61508 (Part 7).

[13] IEC 61508:2010-3 Annex A, Table A.3, Software design and development – support tools and programming language

Technique/Measure*		Ref	SIL			
			1	2	3	4
1a	Structured methods **	C.2.1	HR	HR	HR	HR
1b	Semi-formal methods **	Table B.7	R	HR	HR	HR
1c	Formal design and refinement methods **	B.2.2, C.2.4	---	R	R	HR
2	Computer-aided design tools	B.3.5	R	R	HR	HR
3	Defensive programming	C.2.5	---	R	HR	HR
4	Modular approach	Table B.9	HR	HR	HR	HR
5	Design and coding standards	C.2.6 Table B.1	R	HR	HR	HR
6	Structured programming	C.2.7	HR	HR	HR	HR
7	Use of trusted/verified software elements (if available)	C.2.10	R	HR	HR	HR
8	Forward traceability between the software safety requirements specification and software design	C.2.11	R	R	HR	HR

"HR"	The method is highly recommended for this SIL.
"R"	The method is recommended for this SIL.
"---"	The method has no recommendation for or against its usage for this SIL.
**	Group 1, "Structured methods". Use measure 1a only if 1b is not suited to the domain for SIL 3R4.

Figure 9: Copy of IEC 61508-3 Table A.4[14], with techniques supported by the LDRA static analysis tools highlighted

The software safety requirements require consideration of the following during the design and development phase:

- Completeness with respect to software safety requirements specification
- Correctness with respect to software safety requirements specification
- Freedom from intrinsic design faults
- Simplicity and understandability
- Predictability of behaviour
- Verifiable and testable design
- Fault tolerance / fault detection
- Freedom from common cause failure

The "*Completeness*" and "*Correctness*" are both reflections of the overriding requirement for bi-directional traceability, and that is most easily managed through the application of a requirements traceability tool. The complexity of application code design can be controlled using static analysis to generate industry standard metrics, and industrial coding standards including MISRA C:2012[15], MISRA C++:2008[16], SEI CERT C[17], and JSF++ HR AV[18] are designed to limit the use of constructs most likely to introduce such as common cause failure and unpredictability.

IEC 61508:2010-3 Section 7.4.6: "*Requirements for code implementation*"

This is a short section, mostly consisting of an emphasis for the need for traceability. Best practise dictates that static and dynamic analysis of the code is an ongoing process while the code is being developed, and so the code implementation process is interwoven with module and integration testing, as well as ongoing static analysis.

[14] IEC 61508-3 Annex A Table A.4 – Software design and development – Detailed design
[15] MISRA C:2012: Guidelines for use of the C language in critical systems, ISBN 978-906400-11-8 (PDF), March 2013
[16] MISRA C++:2008 - Guidelines for the use of the C++ language in critical systems, ISBN 978-906400-04-0 (PDF), June 2008.
[17] SEI CERT C Coding Standard, https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
[18] JSF++ HR AV, JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM, Document Number 2RDU00001 Rev C, 2005

IEC 61508:2010-3 Section 7.4.7: *"Requirements for software module testing" and Section 7.4.8: "Requirements for software integration testing"*

	Technique/Measure	Ref	SIL			
			1	2	3	4
1	Probabilistic testing	C.5.1	---	R	R	R
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
3	Data recording and analysis	C.5.2	HR	HR	HR	HR
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	HR
5	Performance testing	Table B.6	R	R	HR	HR
6	Model based testing	C.5.27	R	R	HR	HR
7	Interface testing	C.5.3	R	R	HR	HR
8	Test management and automation tools	C.4.7	R	HR	HR	HR
9	Forward traceability between the software design specification and the module and integration test specifications	C.2.11	R	R	HR	HR
10	Formal verification	C.5.12	---	---	R	R
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 10: Copy of IEC 61508-3 Table A.5[19], with techniques supported by the LDRA static analysis tools highlighted

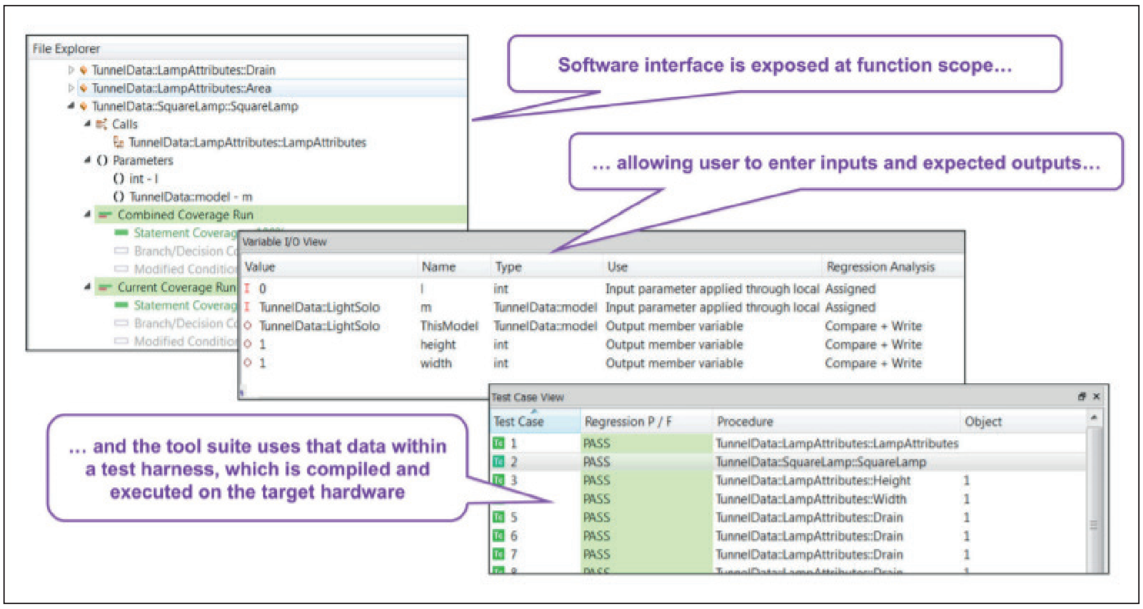
Figure 10 is a reproduction of IEC 61508-3 Table A.5, which refers to IEC 61508-3 Section 7.4.7 "*Software module testing*", IEC 61508-3 Section 7.4.8 "*Requirements for software integration testing*", and IEC 61508-7 Section C.5 "*Verification and modification*". These sections identify methods designed to contribute to the achievement of software safety, such as software module testing and software integration testing.

A combination of code review and software module testing verifies that a software module satisfies its associated specification. Software module testing in particular lends itself well to several objectives specified in the standard:

- Completeness of testing with respect to the software design specification
- Correctness of testing with respect to the software design specification (successful completion)
- Repeatability
- Precisely defined testing configuration

Although module testing can be performed by writing custom code for the purpose, the use of a certified, proven test tool is likely to be much more cost effective unless the code base is very small. Such a tool can automatically generate test drivers and harnesses (wrapper code) with no extra coding or scripting required, enabling tests to be easily and efficiently run on code units (Figure 11).

[19] IEC 61508-3 Annex A Table A.5 – Software design and development –Software module testing and integration


Software interface is exposed at function scope...
... allowing user to enter inputs and expected outputs...
... and the tool suite uses that data within a test harness, which is compiled and executed on the target hardware
File Explorer
Variable I/O View
Test Case View

```
TunnelData::LampAttributes::Drain
TunnelData::LampAttributes::Area
TunnelData::SquareLamp::SquareLamp
  Calls
    TunnelData::LampAttributes::LampAttributes
  Parameters
    int - l
    TunnelData::model - m
  Combined Coverage Run
    Statement Coverag
    Branch/Decision Co
    Modified Conditio
  Current Coverage Run
    Statement Coverag
    Branch/Decision Co
    Modified Conditio
```

Value	Name	Type	Use	Regression Analysis
0	l	int	Input parameter applied through local	Assigned
TunnelData::LightSolo	m	TunnelData::model	Input parameter applied through local	Assigned
TunnelData::LightSolo	ThisModel	TunnelData::model	Output member variable	Compare + Write
1	height	int	Output member variable	Compare + Write
1	width	int	Output member variable	Compare + Write

Test Case	Regression P / F	Procedure	Object
1	PASS	TunnelData::LampAttributes::LampAttributes	
2	PASS	TunnelData::SquareLamp::SquareLamp	
3	PASS	TunnelData::LampAttributes::Height	1
	PASS	TunnelData::LampAttributes::Width	1
5	PASS	TunnelData::LampAttributes::Drain	1
6	PASS	TunnelData::LampAttributes::Drain	1
7	PASS	TunnelData::LampAttributes::Drain	1

Figure 11: Performing requirement based unit-testing using the TBrun® component of the LDRA tool suite

These tests can be subsequently regressed, with clear maintenance tracking and seamless storage of test data and results.

IEC 61508:2010-3 Section 7.5: *"Programmable electronics integration (hardware and software)"*

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Functional and black box testing	B.5.1 B.5.2 Table B.3	**HR**	**HR**	**HR**	**HR**
2	Performance Testing	Table B.6	**R**	**R**	**HR**	**HR**
3	Forward traceability between the system and software design requirements for hardware/software integration and the hardware/software integration test specifications	C.2.11	**R**	**R**	**HR**	**HR**

"HR" The method is highly recommended for this SIL.
"R" The method is recommended for this SIL.
"---" The method has no recommendation for or against its usage for this SIL.

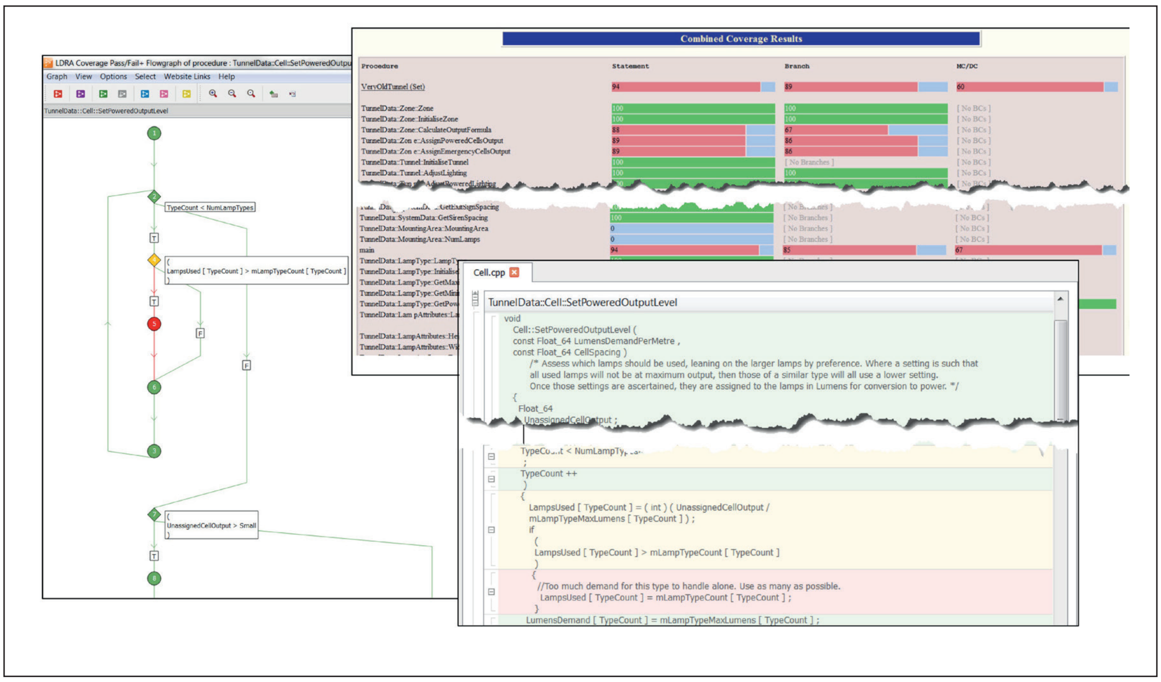
Figure 12: Copy of IEC 61508-3 Table A.6[20], with techniques supported by the LDRA static analysis tools highlighted

It is necessary for the integrated software to be proven on the target programmable electronic hardware by means of a number of specified test techniques. Depending on the SIL, these may include:

- Functional Tests
- Black box tests, to check the dynamic behaviour under real functional conditions, which reveal failures to meet the functional specification
- This includes testing data from:
  - Permissible ranges
  - Inadmissible ranges
  - The range limits
  - Extreme values
  - Combinations of the above classes

[20] IEC 61508-3 Annex A Table A.6 - Programmable electronics integration (hardware and software)

Function and call coverage can be supported by unit test, system test, or a combination of the two operating in tandem (Figure 13). For instance, a preferred approach for a particular project might be to use dynamic system test to generate coverage of most of the source code. That data could then be complemented by coverage generated during unit tests designed to exercise code constructs which are inaccessible during normal operation, such as defensive code.



LDRA Coverage Pass/Fail+ Flowgraph of procedure : TunnelData::Cell::SetPoweredOutpu
Graph View Options Select Website Links Help
TunnelData::Cell::SetPoweredOutputLevel
TypeCount < NumLampTypes
LampsUsed [TypeCount] > mLampTypeCount [TypeCount]
UnassignedCellOutput > Small
Combined Coverage Results
Cell.cpp

Procedure	Statement	Branch	MC/DC
VeryOldTunnel (Set)	94	89	60
TunnelData::Zone::Zone	100	100	[No BCs]
TunnelData::Zone::InitialiseZone	100	100	[No BCs]
TunnelData::Zone::CalculateOutputFormula	88	67	[No BCs]
TunnelData::Zon e::AssignPoweredCellsOutput	89	86	[No BCs]
TunnelData::Zon e::AssignEmergencyCellsOutput	89	86	[No BCs]
TunnelData::Tunnel::InitialiseTunnel	100	[No Branches]	[No BCs]
TunnelData::Tunnel::AdjustLighting	100	100	[No BCs]
TunnelData::SystemData::GetSirenSpacing	100	[No Branches]	[No BCs]
TunnelData::MountingArea::MountingArea	0	[No Branches]	[No BCs]
TunnelData::MountingArea::NumLamps	0	[No Branches]	[No BCs]
main	94	85	67

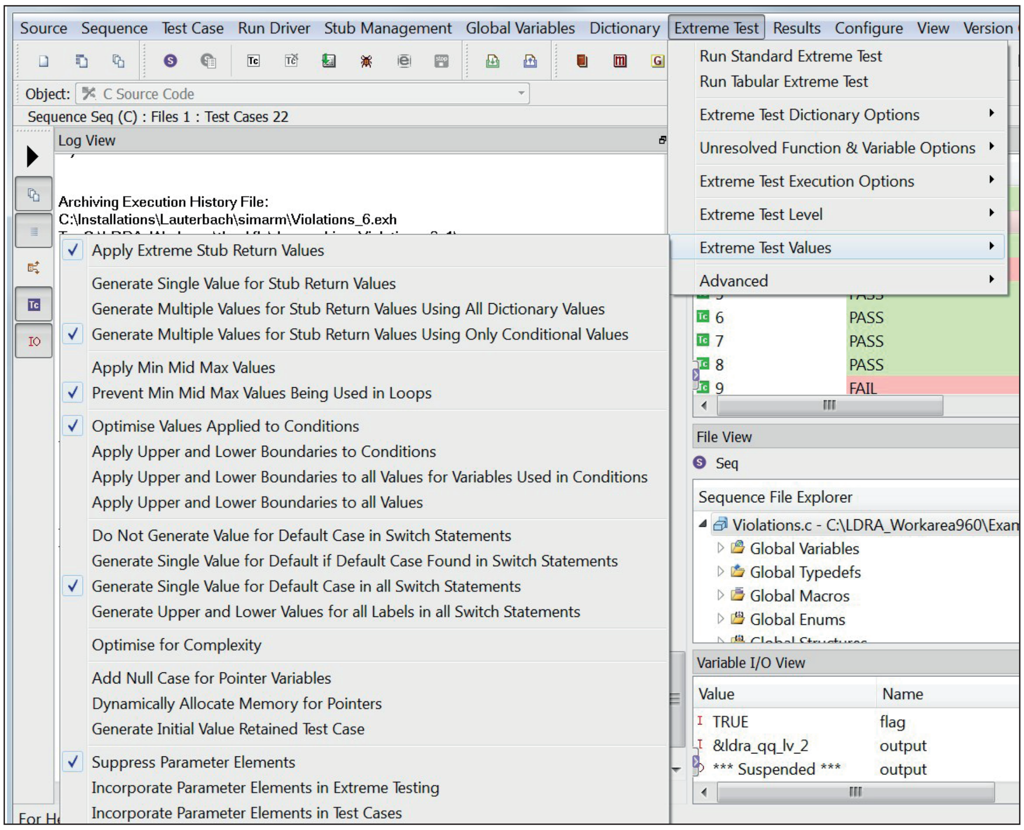
TunnelData::Cell::SetPoweredOutputLevel

```
void
  Cell::SetPoweredOutputLevel (
  const Float_64 LumensDemandPerMetre ,
  const Float_64 CellSpacing )
      /* Assess which lamps should be used, leaning on the larger lamps by preference. Where a setting is such that
      all used lamps will not be at maximum output, then those of a similar type will all use a lower setting.
      Once those settings are ascertained, they are assigned to the lamps in Lumens for conversion to power. */
  {
    Float_64
      UnassignedCellOutput ;

      TypeCount < NumLampTy
      ;
      TypeCount ++
      )
      {
        LampsUsed [ TypeCount ] = ( int ) ( UnassignedCellOutput /
        mLampTypeMaxLumens [ TypeCount ] ) ;
        if
          (
          LampsUsed [ TypeCount ] > mLampTypeCount [ TypeCount ]
          )
          {
            //Too much demand for this type to handle alone. Use as many as possible.
            LampsUsed [ TypeCount ] = mLampTypeCount [ TypeCount ] ;
          }
      LumensDemand [ TypeCount ] = mLampTypeMaxLumens [ TypeCount ] ;
```

Figure 13: Examples of representations of structural coverage within the LDRA tool suite

To complement this structural coverage analysis, boundary values could be provided manually or generated automatically (Figure 14) to check the permissible and inadmissible ranges.



Source Sequence Test Case Run Driver Stub Management Global Variables Dictionary Extreme Test Results Configure View Version
Object: C Source Code
Log View
Run Standard Extreme Test
Run Tabular Extreme Test
Extreme Test Dictionary Options
Unresolved Function & Variable Options
Extreme Test Execution Options
Extreme Test Level
Extreme Test Values
Advanced
File View
Seq
Sequence File Explorer
Violations.c - C:\LDRA_Workarea960\Exam
Global Variables
Global Typedefs
Global Macros
Global Enums
Variable I/O View
For H

Sequence Seq (C) : Files 1 : Test Cases 22

Archiving Execution History File:
C:\Installations\Lauterbach\simarm\Violations_6.exh

✓ Apply Extreme Stub Return Values
Generate Single Value for Stub Return Values
Generate Multiple Values for Stub Return Values Using All Dictionary Values
✓ Generate Multiple Values for Stub Return Values Using Only Conditional Values
Apply Min Mid Max Values
✓ Prevent Min Mid Max Values Being Used in Loops
✓ Optimise Values Applied to Conditions
Apply Upper and Lower Boundaries to Conditions
Apply Upper and Lower Boundaries to all Values for Variables Used in Conditions
Apply Upper and Lower Boundaries to all Values
Do Not Generate Value for Default Case in Switch Statements
Generate Single Value for Default if Default Case Found in Switch Statements
✓ Generate Single Value for Default Case in all Switch Statements
Generate Upper and Lower Values for all Labels in all Switch Statements
Optimise for Complexity
Add Null Case for Pointer Variables
Dynamically Allocate Memory for Pointers
Generate Initial Value Retained Test Case
✓ Suppress Parameter Elements
Incorporate Parameter Elements in Extreme Testing
Incorporate Parameter Elements in Test Cases

6	PASS
7	PASS
8	PASS
9	FAIL

Value	Name
TRUE	flag
&ldra_qq_lv_2	output
*** Suspended ***	output

Figure 14: Using the TBrun and TBeXtreme® components of the LDRA tool suite to automatically create boundary value tests

IEC 61508:2010-3 Section 7.4.6: "*Requirements for code implementation*"

Aside from emphasizing the need for bi-directional traceability, this section is largely a stub, cross-referencing other sections of the standard.

IEC 61508:2010-3 Section 7.7: "*Software aspects of system safety validation*"

Figure 15 is a reproduction of IEC 61508-3 Table A.7 from the standard, which refers to IEC 61508-3 Section 7.7 "*Software aspects of system safety validation*" and IEC 61508-7 Section C.2 "*Requirements and detailed design*". These sections specify the software aspects of system safety validation. These ensure that the integrated system complies with the software safety requirements specification at the required safety integrity level.

	Technique/Measure	Ref	SIL			
			1	2	3	4
1	Probabilistic testing	C.5.1	...	R	R	HR
2	Process simulation	C.5.18	R	R	HR	HR
3	Modelling	Table B.5	R	R	HR	HR
4	Functional and black-box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	HR
5	Forward traceability between the software safety requirements specification and the software safety validation plan	C.2.11	R	R	HR	HR
6	Backward traceability between the software safety validation plan and the software safety requirements specification	C.2.11	R	R	HR	HR
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 15: Copy of IEC 61508-3 Table A.7[21], with techniques supported by the LDRA tool suite highlighted.

Functional and Black-Box testing can be used to check whether the functions of a system or program behave as the specification dictates when executed in a prescribed environment according to established criteria, and the associated configuration files can be stored and used for the automated regression analysis to confirm ongoing adherence to the specified requirements.

Automated requirements traceability tools complement this by providing forward and backward traceability between the software safety requirements specification and software safety validation plan.

IEC 61508:2010-3 Section 7.8: "*Software modification*"

Figure 16 is a reproduction of IEC 61508-3 Table A.8 from the standard, which refers to IEC 61508-3 Section 7.8 "*Software modification*" and IEC 61508-7 Section C.5 "*Verification and modification*".

These sections specify the steps to be followed during the modification of software. They provide guidance on the implementation of corrections, enhancements and adaptations of validated software, ensuring that the adherence to IEC 61508 for the resulting modified system is not compromised.

[21] IEC 61508-3 Annex A Table A.7 – Software aspects of system validation

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Impact analysis	C.5.23	**HR**	**HR**	**HR**	**HR**
2	Reverify changed software module	C.5.23	**HR**	**HR**	**HR**	**HR**
3	Reverify affected software modules	C.5.23	**R**	**HR**	**HR**	**HR**
4a	Revalidate complete system	Table A.7	**...**	**R**	**HR**	**HR**
4b	Regression validation	C.5.25	**R**	**HR**	**HR**	**HR**
5	Software configuration management	C.5.24	**HR**	**HR**	**HR**	**HR**
6	Data recording and analysis	C.5.2	**HR**	**HR**	**HR**	**HR**
7	Forward traceability between the Software safety requirements specification and the software modification plan (including reverification and revalidation)	C.2.11	**R**	**R**	**HR**	**HR**
8	Backward traceability between the software modification plan (including reverification and revalidation) and the software safety requirements specification	C.2.11	**R**	**R**	**HR**	**HR**
”HR” The method is highly recommended for this SIL. **“R“** The method is recommended for this SIL. **“---”** The method has no recommendation for or against its usage for this SIL.						

Figure 16: Copy of IEC 61508-3 Table A.8[22] , with techniques and measures supported by the LDRA tool suite highlighted

The following techniques and measures should be considered with regards to software modification:

- Completeness of modification with respect to its requirements
- Correctness of modification with respect to its requirements
- Freedom from introduction of intrinsic design faults
- Avoidance of unwanted behaviour
- Verifiable and testable design
- Regression testing and verification coverage

In this context, impact analysis is designed to determine whether a change or an enhancement to a software system has affected its overall functionality, or has the potential to do so. There are three possible conclusions:

- Only the changed software module needs to be re-verified
- All affected software modules need to be re-verified, or
- The complete system needs to be re-verified

The level of re-verification required will be influenced by the number of software modules affected, the criticality of the affected software modules, and the nature of the change.

The connected system – a new significance for system modification

With the advent of the connected device and the Internet of Things, system maintenance takes on a new significance. For any connected systems, requirements don’t just change in an orderly manner during development. They change without warning - whenever some smart Alec finds a new vulnerability, develops a new hack, or compromises the system. And they keep on changing throughout the lifetime of the device.

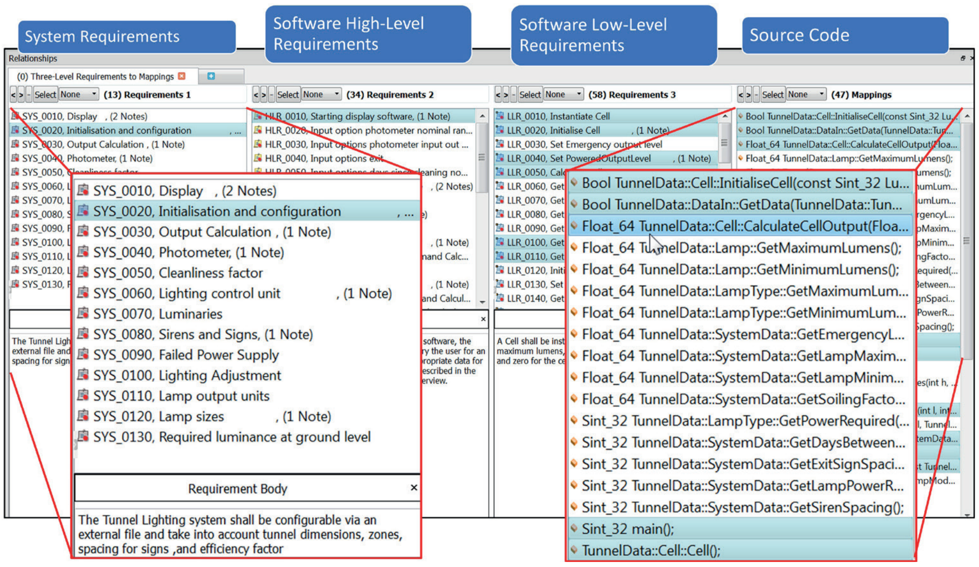
For that reason, the ability of next-generation automated management and requirements traceability tools and techniques to create relationships between requirements, code, static and dynamic analysis results, and unit- and system-level tests is especially valuable for connected systems. Linking these elements already enables the entire software development cycle to become traceable, making it easy for teams to identify problems and

[22] IEC 61508-3 Annex A Table A.8 – Modification

implement solutions faster and more cost effectively. But they are perhaps even more important after product release, presenting a vital competitive advantage in the ability to respond quickly and effectively whenever security is compromised.

Many software modifications will require changes to the existing software functionality – perhaps with regards to additional utilities in the software. In such circumstances, it is important to ensure that any changes made or additions to the software do not adversely affect the existing code.

A requirements traceability tool can help to alleviate this concern by automatically maintaining the connections between the requirements, development, and testing artefacts and activities. In the example shown in Figure 18, suppose that a change is proposed to the system requirement "*Installation and configuration*". The traceability established at development time between requirements, code and tests mean that the tool can show which parts of the code are impacted by the proposed change, as highlighted in the example.



SYS_0010, Display , (2 Notes)
SYS_0020, Initialisation and configuration , ...
SYS_0030, Output Calculation , (1 Note)
SYS_0040, Photometer, (1 Note)
SYS_0050, Cleanliness factor
SYS_0060, Lighting control unit , (1 Note)
SYS_0070, Luminaries
SYS_0080, Sirens and Signs, (1 Note)
SYS_0090, Failed Power Supply
SYS_0100, Lighting Adjustment
SYS_0110, Lamp output units
SYS_0120, Lamp sizes , (1 Note)
SYS_0130, Required luminance at ground level

The Tunnel Lighting system shall be configurable via an external file and take into account tunnel dimensions, zones, spacing for signs ,and efficiency factor

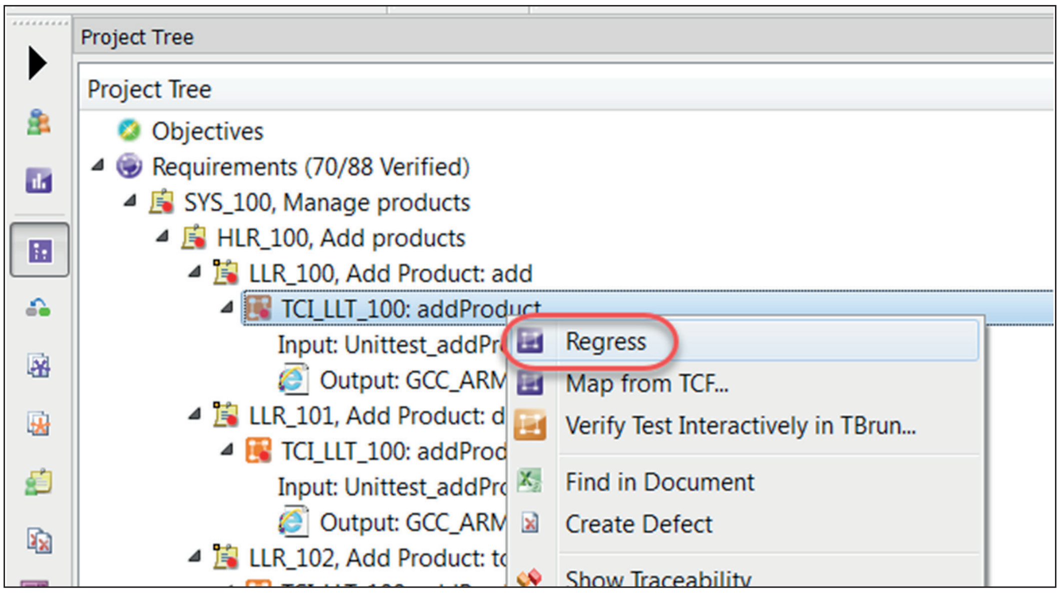
HLR_0010, Starting display software, (1 Note)
HLR_0020, Input option photometer nominal ran...
HLR_0030, Input options photometer input out ...
HLR_0040, Input options exit

LLR_0010, Instantiate Cell
LLR_0020, Initialise Cell , (1 Note)
LLR_0030, Set Emergency output level
LLR_0040, Set PoweredOutputLevel , (1 Note)

Bool TunnelData::Cell::InitialiseCell(const Sint_32 Lu...
Bool TunnelData::DataIn::GetData(TunnelData::Tun...
Float_64 TunnelData::Cell::CalculateCellOutput(Floa...
Float_64 TunnelData::Lamp::GetMaximumLumens();
Float_64 TunnelData::Lamp::GetMinimumLumens();
Float_64 TunnelData::LampType::GetMaximumLum...
Float_64 TunnelData::LampType::GetMinimumLum...
Float_64 TunnelData::SystemData::GetEmergencyL...
Float_64 TunnelData::SystemData::GetLampMaxim...
Float_64 TunnelData::SystemData::GetLampMinim...
Float_64 TunnelData::SystemData::GetSoilingFacto...
Sint_32 TunnelData::LampType::GetPowerRequired(...
Sint_32 TunnelData::SystemData::GetDaysBetween...
Sint_32 TunnelData::SystemData::GetExitSignSpaci...
Sint_32 TunnelData::SystemData::GetLampPowerR...
Sint_32 TunnelData::SystemData::GetSirenSpacing();
Sint_32 main();
TunnelData::Cell::Cell();

Figure 18: Identifying the impact of requirements change with the TBmanager component of the LDRA tool suite

In this scenario, the existing code as launched will also have undergone quality control measures in accordance with the IEC 61508 standard such as static analysis to assess whether coding standards have been met, and unit tests to confirm functionality of each code module.



Project Tree
Objectives
Requirements (70/88 Verified)
SYS_100, Manage products
HLR_100, Add products
LLR_100, Add Product: add
TCI_LLT_100: addProduct
Input: Unittest_addPr
Output: GCC_ARM
LLR_101, Add Product: d
TCI_LLT_100: addProd
Input: Unittest_addPr
Output: GCC_ARM
LLR_102, Add Product: t

Figure 19: Showing functions requiring retest with the TBmanager component of the LDRA tool suite

Figure 19 shows an example system which has been subject to a change request for the "*Add products*" requirement. Those parts of the system which are potentially affected by the change are easily identified by means of a red dot, whereas unaffected functions carry a green dot.

Regression Analysis feature can then be used to verify whether the newly introduced or modified modules have only affected the functionality of the existing system as intended, or the complete system can be re-validated.

IEC 61508:2010-3 Section 7.9: "*Software verification*"

Figure 20 is a reproduction of IEC 61508-3 Table A.9 from the standard, which refers to IEC 61508-3 Section 7.9 "*Software verification*" and IEC 61508-7 Section C.2 "*Requirements and detailed design*".

IEC 61508-3 Section 7.9 considers generic aspects of verification common to several safety lifecycle phases.

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Formal proof	C.5.12	---	R	R	HR
2	Animation of specification and design	C.5.26	R	R	R	R
3	Static analysis	B.6.4 Table B.8	R	HR	HR	HR
4	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
5	Forward traceability between the software design specification and the software verification (including data verification) plan	C.2.11	R	R	HR	HR
6	Backward traceability between the software verification (including data verification) plan and the software design specification	C.2.11	R	R	HR	HR
7	Offline numerical analysis	C.2.13	R	R	HR	HR
Software module testing and integration		See Table A.5				
Programmable electronics integration testing		See Table A.6				
Software system testing (validation)		See Table A.7				
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 20: Copy of IEC 61508-3 Table A.9[23], with techniques and measures supported by the LDRA tool suite highlighted

Conclusions

With its many sections, clauses and sub-clauses, IEC 61508 may at first seem intimidating, and its system of cross-referencing tables in annexes can make it difficult to follow. However, once broken down into digestible pieces, its principles offer sound guidance in the establishment of a high quality software development process - not only leading up to initial product release but into maintenance and beyond. Such a process is paramount for the assurance of true reliability, quality, safety and effectiveness of programmable electronic components. When supported by a complementary and comprehensive suite of tools for analysis and testing, it can smooth the way for development teams to work together to effectively develop and maintain large projects with confidence in their quality.

[23] IEC 61508-3 Annex A Table A.9 – Software Verification

Works Cited

Bi-directional Requirements Traceability, Linda Westfall, http://www.westfallteam.com/Papers/Bidirectional_Requirements_Traceability.pdf

CWE web site https://cwe.mitre.org/

ANSYS Esterel SCADE web site http://www.esterel-technologies.com/products/scade-suite/

High Integrity C++ Coding Standard web site http://www.codingstandard.com/section/index/

IBM Rational DOORS web site http://www-03.ibm.com/software/products/en/ratidoor

IEC 61508:2010-3, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software Requirements

IEC 61508:2010-5, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 5: Examples of methods for the determination of safety integrity levels

IEC 61508:2010-7, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures

ISO 26262:2011, Road vehicles – Functional safety, Copyright © 2015 IEC, Geneva, Switzerland. All rights acknowledged

JSF++ HR AV, JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM, Document Number 2RDU00001 Rev C, 2005

MathWorks Simulink web site https://www.mathworks.com/products/simulink.html

MISRA C:1998 Guidelines for the Use of the C Language in Vehicle Based Software, ISBN 978-0-9524156-6-4, April 1998, October 2002.

MISRA C:2004 Guidelines for the Use of the C Language in Critical Systems, ISBN 0 9524156 2 3 (paperback), ISBN 0 9524156 4 X (PDF), October 2004.

MISRA C:2012 - Addendum 3: Coverage of MISRA C:2012 against CERT C, ISBN 978-906400-19-4 (PDF), January 2018.

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

MISRA C++:2008 - Guidelines for the use of the C++ language in critical systems, ISBN 978-906400-04-0 (PDF), June 2008.

Object Management Group web site http://www.omg.org/spec/ReqIF/

SEI CERT C Coding Standard web site https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

SEI CERT C++ Coding Standard https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

SEI CERT Oracle Coding Standard for Java, https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

Siemens Polarion PLM web site https://polarion.plm.automation.siemens.com/

Appendix: The Annex B tables

IEC 61508 uses a set of tables to identify particular techniques to be applied. These techniques often apply to different stages of the lifecycle which makes them difficult to integrate into the narrative illustrated by the "*V*" model used by the standard, especially when they are sub-referenced – hence this appendix.

IEC 61508:2010-3 Annex B Table B.1: "*Design and coding standards*"

Referenced by table A.4

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	**HR**	**HR**	**HR**	**HR**
2	No dynamic objects	C.2.6.3	**R**	**HR**	**HR**	**HR**
3a	No dynamic variables	C.2.6.3	**---**	**R**	**HR**	**HR**
3b	Online checking of the installation of dynamic variables	C.2.6.4	**---**	**R**	**HR**	**HR**
4	Limited use of interrupts	C.2.6.5	**R**	**R**	**HR**	**HR**
5	Limited use of pointers	C.2.6.6	**---**	**R**	**HR**	**HR**
6	Limited use of recursion	C.2.6.7	**---**	**R**	**HR**	**HR**
7	No unstructured control flow in programs in higher level languages	C.2.6.2	**R**	**HR**	**HR**	**HR**
8	No automatic type conversion	C.2.6.2	**R**	**HR**	**HR**	**HR**
"**HR**" The method is highly recommended for this SIL. "**R**" The method is recommended for this SIL. "**---**" The method has no recommendation for or against its usage for this SIL.						

Figure 18: Copy of IEC 61508-3 Table B.1, with techniques and measures supported by the LDRA tool suite highlighted

To reduce the likelihood of errors in the safety-related code, the use of an appropriate coding standard is a recommended practice in IEC 61508. As illustrated previously, there are several to choose from depending on the nature of the application, and on preference.

IEC 61508-7 Section C.2.6.2, "*Coding Standards*" recommends the use of a modular approach where the software module size limit and software complexity metrics can be defined. It also recommends the implementation of code understandability metrics, which can be generated by means of static analysis.

It is also possible to statically check for the use of interrupts, pointers, recursion, and non-structured control flow. Structured Programming Verification allows for the determining of unstructured parts of the application code.

IEC 61508:2010-3 Annex B Table B.2: "*Dynamic analysis and testing*"

Referenced by tables A.5 and A.9

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Test case execution from boundary value analysis	C.5.4	R	HR	HR	HR
2	Test case execution from error guessing	C.5.5	R	R	R	R
3	Test case execution from error seeding	C.5.6	---	R	R	R
4	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
5	Performance modelling	C.5.20	R	R	R	HR
6	Equivalence classes and input partition testing	C.5.7	R	R	R	HR
7a	Structural test coverage (entry points) 100 % **	C.5.8	HR	HR	HR	HR
7b	Structural test coverage (statements) 100 %**	C.5.8	R	HR	HR	HR
7c	Structural test coverage (branches) 100 %**	C.5.8	R	R	HR	HR
7d	Structural test coverage (conditions, MC/DC) 100 %**	C.5.8	R	R	R	HR

"HR" The method is highly recommended for this SIL.
"R" The method is recommended for this SIL.
"---" The method has no recommendation for or against its usage for this SIL.
** Where 100% coverage cannot be achieved (e.g. statement coverage of defensive code), an appropriate explanation should be given.

Figure 19: Copy of IEC 61508-3 Table B.2, with techniques and measures supported by the LDRA tool suite highlighted

IEC 61508-7 Section C.5.4, "*Boundary value analysis*" focuses upon the detection of software errors occurring at parameter limits or boundaries. Using unit test, a range of input values and variable boundary values can be provided and checked against the expected results, possibly exposing runtime errors.

The same section also references error guessing, which also lends itself to unit test. Error guessing involves the use of particular data combinations designed to show whether the code behaviour is error-prone.

IEC 61508-7 Section C.5.7, "*Equivalence classes and input partition testing*" discusses means to test the software adequately using a minimum of test data. Profile analysis and data set analysis can helps to identify any redundancy.

IEC 61508-7 Section C.5.8, "*Structure-based testing*", states: "*Based on analysis of the program, a set of input data is chosen so that a large (and often pre-specified target) percentage of the program code is exercised. Measures of code coverage will vary as follows, depending upon the level of rigour required. In all cases, 100 % of the selected coverage metric should be the aim. If it is not possible to achieve 100 % coverage, the reasons why 100 % cannot be achieved should be documented in the test report (for example, defensive code which can only be entered if a hardware problem arises)*".

Structural coverage requirements are classified according to SIL as follows:

SIL 4	Structural test coverage (entry points) = 100 % Structural test coverage (statements) = 100 % Structural test coverage (branches) = 100 % Structural test coverage (conditions, MC/DC) = 100 %
SIL 3	Structural test coverage (entry points) = 100 % Structural test coverage (statements) = 100 % Structural test coverage (branches) = 100 %
SIL 2	Structural test coverage (entry points) = 100 % Structural test coverage (statements) = 100 %
SIL 1	Structural test coverage (entry points) = 100 %

Dynamic Coverage Analysis can be used to achieve the required level of coverage for IEC 61508 compliance.

IEC 61508:2010-3 Annex B Table B.3: "*Functional and Black-box testing*"

Referenced by tables A.5, A.6 and A.7

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Test case execution from cause consequence diagrams	B.6.6.2	---	---	R	R
2	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
3	Prototyping/animation	C.5.17	---	---	R	R
4	Equivalence classes and input partition testing, including boundary value analysis	C.5.7 C.5.4	R	HR	HR	HR
5	Process simulation	C.5.18	R	R	R	R
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 20: Copy of IEC 61508-3 Table B.3, with techniques and measures supported by the LDRA tool suite highlighted

This section outlines an approach to test case generation at the software system level, based only on the system functional specification.

IEC 61508-7 Section C.5.27, "*Model based testing (test case generation)*", discusses the automatic generation of test cases from system models, and the generation of highly repeatable suite of test data. Unit tests can be executed on code generated from modelling tools, and traceability to functional requirements can be automatically maintained.

Software test and model based development

Static and dynamic facilities can be integrated with several different model based development tools, such as IBM® Rational® Rhapsody®[24], MathWorks® Simulink®[25], and ANSYS® SCADE Suite[26]. 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.

[24] http://www-03.ibm.com/software/products/en/ratirhapfami
[25] https://uk.mathworks.com/products/simulink.html
[26] http://www.ansys.com/products/embedded-software/ansys-scade-suite

Figure 21 illustrates how an integration with MathWorks Simulink can be deployed with the LDRA tool suite. Design models are developed with Simulink and verified with Simulink tests. Then, code is generated from Simulink, instrumented by the LDRA tool suite, and 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.

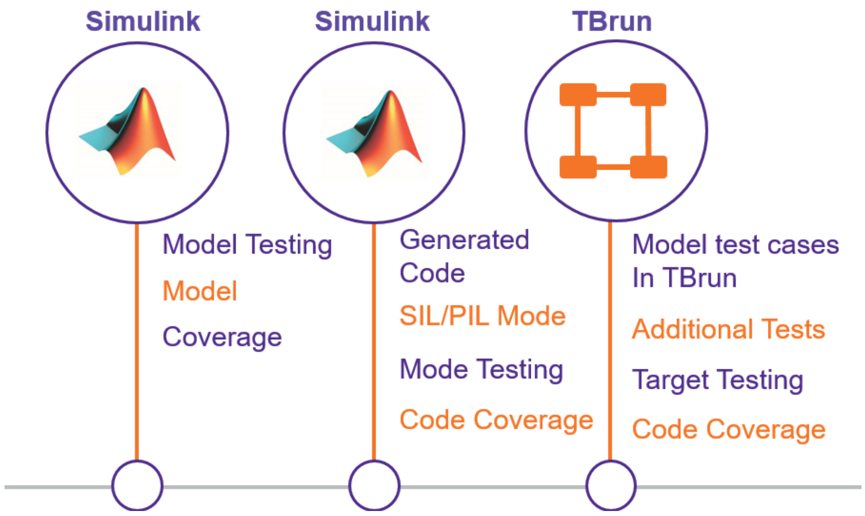


Figure 21: Generating structural coverage data of auto generated code with MathWorks Simulink and the LDRA tool suite, leveraging the TBrun component

Several other tests are available using an integration such as this. The generated source code can be analysed statically to ensure compliance with an appropriate coding standard, such as MISRA C:2012 Appendix E[27]. Additional dynamic testing can be performed at the source level, and requirements based tests can be created to verify functionality and collate structural coverage. Test data can also be imported from Simulink 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 alongside auto-generated code.

IEC 61508:2010-3 Annex B Table B.4: "*Failure analysis*"

Referenced by table A.10

Technique/Measure		Ref	SIL			
			1	2	3	4
1a	Cause consequence diagrams	B.6.6.2	R	R	R	R
1b	Event tree analysis	B.6.6.3	R	R	R	R
2	Fault tree analysis	B.6.6.5	R	R	R	R
3	Software functional failure analysis	B.6.6.4	R	R	R	R

"HR" The method is highly recommended for this SIL.
"R" The method is recommended for this SIL.
"---" The method has no recommendation for or against its usage for this SIL.

Figure 22: Copy of IEC 61508-3 Table B.4, for completeness

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

IEC 61508:2010-3 Annex B Table B.5: "*Modelling*"

Referenced by table A.7

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Data flow diagrams	C.2.2	**R**	**R**	**R**	**R**
2a	Finite state machines	B.2.3.2	**---**	**R**	**HR**	**HR**
2b	Formal methods	B.2.2, C.2.4	**---**	**R**	**R**	**HR**
2c	Time Petri nets	B.2.3.3	**---**	**R**	**HR**	**HR**
3	Performance modelling	C.5.20	**R**	**HR**	**HR**	**HR**
4	Prototyping/animation	C.5.17	**R**	**R**	**R**	**R**
5	Structure diagrams	C.2.3	**R**	**R**	**R**	**HR**
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 23: Copy of IEC 61508-3 Table B.5, with techniques and measures supported by the LDRA tool suite highlighted

IEC 61508-7 Section C.2.2, "*Data flow diagrams*" describes how data input is transformed to output, with each stage in the diagram representing a distinct transformation. Call Graphs and data flow graphs provide the required data flow information, and formal methods are used to generate provide dependency information between each module.

IEC 61508:2010-3 Annex B Table B.6: "*Performance testing*"

Referenced by tables A.5 and A.6

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Avalanche/stress testing	C.5.21	**R**	**R**	**HR**	**HR**
2	Response timings and memory constraints	C.5.22	**HR**	**HR**	**HR**	**HR**
3	Performance requirements	C.5.19	**HR**	**HR**	**HR**	**HR**
"HR" The method is highly recommended for this SIL. **"R"** The method is recommended for this SIL. **"---"** The method has no recommendation for or against its usage for this SIL.						

Figure 24: Copy of IEC 61508-3 Table B.6, with techniques and measures supported by the LDRA tool suite highlighted

IEC 61508-7 Section C.5.21, "*Avalanche/stress testing*" describes the use of tests that determine the robustness of software beyond the limits of normal operation, which is particularly important for mission critical software. It is related to the test regime required to achieve branch coverage, which necessarily includes negative test variations where the software is supposed to fail in some way.

IEC 61508-7 Section C.5.22, "*Response timing and memory constraints*" specifies the use of timing analysis, and unit test can be used to check the execution time of a specific unit of code or module.

IEC 61508:2010-3 Annex B Table B.7: "*Semi-formal methods*"

Referenced by tables A.1, A.2 and A.4

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Logic/function block diagrams	IEC 61131-3	R	R	HR	HR
2	Sequence diagrams	IEC 61131-3	R	R	HR	HR
3	Data flow diagrams	C.2.2	R	R	R	R
4a	Finite state machines/state transition diagrams	B.2.3.2	R	R	HR	HR
4b	Time Petri nets	B.2.3.3	R	R	HR	HR
5	Entity-relationship-attribute data models	B.2.4.4	R	R	R	R
6	Message sequence charts	C.2.14	R	R	R	R
7	Decision/truth tables	C.6.1	R	R	R	R
8	UML	C.3.12	R	R	R	R

"HR" The method is highly recommended for this SIL.
"R" The method is recommended for this SIL.
"---" The method has no recommendation for or against its usage for this SIL.

Figure 25: Copy of IEC 61508-3 Table B.7, with techniques and measures supported by the LDRA tool suite highlighted

Call Graphs and Flow Graphs graphically represent software function blocks, control flow paths, and data flow information. Data flow diagrams document how data input is transformed to output, with each stage in the diagram representing a distinct transformation. Annotated source code provides information on how the data is transferred between each module.

Decision/Truth tables provide a clear and coherent specification and analysis of complex logical combinations and relationships. This method uses two-dimensional tables to concisely describe logical relationships between Boolean program variables. The MC/DC test case planner provides guidance in the selection of appropriate test values with the aim of achieving 100% MC/DC coverage.

IEC 61508:2010-3 Annex B Table B.8: “*Static analysis*”

Referenced by tables A.1, A.2 and A.4

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Boundary value analysis	C.5.4	R	R	HR	HR
2	Checklists	B.2.5	R	R	R	R
3	Control flow analysis	C.5.9	R	HR	HR	HR
4	Data flow analysis	C.5.10	R	HR	HR	HR
5	Error guessing	C.5.5	R	R	R	R
6a	Formal inspections, including specific criteria	C.5.14	R	R	HR	HR
6b	Walk-through (software)	C.5.15	R	R	R	R
7	Symbolic execution	C.5.11	---	---	R	R
8	Design review	C.5.16	HR	HR	HR	HR
9	Static analysis of run time error behaviour	B.2.2, C.2.4	R	R	R	HR
10	Worst-case execution time analysis	C.5.20	R	R	R	R

”HR” The method is highly recommended for this SIL.
“R” The method is recommended for this SIL.
“---” The method has no recommendation for or against its usage for this SIL.

Figure 26: Copy of IEC 61508-3 Table B.8, with techniques and measures supported by the LDRA tool suite highlighted

IEC 61508-7 Section B.2.2 and Section C.2.4, both called “*Formal methods*”, describe the static analysis of run time behaviour. This analysis can reveal issues such as uninitialized local variables, the illegal dereferencing of pointers, uninitialized pointers, uninitialized variables, the non-termination of loops, and the non-termination of calls.

Formal inspection is a structured process performed by the peers of the person(s) responsible for the creation or maintaining of the software in question. The purpose of formal inspection is to identify and resolve defects such that the quality and reliability of the software is enhanced. Entry and exit criteria should be defined based on the properties required for the software element, and static analysis can play its part by providing information on those entry and exit criteria.

IEC 61508:2010-3 Annex B Table B.9: "*Modular approach*"

Referenced by table A.4

Technique/Measure		Ref	SIL			
			1	2	3	4
1	Software module size limit	C.2.9	**HR**	**HR**	**HR**	**HR**
2	Software complexity control	C.5.13	**R**	**R**	**HR**	**HR**
3	Information hiding/encapsulation	C.2.8	**R**	**HR**	**HR**	**HR**
4	Parameter number limit / fixed number of subprogram parameters	C.2.9	**R**	**R**	**R**	**R**
5	One entry/one exit point in subroutines and functions	C.2.9	**HR**	**HR**	**HR**	**HR**
6	Fully defined interface	C.2.9	**HR**	**HR**	**HR**	**HR**

"HR" The method is highly recommended for this SIL.
"R" The method is recommended for this SIL.
"---" The method has no recommendation for or against its usage for this SIL.

Figure 27: Copy of IEC 61508-3 Table B.9, with techniques and measures supported by the LDRA tool suite highlighted

Static analysis can be used to derive a number of complexity metrics to provide a mechanism to control complexity as specified. These include:

- McCabe's Cyclomatic Complexity
- Essential Cyclomatic Complexity
- Knots
- Essential Knots
- Nesting Level
- Halstead's Size Metrics
- Unreachable Code / Dead Code
- Infeasible Code
- LCSAJ Density
- C++ OO Metrics
- Entry/Exit Points
- Procedure/Interface

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