LDRA
Software Technology

Technical White Paper

Verification of Airborne Software in Compliance with DO-178C

Working with the Airborne Software Industry to Meet the Challenges of Achieving Cost-effective Certification

www.ldra.com

Contents

Background

In response to the increased use of software in airborne systems, the Radio Technical Commission for Aeronautics organization[1] (now known as RTCA, Inc.) in collaboration with EUROCAE[2], created the guidance document DO-178 "Software Considerations in Airborne Systems and Equipment Certification." This document has come to be accepted as the international certification standard for airborne software. Originally published in 1982, re-written in 1992 as DO-178B and significantly extended in 2011 to address modern technologies and methodologies in DO-178C, the standard reflects the experience accrued to meet today's aviation industry needs.

LDRA has participated extensively on both the DO-178B[3] and DO-178C[4] committees over nearly two decades. Mike Hennell, LDRA's CEO, was instrumental in the inclusion of several test measurement objectives in the standard, including those relating to structural coverage analysis. The LDRA tool suite® was a forerunner in automated verification for certification to both the DO-178B standard for airborne software systems, and to its companion standard, DO-278[5] for ground-based systems.

The DO-178C standard provides detailed guidance for the development and verification of safety critical airborne software. In accordance with ARP 4754A[6], prior to system development, functional hazard analyses and system safety assessments are performed to determine the contribution of the system to potential failure conditions. The severity of failure conditions on the aircraft and its occupants are then used to determine a Design Assurance Level (DAL), as shown in Figure 1.

DAL	Failure Condition	Description
A	Catastrophic	Failure Conditions, which would result in multiple fatalities, usually with the loss of the airplane.
B	Hazardous	Failure Conditions, which would reduce the capability of the airplane or the ability of the flight crew to cope with adverse operating conditions to the extent that there would be: • A large reduction in safety margins or functional capabilities; • Physical distress or excessive workload such that the flight crew cannot be relied upon to perform their tasks accurately or completely, or • Serious or fatal injury to a relatively small number of the occupants other than the flight crew
C	Major	Failure Conditions which would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to the flight crew, or physical distress to passengers or cabin crew, possibly including injuries.
D	Minor	Failure Conditions which would not significantly reduce airplane safety, and which involve crew actions that are well within their capabilities. Minor Failure Conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as routine flight plan changes, or some physical discomfort to passengers or cabin crew.
E	No effect	Failure Conditions that would have no effect on safety; for example, Failure Conditions that would not affect the operational capability of the airplane or increase crew workload.

Figure 1: : Design Assurance Levels as described in DO-178C (Table 2-1)[7]

[1] http://www.rica.org/

[2] https://www.eurocae.net/

[3] http://www.rica.org/store_product.asp?prodid=581

[4] http://www.rica.org/store_product.asp?prodid=803

[5] http://www.rica.org/store_product.asp?prodid=678%20%20

[6] http://standards.sae.org/arp475a/

[7] Based on table 2-1 from RTCA DO-178C, Copyright © 2011 RTCA, Inc. All rights acknowledged

The ARP 4754A development process then allocates the associated DALs to the subsystems that implement the system's electronic hardware and software requirements. DO-178C establishes five "software levels" and modulates objectives that must be satisfied. This means that the effort and expense of producing a system critical to the continued safe operation of an aircraft (e.g. a flight control system) is necessarily higher than that required to produce a system with only a minor impact on the aircraft in the case of a failure (e.g. a bathroom smoke detector).

DO-178C covers the complete software lifecycle: planning, development and integral processes to ensure correctness and robustness in the software. The integral processes include software verification, software quality assurance, configuration management assurance and certification liaison with the regulatory authorities.

The standards do not oblige developers to use analysis, test, and traceability tools in their work. However, these tools improve efficiency in all but the most trivial projects to the extent that they have a significant part to play in the achievement of the airworthiness objectives for airborne software throughout the development lifecycle. Specialised tools exemplified by the LDRA tool suite are used to help achieve DO-178C objectives including bi-directional traceability, test management, source code static analysis, and dynamic analysis of both source and object code.

This document describes the key software development and verification processes of the standard and to show how automation can help to lower the cost of development and verification and to ensure the deployment of safety critical software.

DO-178B Process Objectives

DO-178C recognizes that software safety must be addressed systematically throughout the software life cycle. This involves life cycle traceability, software design, coding, validation and verification processes used to ensure correctness, control and confidence in the software.

Key elements of the DO-178C software life cycle include the practices of traceability and structural coverage analysis. Bi-directional traceability must be established across the lifecycle, from system requirements to software high-level requirements, from software high-level requirements to low-level requirements, and through to test cases, test procedures and test results. Low-level requirements must then be linked to the source code in which they are implemented. Structural coverage analysis (code coverage, data coupling and control coupling) quantifies the extent to which the source code of a system has been exercised by the testing process. Using these practices, it is possible to ensure that code has been implemented to address every system requirement and that the implemented code has been tested to completeness.

The use of software tools offers particularly significant benefits during software development and software verification, discussed in sections 5.0 and 6.0 of the standard respectively.

DO-178C Section 5.0: SOFTWARE DEVELOPMENT PROCESSES

Five high-level processes are identified in the DO-178C SOFTWARE DEVELOPMENT PROCESSES section; Software Requirements Process (5.1), Software Design Process (5.2), Software Coding Process (5.3), Integration Process (5.4), and Software Development Process Traceability (5.5).

The ideal tools for requirements management (Section 5.1) depend 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 tool such as IBM Rational DOORS[8], Siemens Polarion PLM[9], or more generally, similar tools offering support for standard Requirements Interchange Formats[10].

[8] http://www.-03ibm.com/software/products/en/ratidoor

[9] http://polarion.pim.automation.siemens.com/

[10] http://www.omg.org/spec/ReqIF/

The products of the design phase (Section 5.2) potentially include Model Based Designs, 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 requirements, these design artefacts, and subsequent development phases generally causes a project management headache. This is addressed by section 5.5 of the standard, discussed later.

As part of Section 5.3, DO-178C specifies that software must meet certain software coding process objectives. These objectives state the development of source code should implement low-level requirements and conform to a set of software coding standards.

Further definition of the software coding standards is provided in Section 11.8 of DO-178C:

- Programming language(s) to be used and/or defined subset(s). For a programming language, establish an approach to unambiguously define the syntax, the control behaviour, the data behaviour and side-effects of the language. This may require limiting the use of some features of a language.
- Source code presentation standards including line length restriction, indentation, and blank line usage.
- Source code documentation standards, for example, name of author, revision history, inputs and outputs, and affected global data.
- Naming conventions for components, subprograms, variables and constants.
- Conditions and constraints imposed on permitted coding conventions, such as the degree of coupling between software components and the complexity of logical or numerical expressions and rationale for their use.
- Constraints on the use of coding tools.

Static analysis tools automate the “inspection” of the source code, making compliance checking easier, less error prone and more cost effective by comparing the code under review with the rules dictated by the chosen a software coding standard (Figure 2). Non-conformances are highlighted as required by section 6.4.3d of the standard. The tool suite can also assess the complexity of the code under review to ensure that it stays below a safe threshold for the system, and its data flow analysis facility can be used to identify any uninitialized or unused variables and/or constants as specified by section 6.4.3.f.

TunnelData::Cell::InitialiseCell				
DU anomaly, variable value is not used.	2	Required	70 D	MISRA-C++:2008 0-1-6,0-1-9
Local variable should be declared const.	2	Required	93 D	MISRA-C++:2008 7-1-1
Array has decayed to pointer. : pLampTypeIDs		Required	534 S	MISRA-C++:2008 5-2-12
No brackets to loop body.		Required	11 S	MISRA-C++:2008 6-3-1
TunnelData::Cell::SetEmergencyOutputLevel				
DU anomaly, variable value is not used.	2	Required	70 D	MISRA-C++:2008 0-1-6,0-1-9
Local variable should be declared const. : ThisType		Required	93 D	MISRA-C++:2008 7-1-1
Logical conjunctions need brackets. (analysed w. JSF++ AV)		Required	49 S	MISRA-C++:2008 5-0-2,5-2-1
Expression needs brackets. (analysed w. JSF++ AV)		Advisory	361 S	MISRA-C++:2008 5-0-2

Figure 2: Checking for compliance with the MISRA C++:2008 coding standard using the LDRA tool suite

There are many pre-defined language subsets (sometimes called “coding standards”) available for C, C++ and Ada languages (sidebar), and nothing to suggest that an in-house subset could not be preferred. This could be established entirely from scratch, or more pragmatically, based on an established subset with modifications to suit a particular project. It is important that a deployed static analysis tool should be similarly flexible.

In section 5.5, DO-178C mandates that the correctness of the requirements-based development and verification process is determined by requirements coverage or traceability. This analysis assures that requirements are bi-directionally associated between system and high-level requirements, high-level and low-level requirements, and finally low-level requirements and source code.

Static analysis doesn’t only provide a useful check against coding standards. It also reveals the underlying structure of the software, which is required to confirm that traceability.

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
CERT
CWE

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

Ada
Ravenscar
Spark

Java
CWE
CERT J

If everything follows the development lifecycle in textbook fashion, that is perhaps a one-off, trivial task. The requirements will never change and tests will never throw up a problem. But unhappily, that is rarely the case.

Consider, then, what happens if a problem is highlighted during static analysis.

- Perhaps 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?
- Maybe there is a low-level requirement that is not traceable through to a high-level requirement. What needs to change to resolve that?
- Maybe an end user has a new requirement. What impact will that have on established requirements, design and source code?

Issues surrounding functionality are likely to be flushed out during dynamic analysis later in the life cycle, meaning that this cause-and-effect puzzle becomes even more complex whenever something needs to change.

The use of a requirements traceability/coverage and test management solution that is integrated with code review, data and control coupling analysis, and low-level testing and code coverage tools takes away the project management challenges associated with such complexity (Figure 3). It ensures that the requirements traceability matrix even through disparate repositories and down to the source code and test cases is a great deal simpler to manage, more cost effective, and permanently up-to-date.

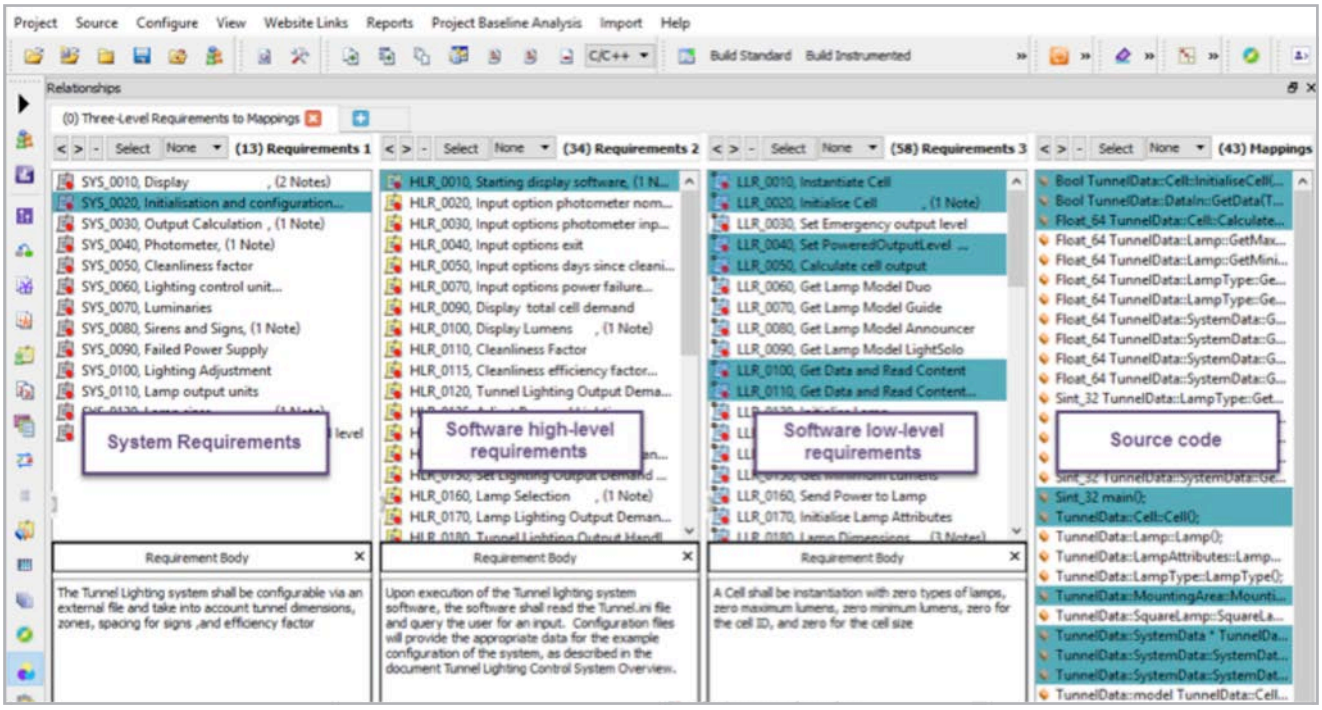


Figure 3: Automating requirements traceability with the TBmanager component of the LDRA tool suite

DO-178C Section 6.0: SOFTWARE VERIFICATION PROCESSES

In contrast to static analysis, which can be thought of as an automated “inspection” of the source code, dynamic analysis involves executing the Executable Object Code (EOC) piecemeal or in its entirety, using a target environment representative of that to be deployed in the completed application. This execution is used to provide evidence both of correct functionality, and of the parts of the code exercised (“structural coverage”).

DO-178C discusses both of these concepts, identifying objectives to achieve test coverage of high and low level requirements, and to achieve appropriate test coverage of both the software structure, and the data and control coupling.

The "*test cases and procedures*" referenced in the standard could include low-level tests (sometimes referred to as unit tests), integration tests, or system tests, and probably a combination of all three.

Low-level tests are designed to verify the implementation of low-level requirements. Test procedures need to be authored, reviewed, and executed to ensure the software does not contain any undesired functionality. Low-level tests can then be executed on the target hardware or simulated environment as specified in the Software Verification Plan (SVP). Once the test procedures are executed actual outputs are captured and compared with the expected results, and pass/fail results reported (Figure 4).

Software integration testing is designed to verify the interrelationships between the software components with respect to both the requirements the software architecture. In practice, the mechanisms used for low-level testing are often extended to use verify behaviour in a call tree.

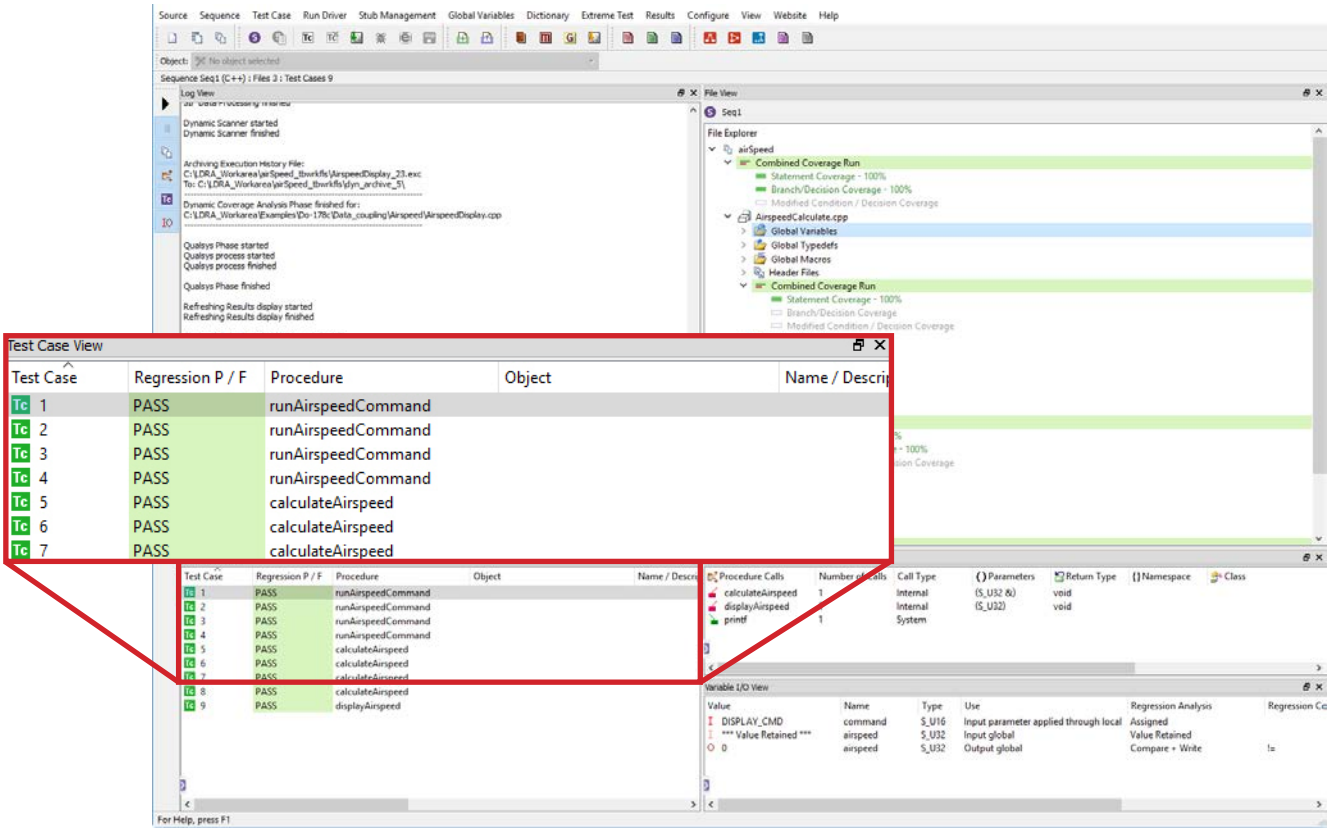


Figure 4: Automating low-level and integration testing with the TBrun component of the LDRA tool suite

Should changes become necessary – perhaps as a result of a failed dynamic test, or in response to a requirement change - then all impacted low-level 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.

Keeping track of the project status in such flux is challenging. Automating the maintenance of the bi-directional relationship between the products of the different development phases saves a great deal of time, and makes errors much less likely – not just as far the development of the requirements and source code, but through to requirements-based testing and test coverage for both high and, where necessary, low-level requirements.

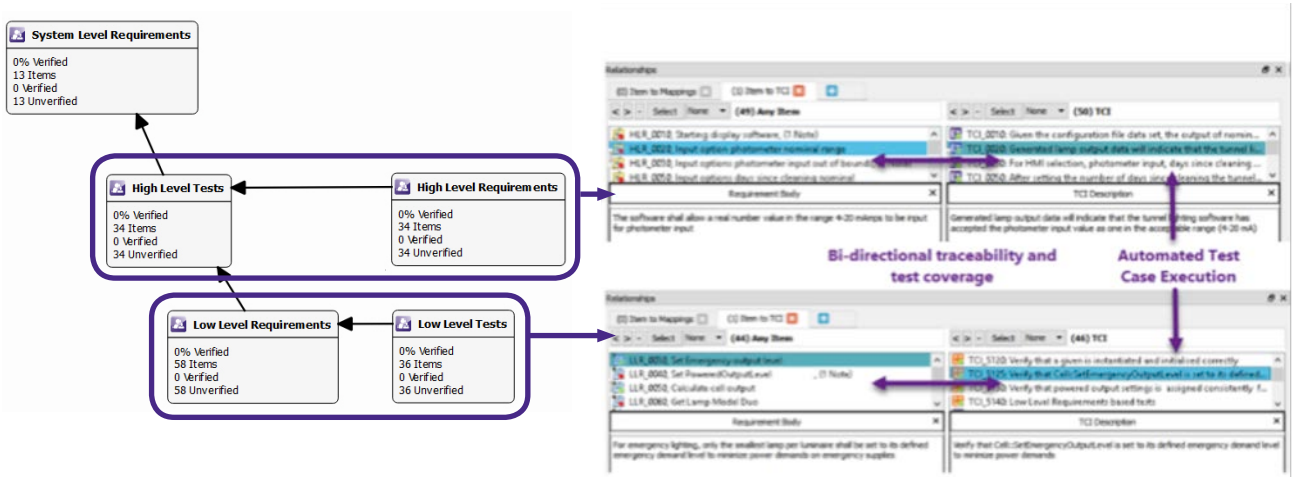


Figure 5: Tracing requirements with the Graphical User Interface (GUI) of TBmanager, a component of the LDRA tool suite

The area on the left of Figure 5 shows how a graphical representation of the traceability policy between requirements and tests cases of different scopes. Requirements and test cases are then authored or imported and linked together, so completeness in requirements decomposition and test coverage can be assessed. Test cases can be reviewed and developed based on requirements identifying and filling gaps in requirements coverage quickly and easily.

Bi-directional analysis can also identify "orphan" test cases that are not linked to requirements, highlighting the need to make appropriate changes to requirements, test cases, or traceability relationships. It can provide analysis in relation to changes in requirements, tests, or code and their potential impact on timescales. It can provide the intelligence for regression testing to be targeted, minimizing the incremental verification and review burden. And it can provide required evidential artefacts such as traceability matrices, to show that test coverage of high-level and low-level requirements has been achieved[11].

Figure 6 shows a Traceability matrix between high-level requirements and functional test cases. Thirty-three out of thirty-four requirements have an associated test case, so the test coverage of high-level requirements objective is still unfulfilled. This level of transparency is essential to ensure that all requirements have associated test cases.

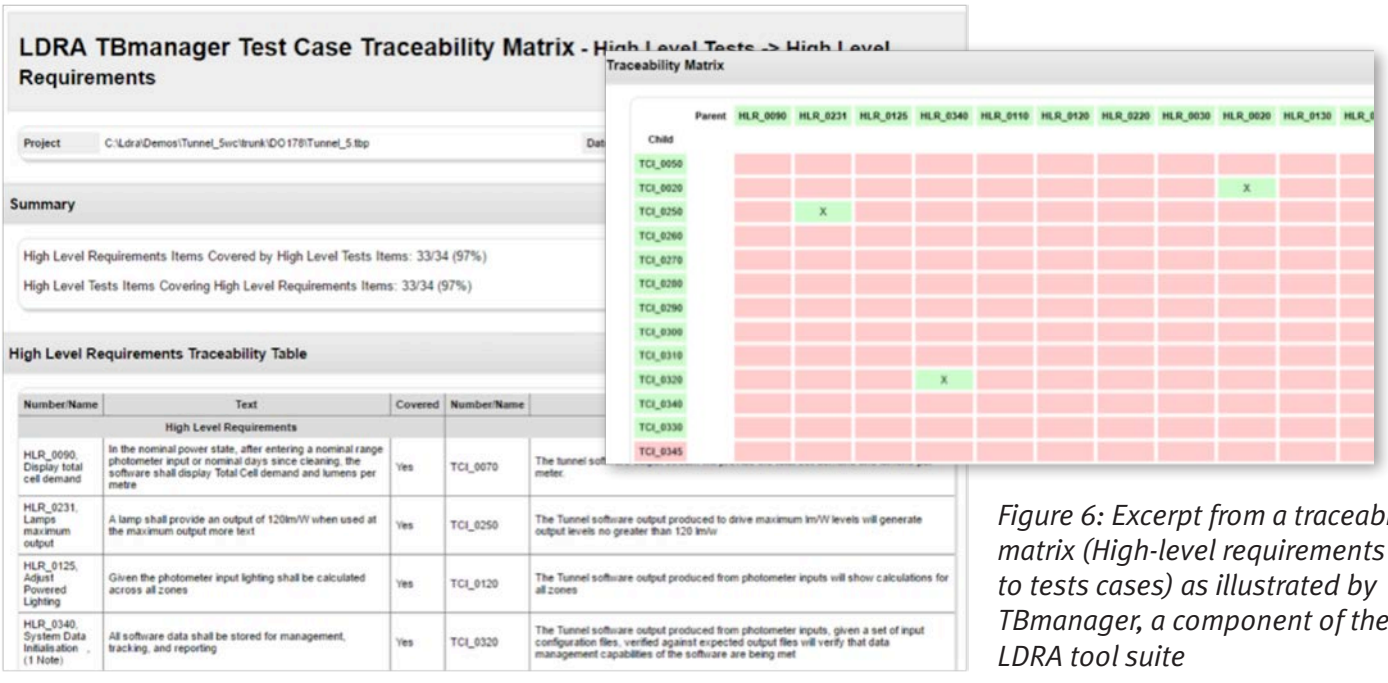


Figure 6: Excerpt from a traceability matrix (High-level requirements to tests cases) as illustrated by TBmanager, a component of the LDRA tool suite

[11] Table A-7 objectives 3 and 4 from RTCA DO-178C, Copyright © 2011 RTCA, Inc. All rights acknowledged.

DO-178C Structural Coverage Analysis Objectives

Structural Coverage (SC) is used to identify which code structures and component interfaces have been exercised during the execution of requirements-based test procedures, facilitating the empirical measurement of requirements-based test effectiveness. As the name implies, Structural Coverage Analysis (SCA) involves the scrutiny of the SC to determine if there are any parts of the code which have not been sufficiently exercised, and if not, why.

The achievement of objectives A-7.5, 6, and 7 (Figure 8) involves the collation of structural coverage metrics, typically by "instrumenting" a copy of the source code – that is, superimposing it with function calls to collate coverage data – and executing that instrumented code using requirement based test cases. These test cases primarily reference high-level requirements, supplemented by low-level requirements as needed.

SCA is then applied to assess the effectiveness of this testing by measuring how much of the code has been exercised. Coverage of portions of code unexecuted thus far may require additional test cases or modifications to existing test cases, changes to requirements, removal of dead code, or perhaps the identification of deactivated code and resulting unintended functionality. An iterative "review, analyse, verify" cycle is typically needed to ensure that software coverage is achieved and low-level requirements are verified, and graphical representation can be a great help.

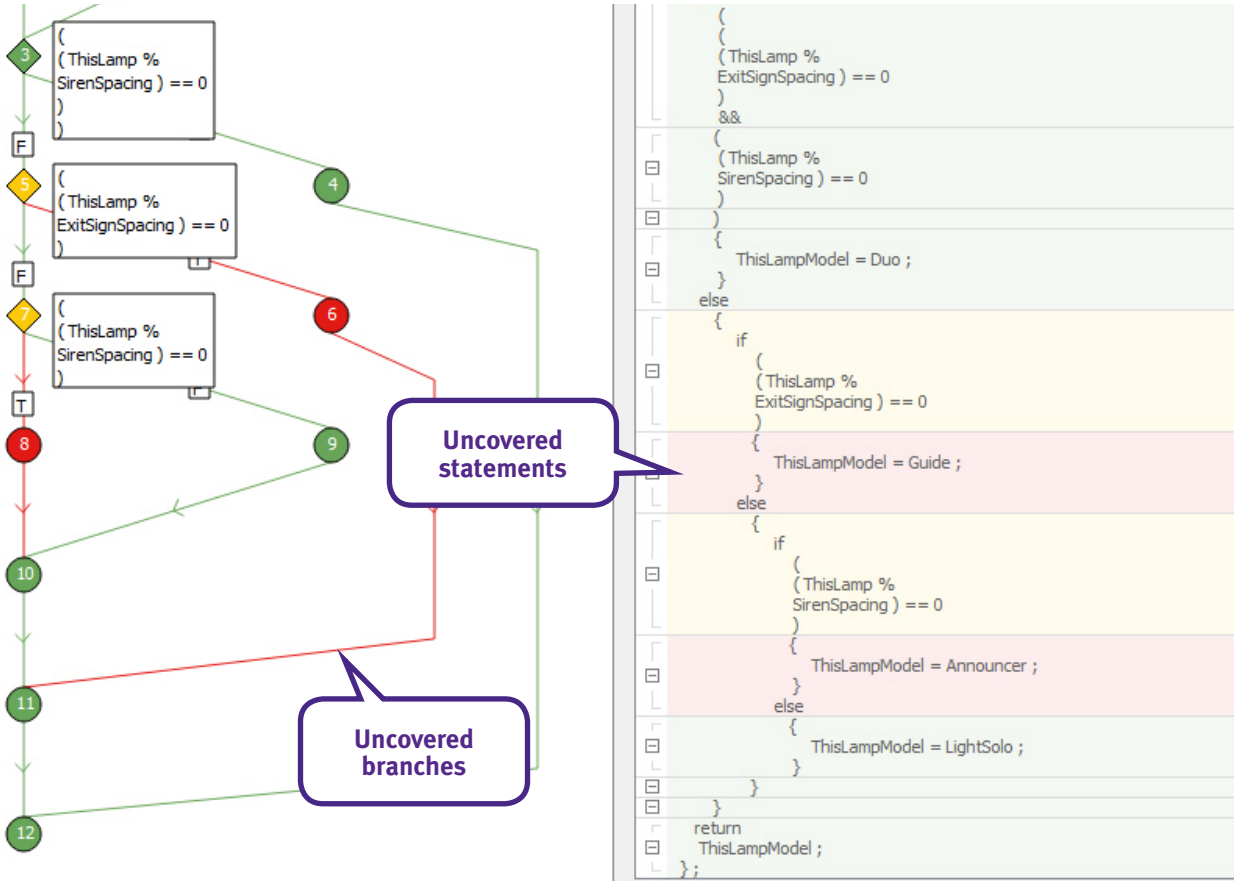


Figure 7: Graphical visualisation of code coverage in a flow graph in the LDRA tool suite

System requirements can be shown to have been correctly decomposed, implemented, and verified by combining a complete trace from requirements through to code and test cases, with the achievement of comprehensive functional test coverage and structural coverage objectives.

Item	Description	DO-178C Reference	DO-178C Levl A	DO-178C Level B	DO-178C Level C	DO-178C Level D
5	Test coverage of software structure (MC/DC) is achieved	6.4.4.2	✓	Not Required	Not Required	Not Required
6	Test coverage of software structure (decision coverage) is satisfied	6.4.4.2a 6.4.4.2b	✓	✓	Not Required	Not Required
7	Test coverage of software structure (statement coverage) is satisfied	6.4.4.2a 6.4.4.2b	✓	✓	✓	Not Required
8	Test coverage of software structure (data coupling and control coupling) is achieved	6.4.4.2c	✓	✓	✓	Not Required
9	Verification of additional code, that cannot be traced to Source Code, is achieved	6.4.4.2d	✓	Not Required	Not Required	Not Required
Note: Items 5, 6, 7 and 8 are not required for DO-178C Levels D and E. Items 1 to 4 (not shown) are manual procedures.						

✓ Satisfied by the LDRA tool suite, which can be used to satisfy the 'with Independence' requirement.

Figure 8: SCA Objectives for Each Software Level [12]

Figure 8 shows that DO-178C objectives A7-5, 6, and 7 relate to the achievement of 100% MC/DC, decision, and statement coverage respectively. The required combination of those objectives depends on the design assurance level.

For Level A systems, structural coverage at the source level isn't enough. Compilers often add additional code or alter control flow, and often their behaviour is not deterministic. To ensure that functionality is not compromised, DO-178C 6.4.4.2.b states:

"if the software level is A and a compiler, linker, or other means generates additional code that is not directly traceable to Source Code statements, then additional verification should be performed to establish the correctness of such generated code sequences".

An automated mechanism to provide evidence of that verification can make that process much more efficient. Because there is a direct one-to-one relationship between object code and assembly code, one way for a tool to represent this is to display a graphical representation of the source code alongside the equivalent representation of the assembly code. Object Code Verification (OCV) measures code coverage at both the source and the assembly level by instrumenting each in turn (Figure 9).

This approach provides a means for the demonstrable and deterministic verification of the Executable Object Code (EOC) on the target system. For OCV to be effective, it therefore needs to support the microprocessor, associated instruction set, and compiler deployed on that system.

[12] Based on table A-7 from RTCA DO-178C, Copyright © 2011 RTCA, Inc. All rights acknowledged.

Three discrete modes are used for each test case to quickly identify the "additional code" referenced in the standard and dramatically reduce laborious manual analysis.

1. The test case is executed without instrumentation to confirm correct functionality.
2. The test case is executed by leveraging instrumentation at the source code level.
3. Finally, the test case is executed with instrumentation at the assembly code level to identify any uncovered statements or branches that may have been inserted or altered during the compilation and linking process.

Typically, a few additional requirements based tests can be added to verify this additional code to meet objective A7-9 (Figure 7).

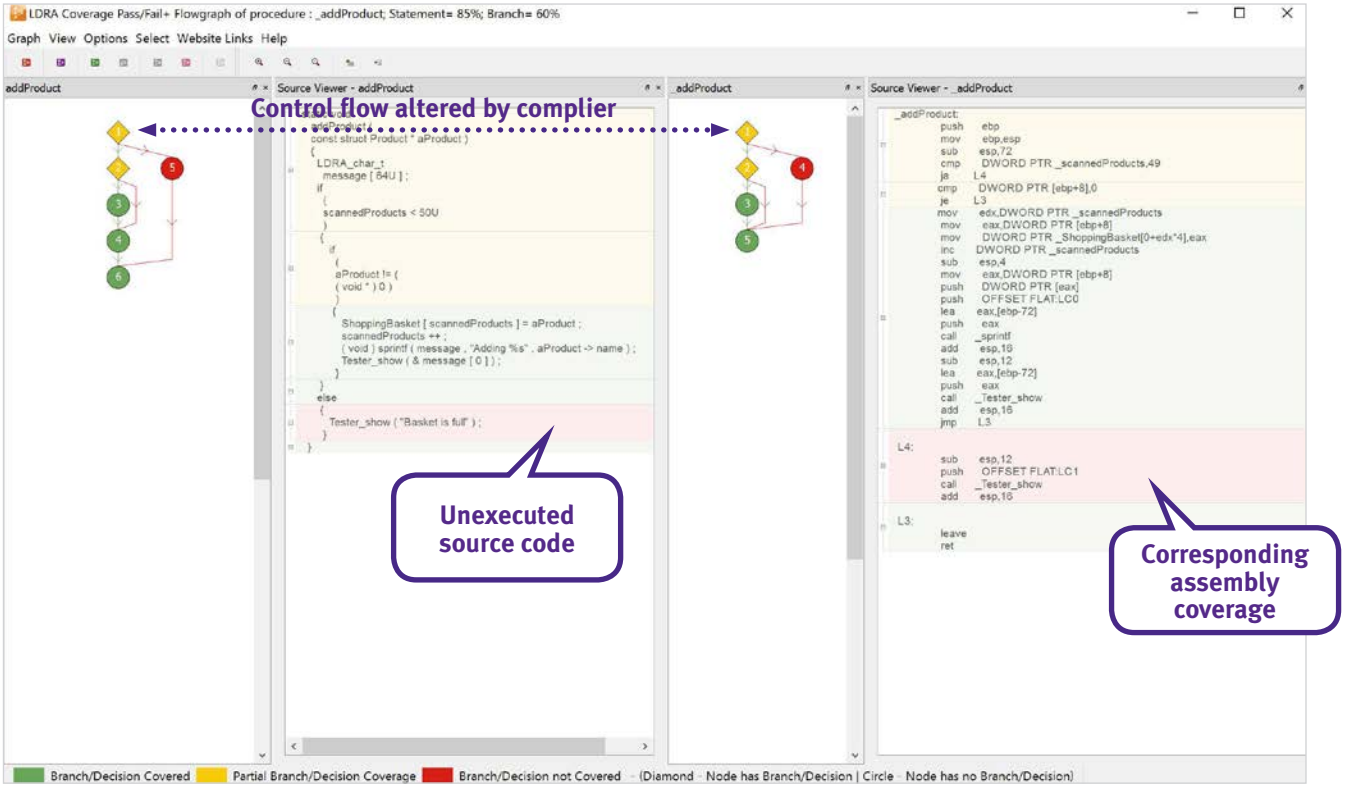


Figure 9: Visualization of control flow and code coverage in C and associated assembly code in the LDRA tool suite

Automated source code instrumentation and coverage data analysis reduces space and time overhead which in turn makes the technology scalable and adaptable to a wide array of cross compilers, targets, in-circuit emulators, and other embedded environments. Target integrations are highly extensible and support processors from simple 8 bit devices to high-performance multi-core architectures, IDEs, and I/O integrations.

Demonstrating Data Coupling and Control Coupling

In the evolution of the standard from DO-178B to DO-178C, there was a change of emphasis in how data and control should be demonstrated.

DO-178B Section 6.4.4.2.c required that "analysis should confirm the data coupling and control coupling between the code components."

DO-178C Section 6.4.4.2.c requires "analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components."

DO-178C therefore changes the DCCC objective from "an analytical exercise against the test design to a measurement exercise against the test execution"[13].

[13] White Paper Object Code Verification and DO-178C Objective A7-9 v2.1 (Available on http://www.ldra.com/whitepapers)

This essentially means that data coupling and control coupling analysis needs to be performed post execution, and generated artefacts reviewed against the system requirements and architecture. That, in turn, places a new burden on the development cycle.

Control Coupling

Control Coupling is defined by DO-178C as "The manner or degree by which one software component influences the execution of another software component." Procedure/functional call coverage reporting can be presented as illustrated in Figure 7, or in report format for archival and audit purposes. Both the visual and reporting approaches help to identify any gaps and guide targeted verification activities.

SCA and associated artefacts provide visibility and data to perform these activities and meet the associated objective A-7.8 (Figure 7).

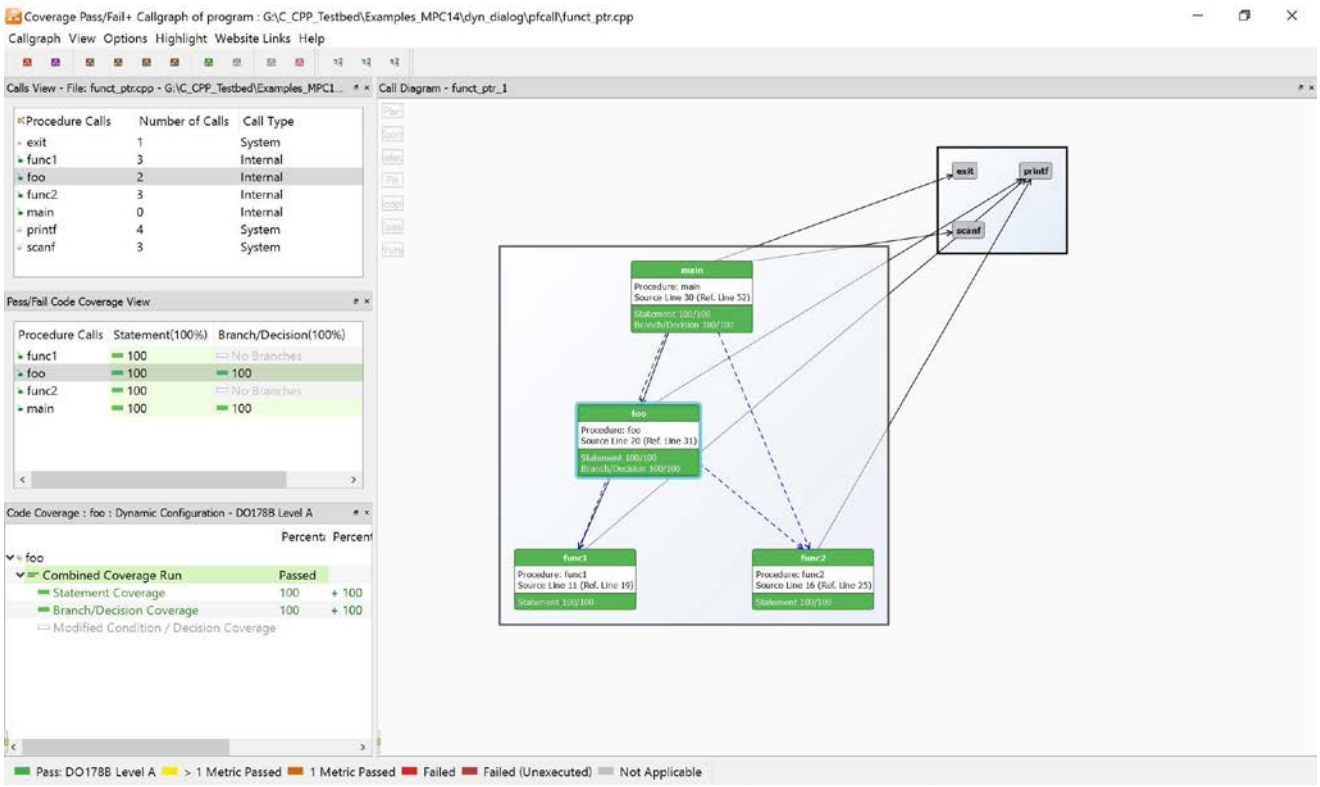


Figure 10: Procedure/function call coverage as seen in call graphs generated by the LDRA tool suite

Data Coupling

Data coupling is defined by DO-178C to be "The dependence of a software component on data not exclusively under the control of that software component". Objective A-7.8 requires that "Test coverage of software structure, both data and control coupling, is achieved." As with control coupling analysis, any dataflow measurements must be derived from the execution of requirements based tests.

The example in Figure 11 is duplicated from DO-248C, and its implementation is illustrated in the function runAirspeedCommand on the right. The expected behaviour of this source code is to first calculate the airspeed and then display it, in that order.

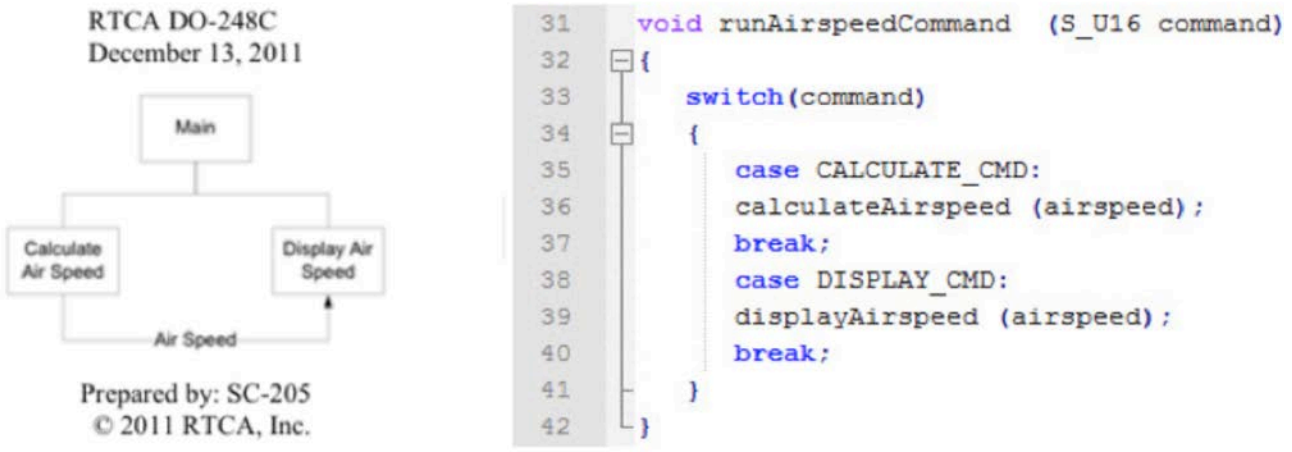


Figure 11: Example from DO-248C[14]

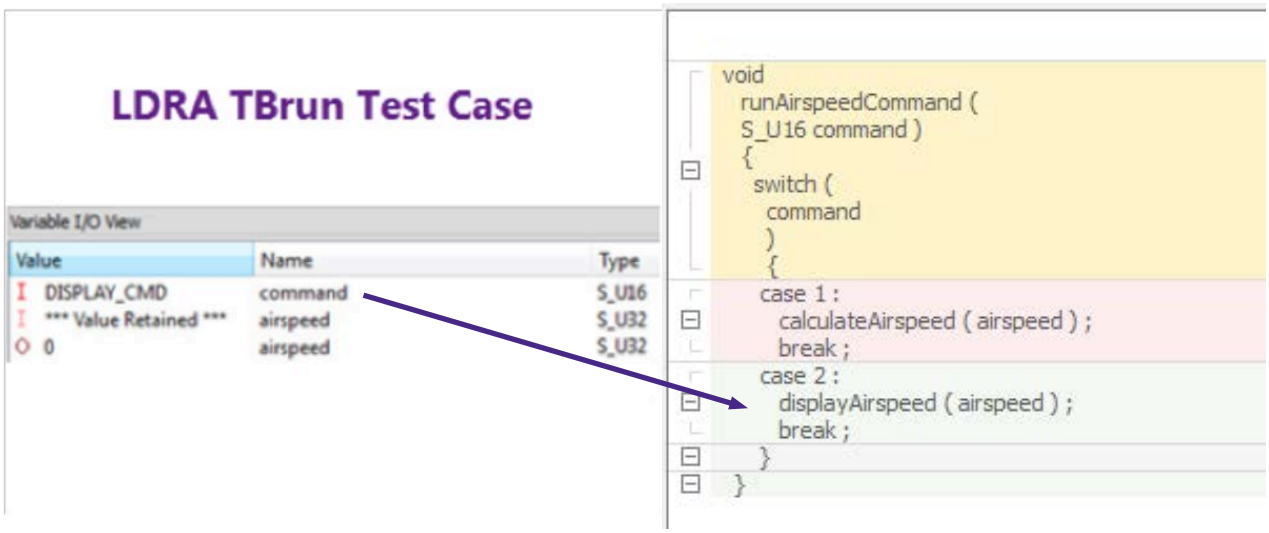


Figure 12: Test case exercising runAirspeedCommand with the resulting control flow and structural coverage represented in green. Example from TBrun, a component of the LDRA tool suite

The test case in the figure above exercises the second case in the switch statement as reflected by the structural coverage below. It also shows that display is called without updating the airspeed to its latest value. Additional test cases may invoke the calculateAirspeed command but not necessarily after a call to *displayAirspeed*.

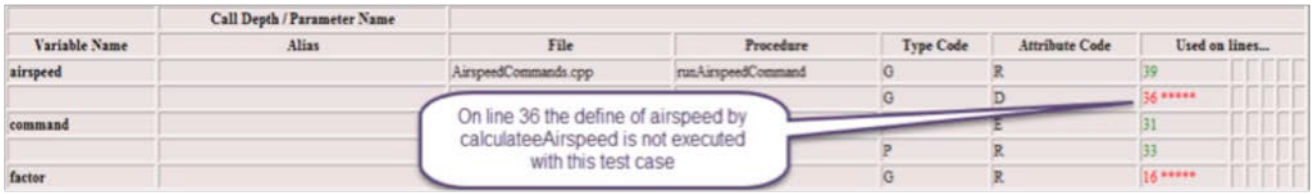


Figure 13: LDRA tool suite report showing unreferenced variable in run time. These artefacts are used to meet objective A-7.8

The report above was generated from executing the test case above. It shows dynamic data flow information revealing that airspeed was in fact not written to on line 36 and wasn't updated before it was displayed, potentially displaying inaccurate information. In general, the observed data flow provides the information required to reconcile the data interaction, requirements and architecture, and the behaviour of the application.

[14] Extracted from RTCA DO-178C, Copyright © 2011 RTCA, Inc. All rights acknowledged.

Data coupling analysis is focused on the observation and analysis of data elements, such as airspeed, as they are set and used ("set/use pairs") across software component boundaries. Manually performing these exercises with debuggers is labour intensive, difficult to repeat, and error prone. Automating the activity dramatically reduces that overhead.

Object-Oriented Technology

In the early 2000s, object-oriented technology was viewed in the commercial avionics space as novel and unproven. Around that time the Certification Authorities Software Team (CAST) published papers (CAST 4 and 8 in 2000 and 2002) to enumerate concerns and limitations.

As DO-178B was updated to DO-178C it was decided that these concerns, vulnerabilities, and subsequent additional objectives associated with object-oriented technologies would be addressed not by the original standard but rather a supplement, DO-332[15]. This new supplement describes concepts and key features of object-oriented technologies and related techniques, discusses their impact on the planning, development, and verification processes, and enumerates their vulnerabilities.

OO Objectives

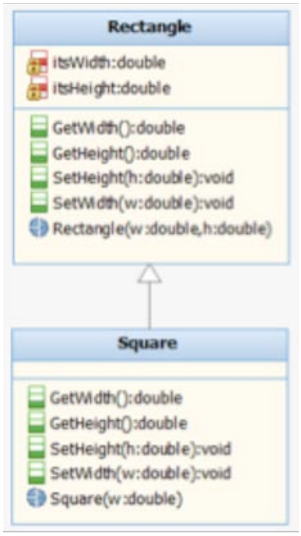
Two objectives were included in the DO-332 supplement:

- A-7 OO.10 Verify local type consistency (section OO.6.7.1)
- A-7 OO.11 Verify the use of dynamic memory management is robust

It is useful to understand Liskov's Substitution Principle in relation to the first of these.

"Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T"

In object-oriented languages, inheritance allows the behaviour of superclasses to be overridden by subclasses. As described in DO-332 FAQ #14, ensuring safe use of inheritance, method override, and dynamic dispatch is challenging as the nature of these techniques can make it unclear from a simple review which method is executed at any call point in a program. Overridden behaviour in instantiated subclasses may alter the behaviour beyond the intended scope of the superclass and violate type consistency Figure 14. As DO-332 section OO.6.7.1 further describes, this means that the preconditions of the parent class must not be strengthened, and the postconditions and invariants defined on the state of a class must not be weakened.




```
virtual void Rectangle::SetWidth (double w) {
    itsWidth=w;
}

    Post condition: itsHeight does not change

void Square::SetWidth (double w) {
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

    Post condition: itsHeight set to w
```

Figure 14: Parent and child class showing code that violates type consistency

[15] RTCA DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A

From a verification standpoint, DO-332 OO.6.7.2 suggests that one of the following activities must be performed:

- Verify substitutability using formal methods.
- Ensure that each class passes all the tests of all its parent types which the class can replace
- For each call point, test every method that can be invoked at that call point (pessimistic testing)

The first of these applies to the small minority of development teams who are using formal methods, whilst the third (once commonly referred to as flattened class testing) requires that each possible dispatch is tested at every call point in a program. That can easily cause a combinatorial explosion of test cases, dramatically increasing the verification burden.

That leaves the second option as the most practical for most people - to ensure type consistency, without the burden of pessimistic testing. Doing so requires that each class and its methods must pass all tests of every superclass for which it can be substituted (DO-332 FAQ #34).

Figure 14 shows that the superclass Rectangle and its methods explicitly set the height and width with respective methods, but the Square class "shortcuts" that by setting them both in the *SetWidth* function. Since a square's height and width are the same, this seems acceptable. However, because type consistency is violated, test cases for the SetWidth method for the Rectangle class may not pass those for its subtype Square.

Reusing test cases from the parent class Rectangle on the subclass Square will highlight such type consistencies (Figure 15).

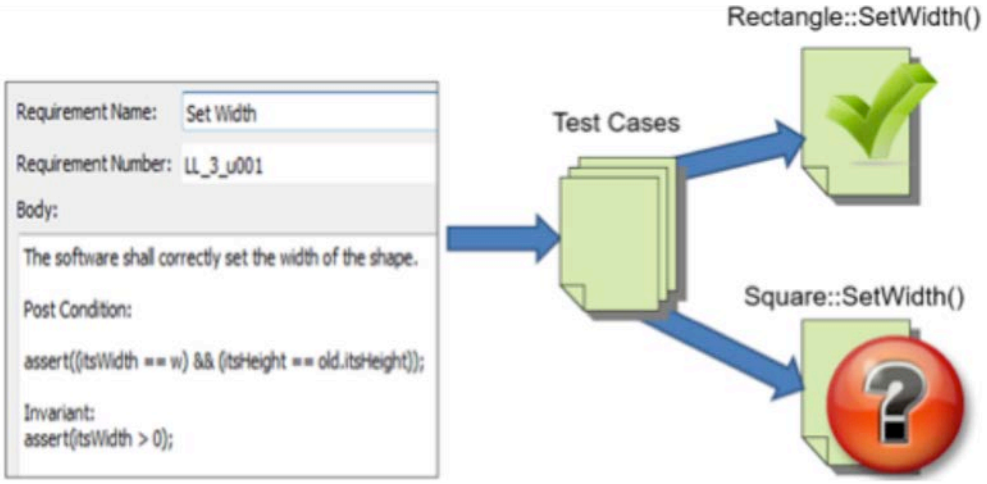


Figure 15: The Rectangle Class test cases are applied to its Square subclass to ensure local type consistency

A negative test case for the Rectangle class shows that as expected, setting the width has no impact on the height. When that same negative test is applied to the Square class's SetWidth method, one can clearly see that itsHeight is changed, violating type consistency (Figure 16).

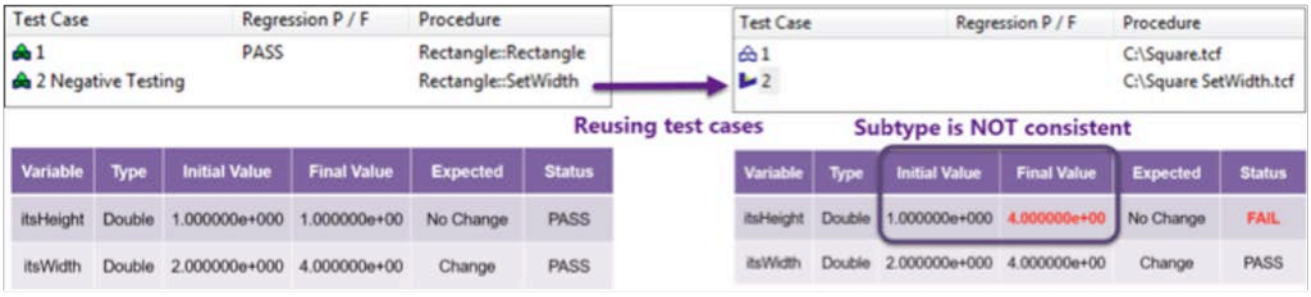


Figure 16: Parent class test cases being reused to test a subclass to detect an inconsistent subtype

A range of static and dynamic analysis techniques can be deployed in order to fulfil DO-332 A-7 OO.6.8.1 "Verify the use of dynamic memory management is robust" and the related vulnerabilities outlined in Annex OO.D.1.6.1.

Tracking memory allocation and deallocation helps to ensure the proper freeing of memory, as do associated checks prior to dereferencing. Low-level testing provides a mechanism to explore various allocation/deallocation scenarios to help ensure that vulnerabilities described in (OO.D.1.6.1) are addressed. Timing hooks within the low-level tests help characterize allocation/deallocation timing and dynamic data flow analysis tracks references and updates of data elements in runtime to detect lost updates and stale references.

Other Considerations

When using object-oriented technologies or related techniques, there are various other factors to consider:

Source to Object traceability

As mentioned in OO.D.1.2.1, source to object code traceability may be more difficult to correlate in object-oriented languages. OCV solutions provide a graphical comparison of assembly code coverage and high-order language coverage (i.e. C++) to ensure that the source coverage data accounts for variations in the structure of executable object code (EOC) as compared to the source code.

Traceability to child classes

OO.5.2.2.i states "Develop a locally type consistent class hierarchy with associated low-level requirements whenever substitution is relied upon". In other words, a requirement that traces to a method implemented in a class should also trace to the method in its subclasses when the method is overridden in that subclass (FAQ #9). Static analysis and code visualization exposes inheritance relationships within the analysed code, making traceability gaps across class hierarchies easier to detect and remedy.

Coding standard for object-oriented languages

Languages such as C++ allow for tremendous syntactic/semantic flexibility. Standards such as MISRA C++ 2008 and JSF AV++ help quickly define a language subset and best practices to provide a baseline for software coding standards used in specific projects.

Challenges with structural coverage and low-level testing

Structural coverage of destructors, instantiating complex data types for testing, testing templated classes and overloaded operators, and accessing private members, are just some of the challenges that arise when working with object-oriented technologies or related techniques. Tools need to be equipped to address these challenges and reduce cost of verification, while preserving the integrity/credibility of the verification activities.

Model-Based Development

As with object-oriented technologies, model-based development (MBD) is addressed within a supplement to DO-178C, called DO-331[16].

DO-331 takes the approach that specification models or design models take the place of high-level and low-level requirements respectively. Textual requirements may be linked to models upstream or downstream (Figure 17).

[16] RTCA DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A

Process that generates the life cycle data	MB Example 1	MB Example 2	MB Example 3 (See Note 1)	MB Example 4 (See Note 1)	MB Example 5 (See Note 1)
System Requirement and System Design Processes	**Requirements allocated to software**	**Requirements from which the Model is developed**	**Requirements from which the Model is developed**	**Requirements from which the Model is developed**	**Requirements from which the Model is developed**
Software Requirement and Software Design Processes	Requirements from which the Model is developed	Specification Model (See Note 2)	Specification Model	**Design Model**	**Design Model**
	Design Model	Design Model	Textual description (See Note 3)		
Software Coding Process	Source Code	Source Code	Source Code	Source Code	Source Code

Figure 17: Model Usage Examples[17]

Popular tools such as MathWorks® Simulink®[18], IBM® Rational® Rhapsody®[19], and ANSYS® SCADE[20] can generate code automatically. DO-331 MB.5.0 (Software Development Processes) addresses traceability, model standards and more for both software requirements and design processes where such tools are used. MB.5.3 (Software Coding Process) is merely a cross-reference the equivalent section in DO-178C, underlining the fact that best-practice coding-related process activities still apply whether code is hand-coded based on a set of textual requirements, hand-coded based on design models, or auto-generated from a tool.

Projects using auto-generated code almost always contain some hand-code too, and often include legacy hand-coded components. It is possible to apply different coding standards to these different code subsets , such as MISRA-C 2012 for hand-code, MISRA-C 2012 Appendix E for auto-generated code, and a custom coding standard for legacy code.

DO-331 MB 6.0 (Software verification process) expands on how best practice applies to MBD, with DO-331 MB.6.8.2 (Model Simulation for Verification of Executable Object Code) expanding upon which verification objectives can be partially satisfied at the model level, and which must be performed at the target level.

Partial Credit in the Model

"Verification of the Executable Object Code is primarily performed by testing. This can be partially assisted by a combination of model simulation and specific analysis ... This combination can be used to partially satisfy the following software testing objectives".

Those objectives include the compliance of EOC with high-level and low-level requirements, test coverage of software structure, and data coupling and control coupling. Additional verification activities must be performed on the target hardware to fully satisfy these objectives. The document goes on to say that when certification credit is sought from model simulation to partially satisfy software testing objectives and test coverage regarding high-level requirements then it must be ensured that the same design model is used for code generation and to produce the EOC.

It also specifies that plans are required to define which requirements and associated test and test coverage activities are to be satisfied at the model level, and which are to be exercised on the target.

[17] Based on table MB.1.1 from RTCA DO-331. Copyright © 2011 RTCA, Inc. All rights acknowledged.

[18] https://uk.mathworks.com/products/simulink.html

[19] http://www.-03.ibm.com/software/products/en/ratirhapfami

[20] http://www.ansys.com/products/embedded-software/ansys-scade-suite

Verification on Target Required

DO-331 MB.6.8.2 states that

"*... specific tests should still be performed in the target environment ... The following software testing and test coverage objectives cannot be satisfied by the model simulation since simulation cases should be based on the requirements from which the model is developed*."

These objectives listed include EOC robustness, its compliance with low-level requirements, and test coverage of low-level requirements.

The supplement then goes on to identify the various forms of verification objectives that can only be met on the target, including confirmation of compatibility with the target hardware, and hardware/software integration testing. It also lists various types of errors that can and cannot be revealed at the simulation level and can only be detected on the target hardware.

Finally, DO-331 MB.B.11 (FAQ #11) addresses the questions of model coverage activity:

"*Model coverage analysis is different than structural coverage analysis and therefore model coverage analysis does not eliminate the need to achieve the objectives of structural coverage analysis per DO-178C section 6.4.4.2*."

It goes on to state that model coverage analysis can be considered in very specific scenarios, "*...on a case-by-case basis and agreed upon by the certification authorities ...*"

As a result, most organizations do some of the verification activities within the model but then re-affirm the results of those activities on the target hardware to ensure that they meet the necessary criteria for meeting objectives.

The integration of test and modelling tools help to achieve that seamlessly, including the static analysis of generated code, the collection of code coverage from model execution, and the migration of model tests into an appropriate form for execution on the target hardware (Figure 17).

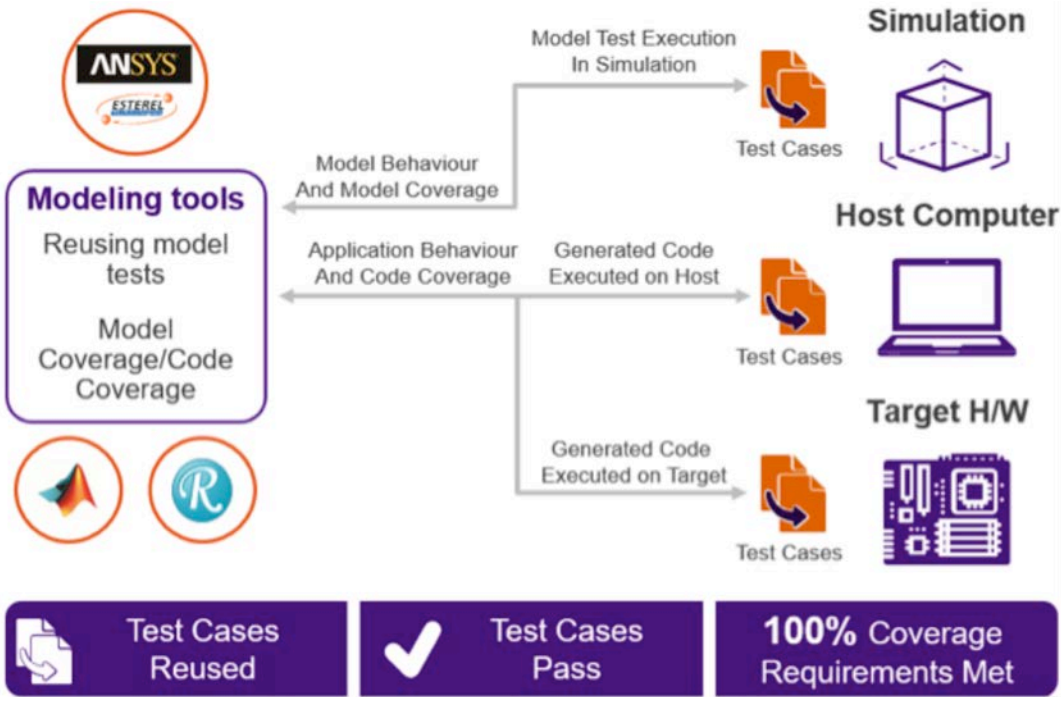


Figure 18: Migrating test cases from modelling tools to the LDRA tool suite for regression on target

Tool Qualification

If software tools are to automate significant numbers of DO-178C activities while producing evidential artefacts showing that objectives have been met, it is essential to ensure that those tools can be relied upon. DO-178C states that:

"*the purpose of the tool qualification process is to ensure that the tool provides confidence at least equivalent to that of the processes of this document are eliminated, reduced, or automated*."

Tool qualification is a vital part of to the certification process, and it is documented in the supplement Software Tool Qualification Considerations (DO-330)[21].

DO-330 introduces the concept of Tool Qualification level (TQL) on the basis of three criteria:

1) A tool whose output is part of the airborne software and thus could insert an error

2) A tool that automates verification processes and thus could fail to detect an error, and whose output is use to justify the elimination or reduction of:
   a) Verification processes other than that automated by the tool, or
   b) Development processes that could have an impact on the airborne software.

3) A tool that, within the scope of its intended use, could fail to detect an error.

Where a tool is designed to be used for verification purposes, its output is not used as part of the airborne software and it therefore cannot introduce errors into the software, making it a criteria 3 tool. Irrespective of the application DAL, such a tool is always assigned Tool Qualification Level 5 (Figure 19).

Software Level	Criteria		
	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Figure 19: Tool Qualification Level Matrix

Certification authorities such as the FAA, CAA, JAA, and ENAC undertake tool qualification on a project by project basis, so the responsibility for showing the suitability of any tools falls on to the organisation developing the application. However, they can make use of Tool Qualification Support Packages (TQSP) provided by the vendor. Such packages typically contain a series of documents, starting with the Tool Operation Requirements that identify the development process needs satisfied by the tool and including test cases to demonstrate that the tool is operating to specification in the verification environment.

Tool Qualification documentation must be referenced in other planning documents, and it plays a key role in the compliance process (Figure 20).

[21] RTCA DO-330 Software Tool Qualification Considerations Supplement to DO-178C and DO-278A

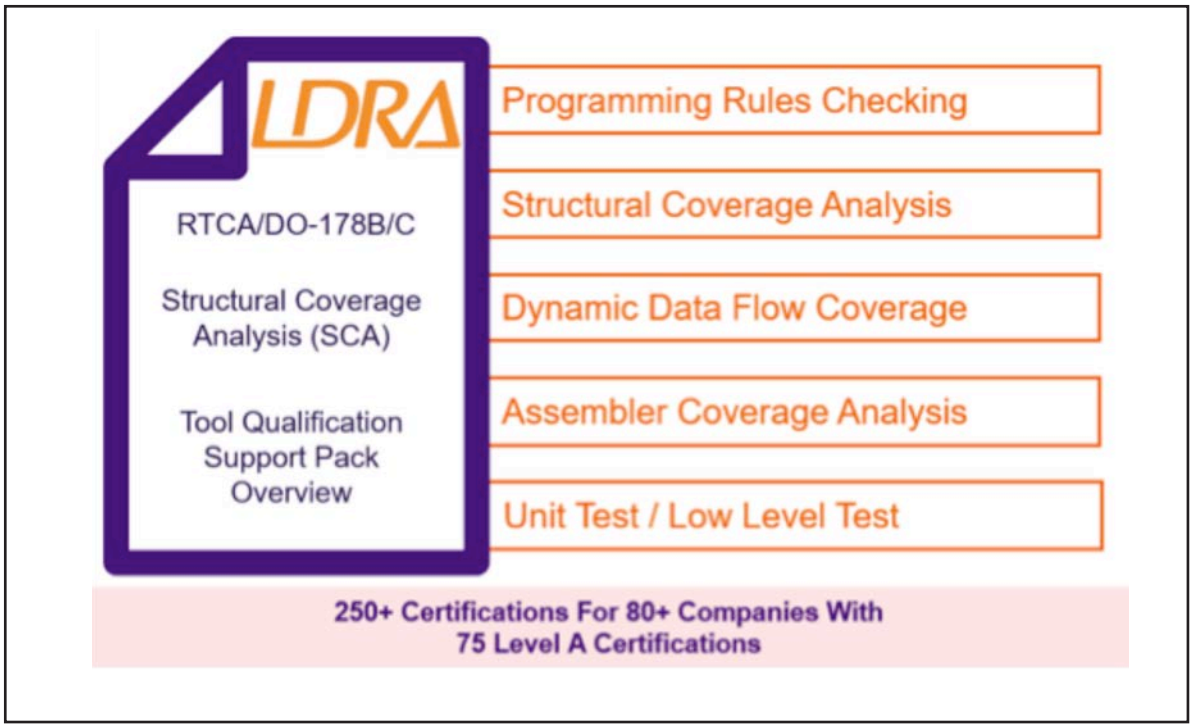


Figure 20: LDRA Tool Qualification Support Packages

Tool Selection

The use of traceability, test management and static/dynamic analysis tools for an airborne software project that meet the DO-178C certification requirements offers significant productivity and cost benefits. Tools make compliance checking easier, less error prone and more cost effective. In addition, they make the creation, management, maintenance and documentation of requirements traceability straightforward and cost effective. When selecting a tool to assist in achieving DO-178C acceptance the following criteria should be considered:

- Does the tool provide a complete 'end-to-end' traceability across the lifecycle through requirements, code, tests, artefacts, and objectives?
- Does the tool provide static analysis to ensure conformance to industry leading coding standards such as MISRA, CERT, and others?
- Does the tool enable Structural Coverage Analysis on the target hardware, as laid out in section 6.4.4.2 of the standard, including coverage at the source and object levels for Level A projects?
- Is the tool available for all the languages, platforms, tool chains, and targets required in the project?
- Has the tool been utilized in this manner successfully already?
- Will the tool vendor assist in tool qualification?
- Is tool support both flexible and extensive enough to meet changing requirements?
- Is the tool easy to use?

Works Cited

"Liskov's Substitution Principle",
OODesign.com, http://www.oodesign.com/liskov-s-substitution-principle.html

"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.

"EUROCAE DO-178B Software Considerations in Airborne Systems and Equipment Certification",
Prepared by EUROCAE Working Group 12 and RTCA Special Committee 167, December 10, 1992

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

"RTCA DO-330 Software Tool Qualification Considerations Techniques Supplement to DO-178C and DO-278A",
Prepared by SC-205, December 13, 2011

"RTCA DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A",
Prepared by SC-205, December 13, 2011

"RTCA DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A",
Prepared by SC-205, December 13, 2011

www.ldra.com

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

LDRA Technology Inc.
2540 King Arthur Blvd, 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

LDRA tool suite compliance and DO-178C v2.0 06/17