LDRA
Software Technology

Technical White Paper

Verification of Medical Device Software in Compliance with IEC 62304 1:2006

Working with the Medical Device Industry to Meet the Challenges of Achieving Cost-effective Certification

www.ldra.com

Contents

Background

Paraphrasing European Union directive 2007/47/EC of the European parliament of the council[1], a medical device can be defined as:

"Any instrument, apparatus, appliance, software, material or other article, whether used alone or in combination … to be used for human beings for the purpose of:

- Diagnosis, prevention, monitoring, treatment, or alleviation of disease
- Diagnosis, monitoring, treatment, alleviation of, or compensation for an injury or handicap
- Investigation, replacement, or modification of the anatomy or of a physiological process
- Control of conception"

FDA's Centre for Devices and Radiological Health (CDRH) is responsible for regulating firms who manufacture, repackage, relabel, and/or import medical devices sold in the United States. The FDA's introduction to its rules for medical device regulation states[2]:

"Medical devices are classified into Class I, II, and III. Regulatory control increases from Class I to Class III. The device classification regulation defines the regulatory requirements for a general device type. Most Class I devices are exempt from Premarket Notification 510(k); most Class II devices require Premarket Notification 510(k); and most Class III devices require Premarket Approval".

Given that such definitions encompass a large majority of medical products other than drugs, it is small wonder that medical device software now permeates a huge range of diagnostic and delivery systems. The reliability of the embedded software used in these devices and the risk associated with it has been an ever-increasing concern as that software becomes ever more prevalent.

In initial response to that concern, the functional safety standard IEC 62304[3] "Medical device software – Software life cycle processes" emerged in 2006 as an internationally recognized mechanism for the demonstration of compliance with the relevant local legal requirements. The set of processes, activities, and tasks described in this standard established a common framework for medical device software life cycle processes as shown in Figure 1.

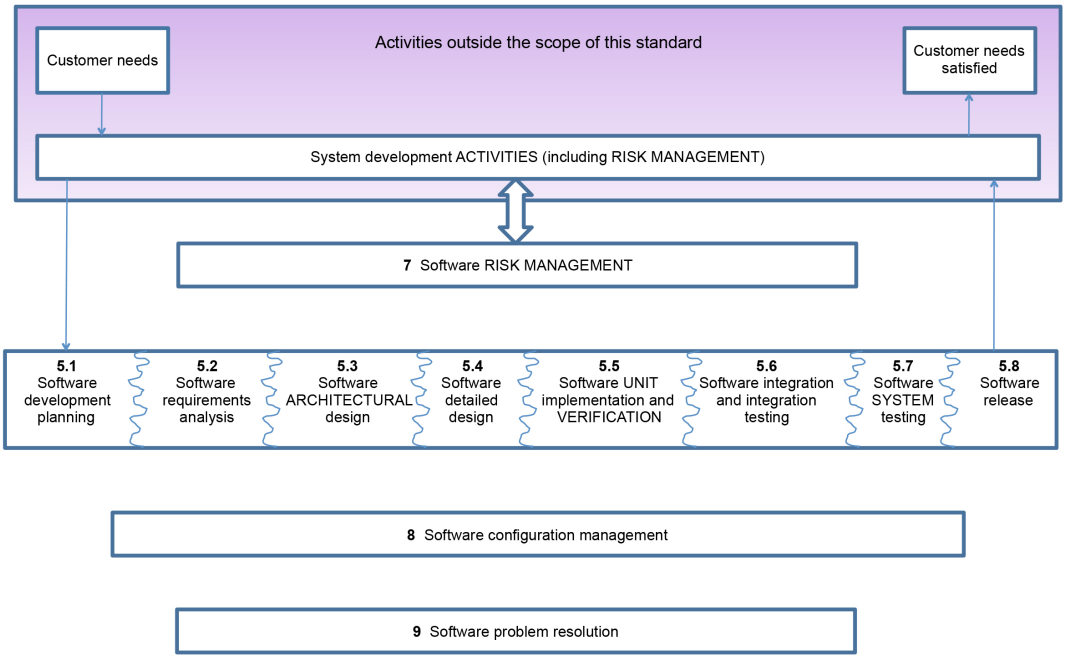

Figure 1: Overview of software development processes and activities according to IEC 62304:2006[4]

[1] "Directive 2007/47/ec of the European parliament and of the council". *Eur-lex Europa. 5 September 2007.*

[2] https://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/

[3] IEC 62304 International Standard Medical device software – Software life cycle processes Edition 1 2006-05

[4] IEC 62304:2006 - Medical device software - Software life cycle processes Figure 1 – Overview of software development PROCESSES and ACTIVITIES

In practice, for all but the most trivial applications compliance with IEC 62304 can only be demonstrated efficiently with a comprehensive suite of automated tools. This white paper describes the key software development and verification processes of the standard and to show how automation minimizes the cost of development and verification, and goes on to provide a sound foundation for an effective maintenance system once the product is in the field.

On June 15, 2015, the International Electrotechnical Commission, IEC, published Amendment 1:2015 to the IEC 62304 standard[5] and work on a second, updated edition of IEC 62304 is ongoing. The 2nd edition will possibly be published in 2018.

This document references only the first edition.

Classification

The standard defines three software safety classes (A, B and C) "... according to the possible effects on the patient, operator or other people". The classes are defined as follows:

Class A
No injury or damage to health is possible

Class B
Non SERIOUS INJURY is possible

Class C
Death or SERIOUS INJURY is possible

Partitioning of software items

The classification assigned to any medical device software has a tremendous impact on the code development process from planning, developing, testing, and verification through to release and beyond. It is therefore in the interests of medical device manufacturers to invest the effort to get it right the first time, minimizing unnecessary overhead by resisting over classification, but also avoiding expensive and time-consuming rework resulting from under classification.

IEC 62304:2006 helps to minimize development overhead by permitting software items to be segregated. In doing so, it requires that *"The software ARCHITECTURE should promote segregation of software items that are required for safe operation and should describe the methods used to ensure effective segregation of those SOFTWARE ITEMS"*

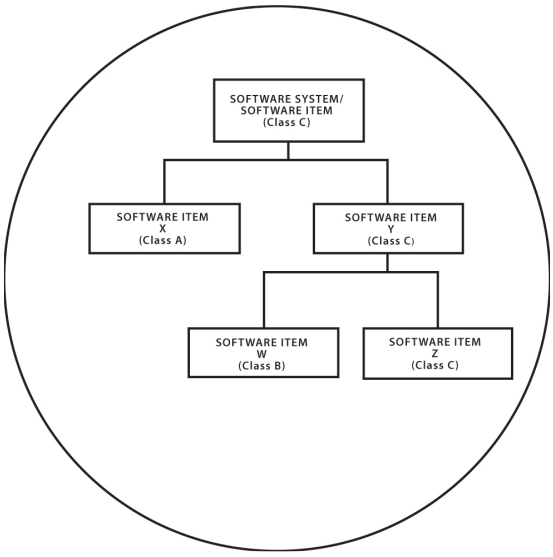

Figure 2: Example of partitioning of software items according to IEC 62304:2006 Figure B.1[6]

[5] IEC 62304:2006 - Medical device software - Software life cycle processes consolidated version Edition 1-1 2015-06

[6] IEC 62304:2006 - Medical device software - Software life cycle processes Figure B.1 – Example of partitioning of SOFTWARE ITEMS

Figure 2 shows the example used in the standard. In it, a software system has been designated Class C. That system can be segregated into one software item to deal with functionality of limited safety implications (software item X), and another to handle highly safety critical aspects of the system (software item Y).

That principle can be repeated in a hierarchical manner, such that software item Y can itself be segregated into software items W and Z, and so on – always on the basis that no segregated software item can negatively affect another. At the bottom of the hierarchy, software items such as X, W and Z that are divided no further are defined as software units.

Clause 5. Software Development PROCESS

Classification in practice

In practice, any company developing medical device software will carry out verification, integration and system testing on all software regardless of the safety classification, but the depth to which each of those activities is performed varies considerably. Table 1 is based on table A1 of the standard, and gives an overview of what is involved.

For example, subclass 5.4.2 of the standard states that *"The MANUFACTURER shall document a design with enough detail to allow correct implementation of each SOFTWARE UNIT"*.

Reference to Figure 3 shows that it applies only to Class C code.

Software Development PROCESS requirements by software safety CLASS		Class A	Class B	Class C
Clause	Sub-clauses			
5.1 Software development planning	5.1.1, 5.1.2, 5.1.3, 5.1.6, 5.1.7, 5.1.8, 5.1.9	X	X	X
	5.1.5, 5.1.10, 5.1.11		X	X
	5.1.4			X
5.2 Software requirements analysis	5.2.1, 5.2.2, 5.2.4, 5.2.5, 5.2.6	X	X	X
	5.2.3		X	X
5.3 Software ARCHITECTURAL design	5.3.1, 5.3.2, 5.3.3, 5.3.4, 5.3.6		X	X
	5.3.5			X
5.4 Software detailed design	5.4.1		X	X
	5.4.2, 5.4.3, 5.4.4			X
5.5 SOFTWARE UNIT implementation and verification	5.5.1	X	X	X
	5.5.2, 5.5.3, 5.5.5		X	X
	5.5.4			X
5.6 Software integration and integration testing	All requirements		X	X
5.7 SOFTWARE SYSTEM testing	All requirements		X	X
5.8 Software release	5.8.4	X	X	X
	5.8.1,,5.8.2,5.8.3,5.8.5,5.8.6		X	X
	5.8.7, 5.8.8			

Figure 3: Summary of which software safety classes are assigned to each requirement in the development lifecycle requirement, highlighting clause 5.4.2 as an example[7].

[7] Based on IEC 62304:2006 - Medical device software - Software life cycle processes Table A.1 – Summary of requirements by software safety class

IEC 62304 is essentially an amalgam of existing best practice in medical device software engineering, and the functional safety principles recommended by the more generic functional safety standard IEC 61508[8], which has been used as a basis for industry specific interpretations in a host of sectors as diverse as the rail industry, the process industries, and earth moving equipment manufacture.

A process-wide, proven tool suite has been shown to help ensure compliance to such software safety standards (in addition to security standards) by automating both the analysis of the code from a software quality perspective as well as the required validation and verification work. Equally important, the tool suite enables life-cycle transparency and traceability into and throughout the development and verification activities facilitating audits from both internal and external entities.

The V diagram in Figure 4 illustrates how the LDRA tool suite can help through the software development process described by IEC 62304. The tools also provide critical assistance through the software maintenance process (clause 6) and the risk management process (clause 7). Clause 5 of IEC 62304 details the software development process through eight stages ending in release. Notice that the elements of Clause 5 map to those in Figure 1 and Figure 3.

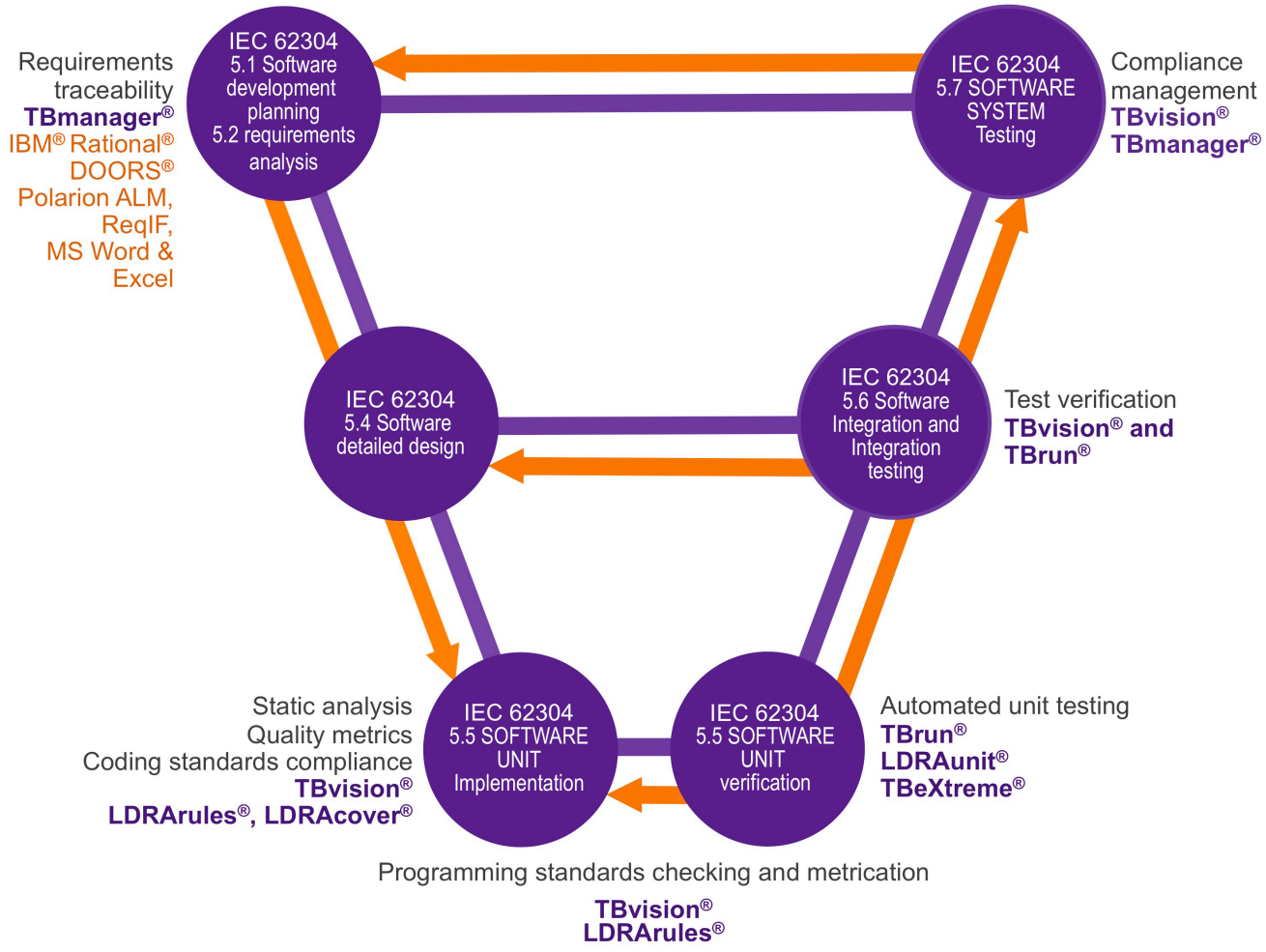

Figure 4: Mapping the capabilities of the LDRA tool suite to the guidelines of IEC 62304:2006

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

Sub-clause 5.1 Software Development Planning

The first objective in the software development process is to plan the tasks needed for development of the software in order to reduce risks as well as to communicate procedures and goals to members of the development team. This helps ensure that system quality requirements for the medical device software are understood and met by all team members and can be verified within the project.

Development planning applies to all classes of devices. It involves the creation of the plan with reference to the system design and the definition of system requirements, and the selection and definition of the development standards, methods, and tools to be used. There should be a defined procedure for integration and integration testing. A format for documentation should be specified along with plans for configuration management and verification.

Thorough software development planning is vital to success. The foundations for an efficient development cycle can be established by using tools that can facilitate structured requirements definition, such that those requirements can be confirmed as met by means of automated document (or "artefact") generation. IEC 62304 dictates that medical device software must include appropriate risk control measures in its system requirements.

The preparation of a mechanism to demonstrate that the requirements have been met will involve the development of detailed plans. A prominent example would be the software verification plan to include tasks to be performed during software verification and their assignment to specific resources.

Sub-clause 5.2 Software Requirements Analysis

Once each system requirement has been identified in such a way as to make it possible to demonstrate traceability right from the system requirement through to software system testing, and to show this risk control measures have been accounted for, the next step is to derive and document the software requirements from the system requirements.

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®[9], Siemens® Polarion® PLM®[10], or more generally, similar tools offering support for standard Requirements Interchange Formats[11]. 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.

Bi-directional traceability

Requirements traceability is widely accepted as a development best practice to ensure that all requirements are implemented and that all development artefacts can be traced back to one or more of them. With its constant emphasis on the need for the derivation of one development tier from the one above, IEC 62304:2006 is no exception to the countless functional safety standards that encourage it. For example:

- Sub-clause 5.1.1 section c: *"The [Software development] plan shall address the following … TRACEABILITY between SYSTEM requirements, software requirements, SOFTWARE SYSTEM test and RISK CONTROL measures implemented in software"*
- Sub-clause 5.1.6 section a: *"The MANUFACTURER shall include or reference in the software development plan the following information….DELIVERABLES requiring VERIFICATION"*
- Sub-clause 5.2.2 specifies in detail what the software requirements should include, and clause 5.2.4 calls for the system requirements to be re-evaluated as appropriate

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

[10] https://polarion.plm.automation.siemens.com/

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

- Sub-clause 5.3.1: *"The MANUFACTURER shall transform the requirements for the MEDICAL DEVICE SOFTWARE into a documented ARCHITECTURE"*
- Sub-clause 5.4.4 requires that the detailed design "...implements the software ARCHITECTURE; and ... is free from contradiction with the software ARCHITECTURE"
- Sub-clause 5.5.3 discusses acceptance criteria for software unit test, including *"Does the software code implement requirements including RISK CONTROL measures?"*
- Sub-clause 5.6.4 calls for integration testing to *"address whether the integrated SOFTWARE ITEM performs as intended"*, including *"the required functionality of the software"*

It would be easy to dismiss each of these disparate tasks as trivial, but their combined effect on project management overhead can be significant.

For instance, consider the sub-clause 5.6.4 from the above examples. The verification that integrated software items are performing as intended would demand an ability to demonstrate that the requirements of the detailed design have an implementation in the code and a test for correctness.

Then imagine 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[12] (Figure 5). When requirements are managed well, traceability can be established from the source requirement to its lower level requirements and from the lower level requirements back to their source. 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. Requirements traceability can also cover the relationships to other entities such as intermediate and final work products, changes in design documentation, and test plans.

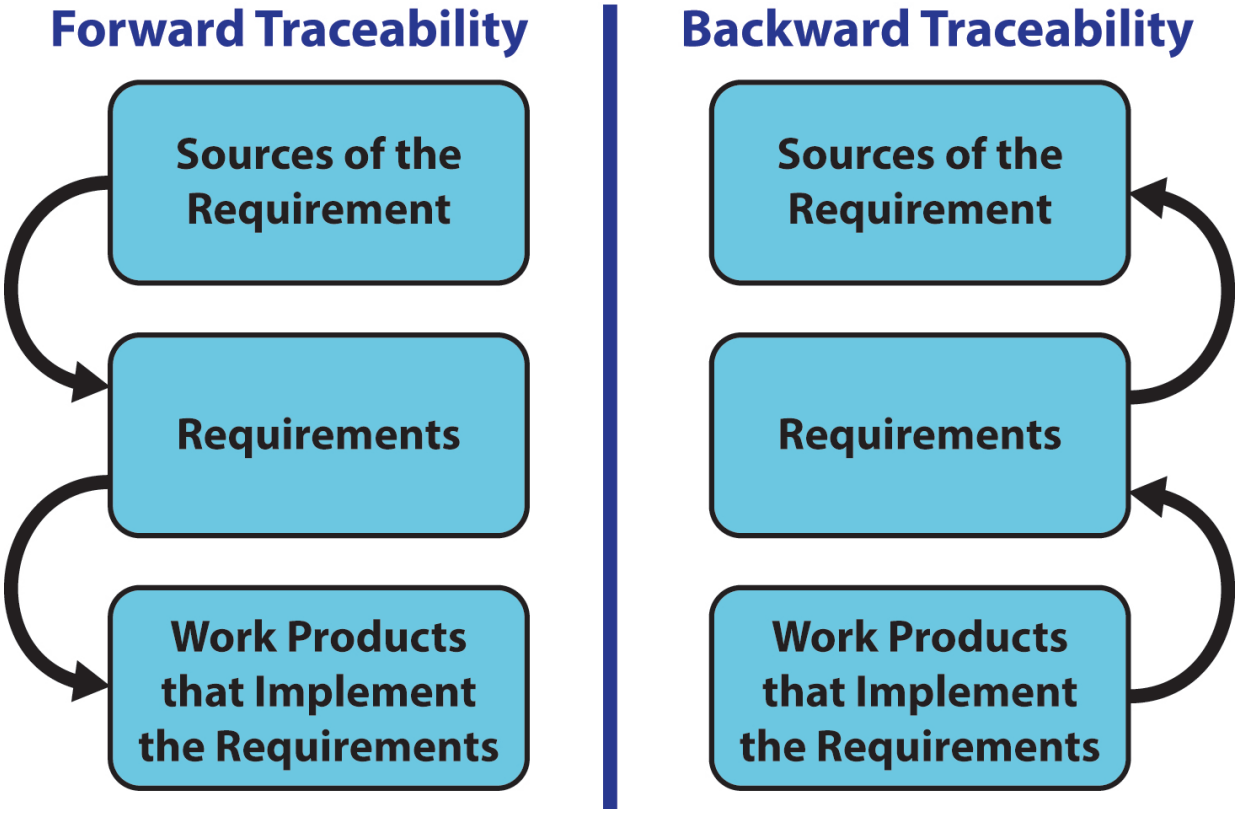

Figure 5: An Illustration of the principles of Bi-directional Traceability

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

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 headache into maintenance phase, requiring revision whenever a vulnerability is exposed.

A requirements traceability tool 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 tests required to be revisited can be dealt with accordingly (Figure 6).

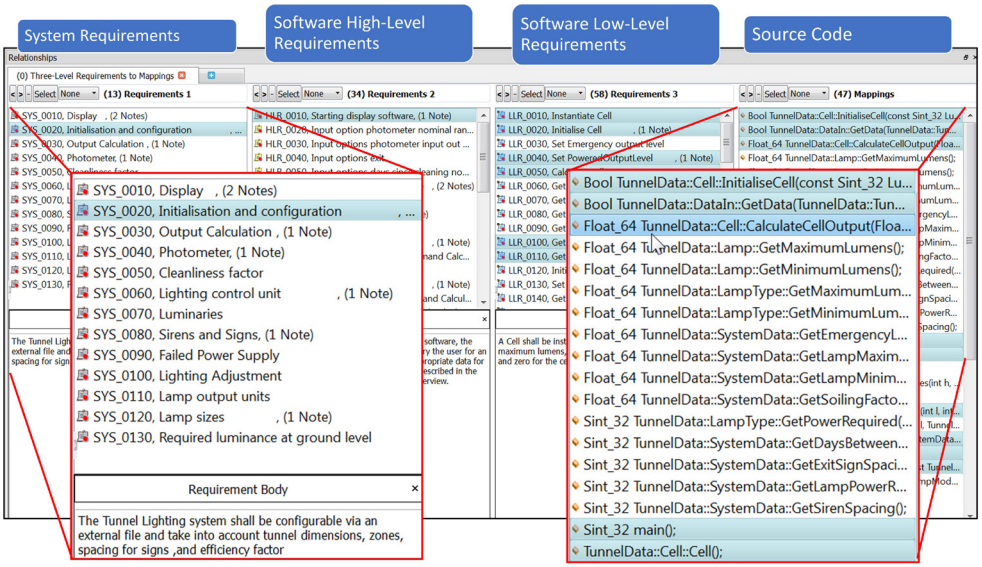

Figure 6: Automating requirements traceability with the TBmanager component of the LDRA tool suite

Sub-clause 5.3 Software Architectural Design

The software architectural design activity requires the manufacturer to define the major structural components of the software, their externally visible properties, and the relationships between them. Any software component behaviour that can affect other components should be described in the software architecture, such that all software requirements can be implemented by the specified software items. This is generally verified by technical evaluation.

Since the software architecture is based on the requirements, developing the architecture means defining the interfaces between the software items that will implement the requirements. Many of these software elements will be created by the project, but some may be brought in from other projects, possibly in the form of third-party libraries. Such code is an integral part of the system and is referenced in the standard as “SOUP”, an acronym for “Software of unknown provenance”. Repeated references to SOUP throughout the standard reflect the fact that its performance needs to be validated and verified.

Some architectural elements may need to be segregated for risk management purposes, such that their connections to the other elements then become part of the overall architecture. Tools can help in many elements of the architectural design process, including requirements specification, design model traceability and verification.

If a model-based approach is taken to software architectural design - for example, using MathWorks® Simulink®[13], IBM® Rational® Rhapsody®[14], or ANSYS® SCADE[15] then a tool suite that is integrated with the chosen modelling tools will make the analysis of generate code and traceability to the models far more seamless.

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

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

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

Sub-clause 5.4 Software Detailed Design

Once requirements and architecture are defined and verified, detailed design can be built on this foundation. Detailed design specifies algorithms, data representations, and interfaces between different software units and data structures. Because implementation depends on detailed design, it is necessary to verify the detailed design before the activity is complete, generally by means of a technical evaluation of the detailed design as a whole, and verification of each software unit and its interfaces.

Later in the development cycle, a tool suite can help by generating graphical artefacts suited to the review of the implemented design by means of walkthroughs or inspections. One approach is to prototype the software architecture in an appropriate programming language, which can also help to find any anomalies in the design. Graphical artefacts like call graph and flow graphs, are well suited for use in the review of the implemented design by visual inspection (Figure 7).

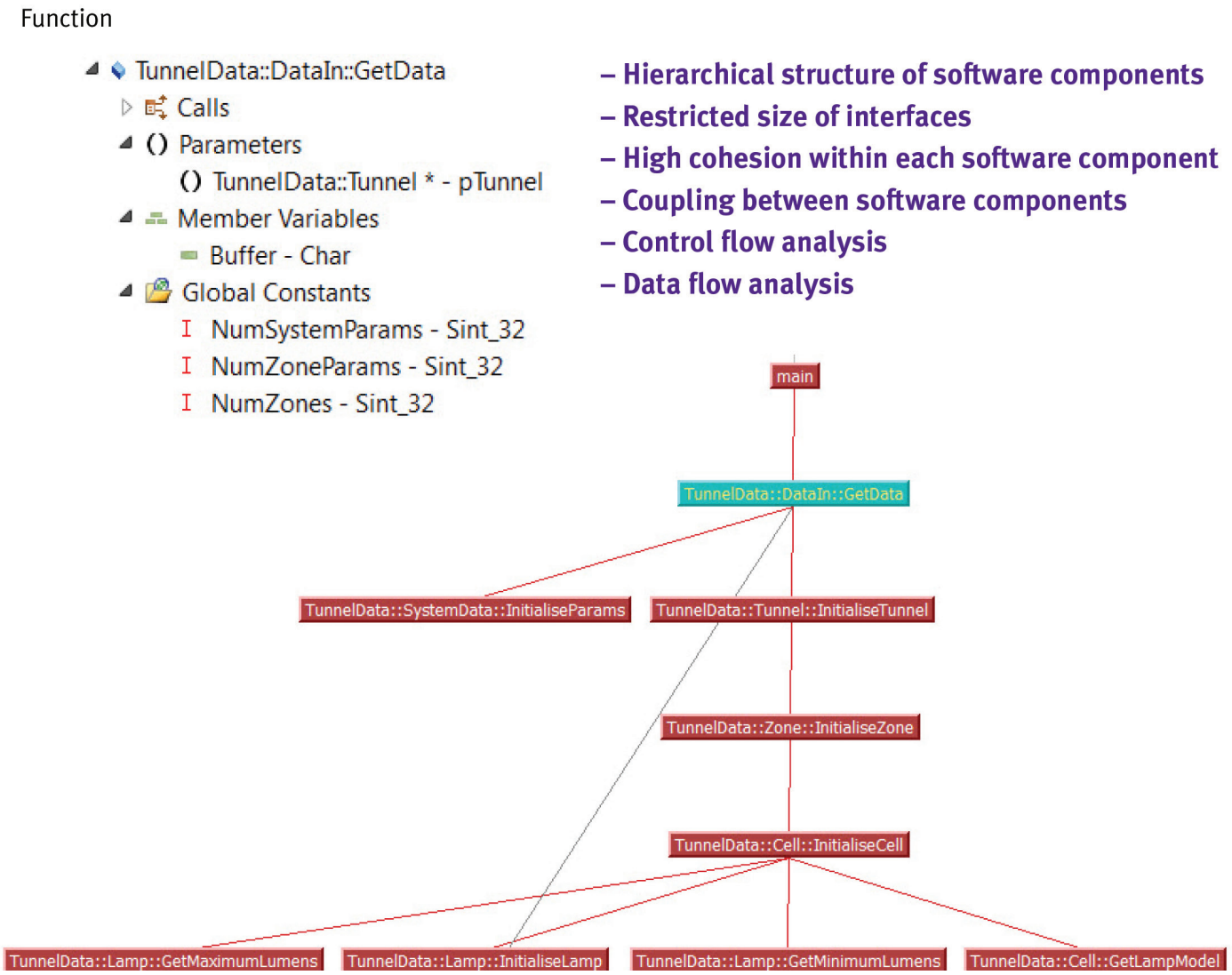

Figure 7: Diagrammatic representations of control and data flow generated from source code by the LDRA tool suite aid verification of software architectural and detailed design

Sub-clause 5.5 Software Unit Implementation and Verification

Next comes the process of translating the detailed design into source code.

To consistently achieve the desirable code characteristics, coding standards should be used to specify a preferred coding style, aid understandability, apply language usage rules or restrictions, and manage complexity. The code for each unit should be verified using a static analysis tool to ensure that it complies in a timely and cost-effective manner.

Verification tools such as the TBvision component of the LDRA tool suite largely offer support for a range of coding standards such as MISRA C and C++, JSF++ AV, HIS, CERT C, and CWE. The better tools will be able to confirm adherence to a very high percentage of the rules dictated by each standard, whereas more lightweight tools will be unable to detect the more subtle violations. More comprehensive solutions will also support the creation of and adherence to in-house standards from both user-defined and industry standard rule sets.

IEC 62304 also states that the manufacturer shall establish strategies, methods, and procedures for verifying each software unit. Where verification is done by testing, the test procedures shall be evaluated for correctness. Amongst the acceptance criteria are considerations such as the verification of the proper event sequence, data and control flow, fault handling, memory management and initialization of variables, memory overflow detection and checking of all software boundary conditions.

Unit test tools such as the TBrun component of the LDRA tool suite often provide a graphical user interface for unit test specification which is used to create tests according to the defined specification and to present a list of all defined test cases with appropriate pass/fail status. The ability to automatically present the graphical presentation of control flow graphs, and to create test harnesses, stub functions, and cover for missing member or global variables means that unit test execution and interpretation becomes a much quicker and easier process, requiring a minimum of specialist knowledge. By extending the process to the automatic generation of test vectors, the tools provide a straightforward means to analyse boundary values without creating each test case manually. Test sequences and test cases are retained so that they can be repeated (“regression tested”), and the results compared with those generated when they were first created.

Thorough verification also requires static and dynamic data and control flow analysis. Static data flow analysis produces a cross reference table of variables, which documents their type, and where they are utilized within the source file(s) or system under test. It also provides details of data flow anomalies, procedure interface analysis and data flow standards violations.

Dynamic data flow analysis builds on that accumulated knowledge, mapping coverage information onto each variable entry in the table for current and combined datasets and populating flow graphs to illustrate the control flow of the unit under test.

Sub-clause 5.6 Software Integration and Integration Testing

Software integration testing focuses on the transfer of data and control across a software module’s internal interfaces and external interfaces such as those associated with medical device hardware, operating systems, and third party software applications and libraries. This activity requires the manufacturer to plan and execute integration of software units into ever larger aggregated software items, ultimately verifying that the resulting integrated system behaves as intended.

Integration testing can also be used to demonstrate program behaviour at the boundaries of its input and output domains and confirms program responses to invalid, unexpected, and special inputs. The program’s actions are revealed when given combinations of inputs or unexpected sequences of inputs are received, or when defined timing requirements are violated. The test requirements in the plan should include, as appropriate, the types of white box testing and black box testing to be performed as part of integration testing.

Testing tools need to have the capability to provide dynamic structural coverage analysis, both at system test level and at unit test level, which is a mechanism to show which parts of the code base have been exercised during testing. The coverage data derived from unit and system test can be combined to provide the most effective way of working for the particular needs of a development project. A common approach is to operate them in tandem, so that (for instance) coverage can be generated for most of the source code through a dynamic system test, and complemented using unit tests to exercise defensive code and other aspects.

Testing of changed source code can again be performed using regression testing. It is advisable to validate test cases as a matter of course and perhaps automatically, to ensure that any changed code has not affected proven functionality elsewhere. The requirements for structural coverage metrics like statement, branch, condition, procedure/function call, and data flow coverage varies depending on device classification, and all are provided by both the unit test and system test facilities within tools such as the LDRA tool suite.

Sub-clause 5.7 Software System Testing

Testing at the system level requires the manufacturer to verify the software's functionality by verifying that the requirements for the software have been successfully implemented. Software system testing demonstrates that the specified functionality exists in the system as it will be deployed, and that performance of the program is as specified.

Software system testing is in many ways the ultimate integration test, and comments relating to integration testing still apply here. Functional (black box) testing with no code coverage can also be performed although it might be desirable to use white box methods to more efficiently accomplish certain tests, initiate stress conditions or faults, or increase code coverage of the qualification tests.

Software system testing tests the integrated software and can be performed either in a simulated environment, on actual target hardware, or on the implemented medical device.

The Connected System – A New Significance for System Maintenance

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, 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 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.

Clause 6. Software Maintenance PROCESS

Medical devices require maintenance when out in the field, and that also involves risks which must be tracked, managed and mitigated in accordance with the processes and procedures laid out in the standard (Figure 8).

A high percentage of medical device software defects have always been introduced after product release, and many are related to the application of inappropriate software updates and upgrades. For that reason, the software maintenance process is considered to be as important as the software development process. Clause 6 of the IEC 62304 standard spells out details for the software maintenance process, which has much in common with the software development process.

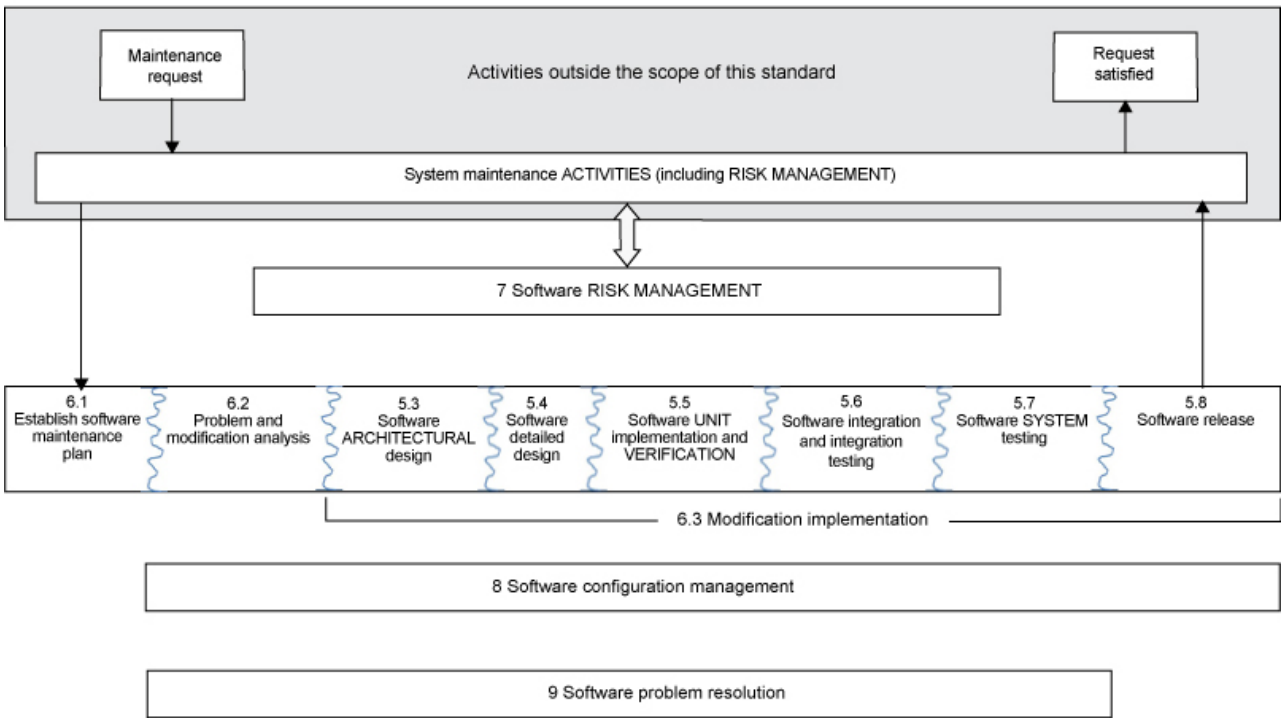

Figure 8: Overview of software maintenance processes and activities

Sub-clause 6.3.1 states that the manufacturer shall use the software development process or an established maintenance process to implement the modifications. Either way, it is important to manage changes in accordance with sub-clause 7.4.1 which requires that changes be analysed to prevent introducing any harmful causes which could contribute to a hazardous situation. Furthermore, those changes are to be assessed to determine whether additional software risk control measures are required.

Given the similarity of the required processes before and after release, and assuming successful deployment of high assurance software in the first place, it makes sense to continue to apply the same tool chain as was used in development.

Establishing a software maintenance plan requires the manufacturer to create or identify procedures for implementing maintenance activities and tasks. To implement corrective actions, control changes during maintenance, and manage the release of revised software, the manufacturer should document and resolve reported problems and requests from users, as well as manage upgrades and associated modifications to the functionality of the medical device software.

The impact of each modification can therefore vary enormously, from a minor correction to the migration of an application to a new environment or platform. In each case, the objective is to modify the released medical device software while preserving its integrity. Where the changes require new development work, they should be managed in accordance with clauses from the body of the standard as well as those specific to maintenance.

Just like the requirements specified in the development of the device, 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®[16], Siemens® Polarion® PLM®[17], or more generally, similar tools offering support for standard Requirements Interchange Formats[18]. Smaller projects can cope admirably with carefully worded Microsoft® Word® or Microsoft® Excel® documents, written to facilitate links between phases the development lifecycle, whatever model is applied.

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

[17] https://polarion.plm.automation.siemens.com/

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

Identifying necessary retest

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 such as LDRA TBmanager® helps alleviate this concern by automatically maintaining the connections between the requirements, development, and testing artefacts and activities. In the example shown in Figure 9, a change is proposed to the System Level 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.

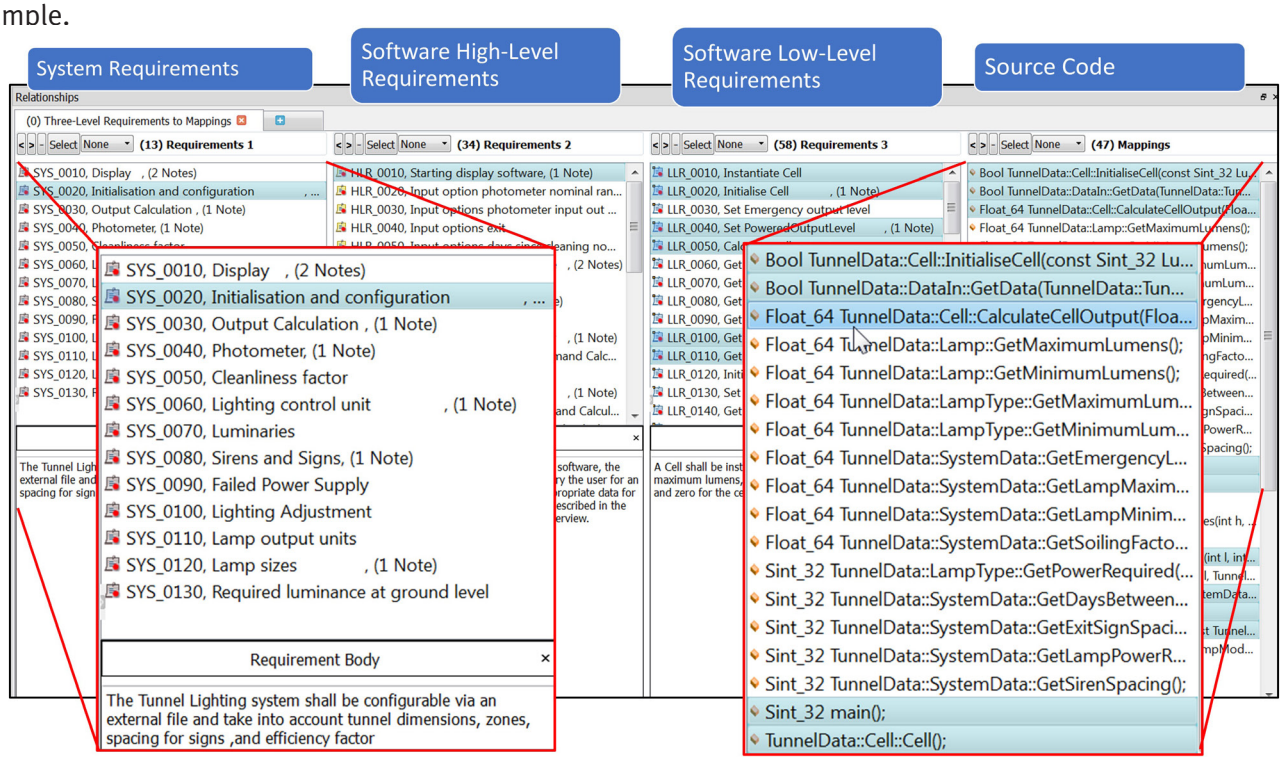

Figure 9: Identifying the impact of requirements change with the TBmanager component of the LDRA tool suite

The existing code as launched should also have undergone quality control measures such as static analysis to assess whether coding standards have been met, unit tests to confirm functionality of each code module, and dynamic coverage analysis to show that all parts of the code have been exercised.

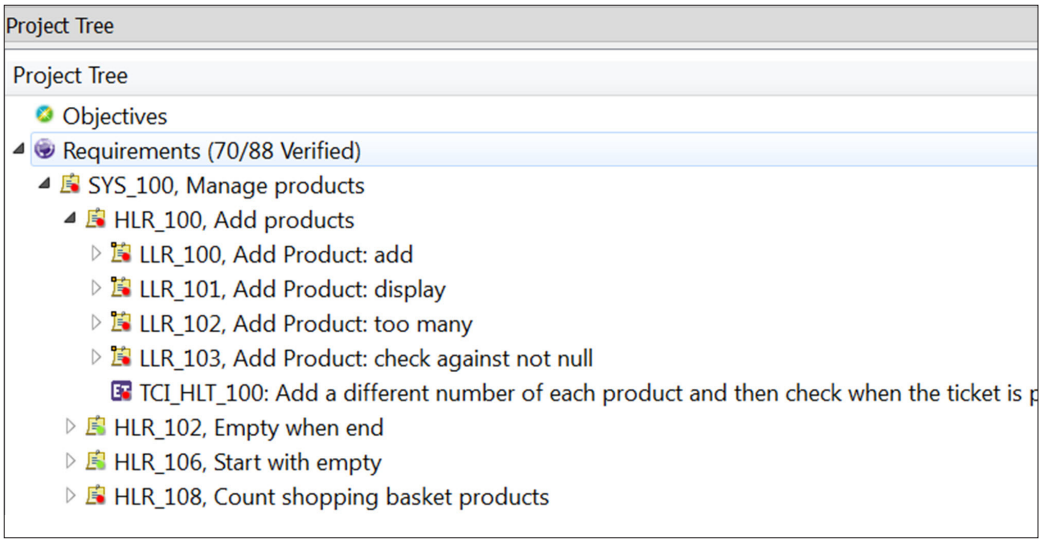

Figure 10: Showing functions requiring retest with the TBmanager component of the LDRA tool suite

Figure 10 shows a display from the TBmanager requirements traceability component of the LDRA tool suite. In this example, a system 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.

In the example, there is a test case file (tcf) associated with each of four low-level requirements – “add”, “display”, “too many” and “check against not null”. Those files retain the test vectors associated each of these low level requirements, meaning that they can all be re-run at the touch of a button and the code’s functionality confirmed.

Of course, some tests will require change to reflect the new functionality dictated by the updated requirement, but in many cases it is enough to confirm that things work as they did previously. Automating the process in this way means that doing so requires minimal effort.

Development process artefacts such as coding standards compliance reports and code coverage reports are also retained and associated with requirements, so that as the tests are re-run (“regression tested”) the artefacts are re-generated to go with them (Figure 11).

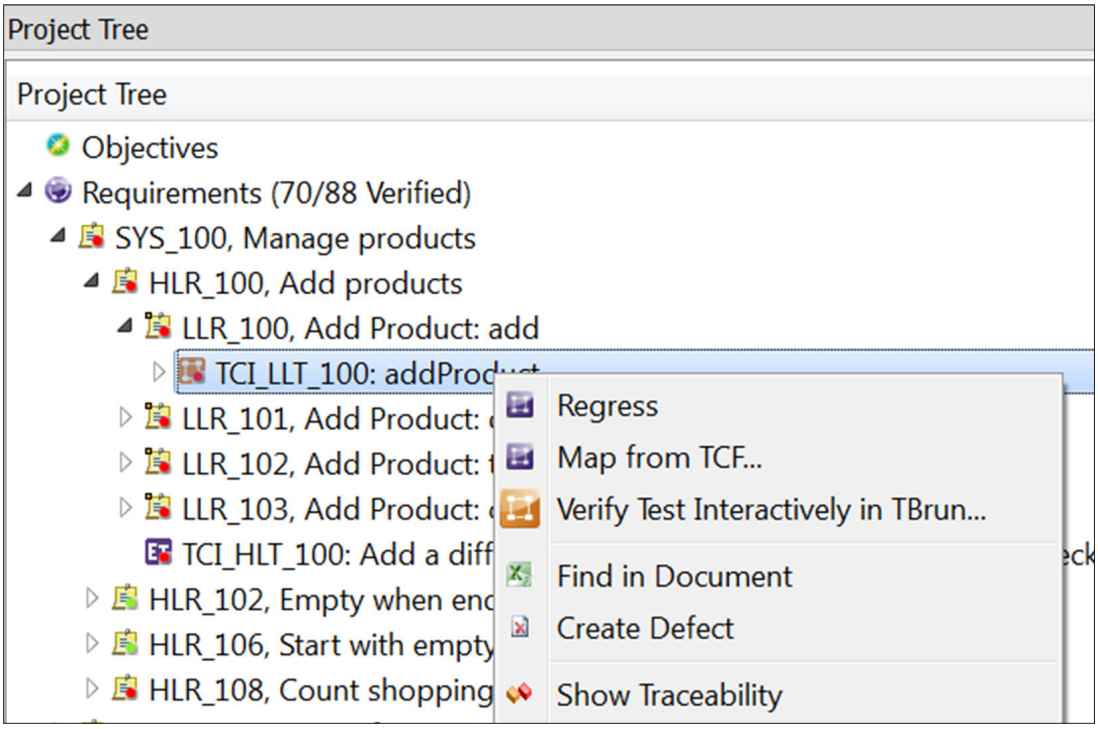

Figure 11: “Regression Testing” – Re-running unit tests to show that the functionality they describe still holds true, using TBmanager and TBrun components of the LDRA tool suite

Conclusion

A software functional safety standard such as that prescribed by IEC 62304 with its many sections, clauses and sub-clauses may at first seem intimidating. However, once broken down into digestible pieces, its guiding 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 and quality—and above all the safety and effectiveness of medical devices. When used with 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.

Works Cited

"Directive 2007/47/ec of the European parliament and of the council". Eur-lex Europa. 5 September 2007.

US Food and Drug Administration website https://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/

IEC 62304 International Standard Medical device software – Software life cycle processes Edition 1 2006-05

IEC 62304 International Standard Medical device software – Software life cycle processes Consolidated Version Edition 1.1 2015-06

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

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

Siemens Polarion ALM website https://polarion.plm.automation.siemens.com/

Object Management Group Requirements Interchange Format website http://www.omg.org/spec/ReqIF/

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

MathWorks SIMULINK website https://uk.mathworks.com/products/simulink.html

IBM Rational Rhapsody family website http://www-03.ibm.com/software/products/en/ratirhapfami

ANSYS SCADE Suite website http://www.ansys.com/products/embedded-software/ansys-scade-suite

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 and IEC 62304:2006 v3.0 04/18