Developing Medical Device Software to be compliant with IEC 62304-Amendment 1:2015

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* [disability]
- *Investigation, replacement, or modification of the anatomy or of a physiological process*
- *Control of conception*"

FDA's Center 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[4]. 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.

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

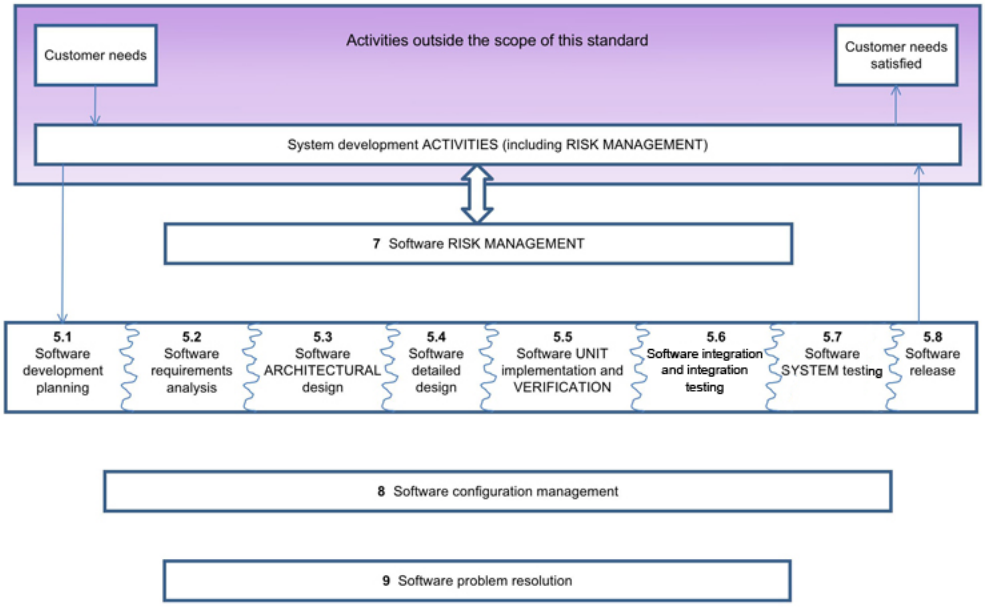


Figure 1: Overview of software development processes and activities according to IEC 62304:2006 +AMD1:2015[5]

On June 15, 2015, the International Electrotechnical Commission, IEC, published Amendment 1:2015 to the IEC 62304 standard "Medical device software – software life cycle processes"[6]. The amendment complements the 1st edition from 2006 by adding and amending various requirements, including those relating to safety classification, the handling of legacy software, and software item separation.

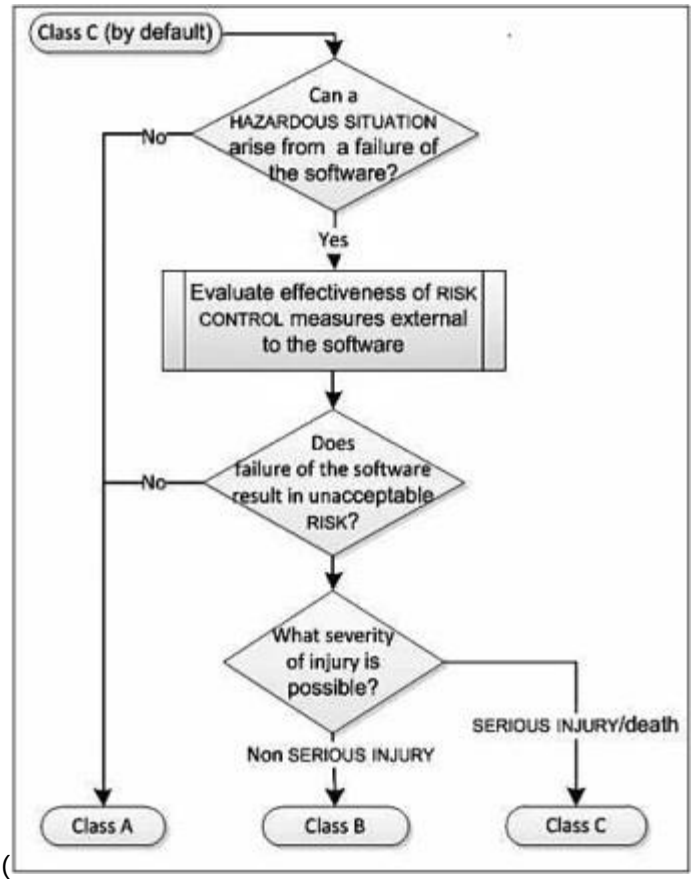
In practice, for all but the most trivial applications compliance with IEC 62304 can only be demonstrated efficiently with a comprehensive suite of automated tools. This paper describes the key software development and verification processes of the standard, and shows how automation both minimizes the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

Work on the second, updated edition of IEC 62304 is ongoing. The 2nd edition will possibly be published in 2018. It seems very likely that the changed requirements included in Amendment 1:2015 will be integrated into the updated edition.

Classification

One of the more significant changes concerns the new risk-based approach to the safety classification of medical device software. The previous concept was based exclusively on the severity of the resulting harm. Downgrading of the safety classification of medical device software from C to B or B to A used to be possible by adopting hardware-based risk mitigation measures external to the software. The new amendment now replaces this concept with safety classification as shown in a

[5] IEC 62304:2006/AMD1:2015 Amendment 1 - Medical device software - Software life cycle processes Figure 1 – Overview of software development PROCESSES and ACTIVITIES
[6] IEC 62304:2006/AMD1:2015 Amendment 1 - Medical device software - Software life cycle processes



decision tree (*Figure 2*).

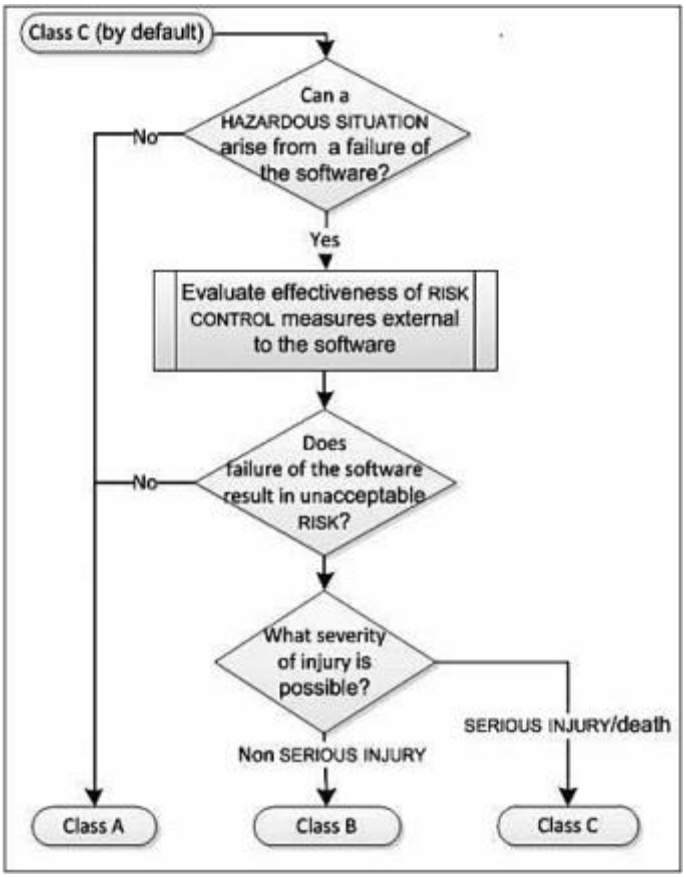


Figure 2: Safety classification according to IEC 62304:2006 +AMD1:2015[7]

The three classes are defined in the standard as follows:

Class A

The software system cannot contribute to a hazardous situation, or the software system can contribute to a hazardous situation which does not result in unacceptable risk after consideration of risk control measures external to the software system.

Class B

The software system can contribute to a hazardous situation which results in unacceptable risk after consideration of risk control measures external to the software system, but the resulting possible harm is non-serious injury.

Class C

The software system can contribute to a hazardous situation which results in unacceptable risk after consideration of risk control measures external to the software system, and the resulting possible harm is death or serious injury.

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

[7] IEC 62304:2006/AMD1:2015 Amendment 1 - Medical device software - Software life cycle processes Figure 3 – Assigning software safety classification

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 +AMD1:2015 helps to minimise 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*"

Amendment 1 clarifies the position on that software segregation by stating that segregation is not restricted to physical separation, but instead permits "*any mechanism that prevents one SOFTWARE ITEM from negatively affecting another*" suggesting that separation in software is similarly valid.

Figure 3: Example of partitioning of software items according to IEC 62304:2006 +AMD1:2015 Figure B.1[8]

Figure 3 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

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

[8] IEC 62304:2006/AMD1:2015 Amendment 1 - Medical device software - Software life cycle processes Figure B.1 – Example of partitioning of SOFTWARE ITEMS

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 4 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, 5.1.11,5.1.12		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	X
5.8 Software release	5.8.1,,5.8.2,5.8.4,5.8.7,5.8.8	X	X	X
	5.8.3, 5.8.5, 5.8.6,		X	X

Figure 4: Summary of which software safety classes are assigned to each requirement in the development lifecycle requirement, highlighting clause 5.4.2 as an example[9].

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[10], 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 and the required validation and verification work. Equally important, such a 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 5 illustrates how tools 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 5.

[9] Based on IEC 62304:2006/AMD1:2015 Amendment 1 - Medical device software - Software life cycle processes Table A.1 – Summary of requirements by software safety class

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

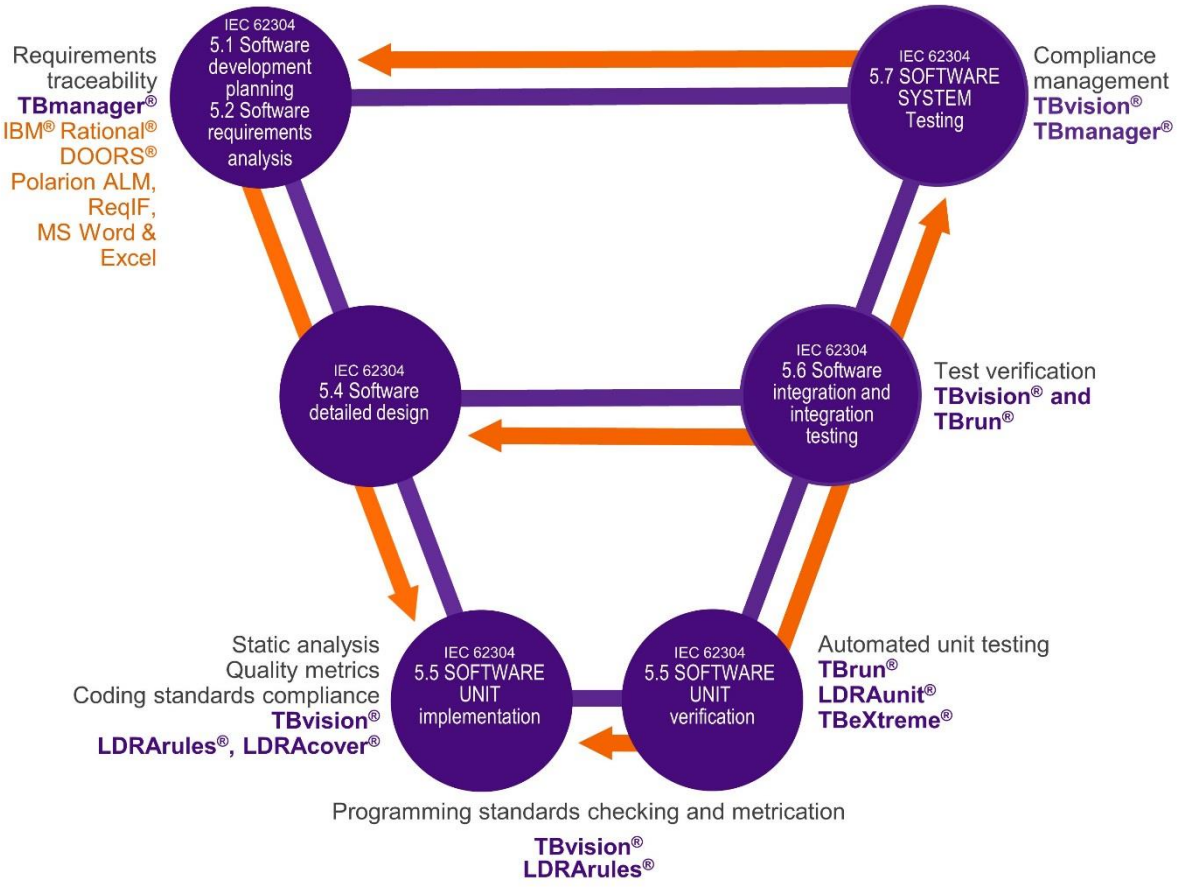


Figure 5: Mapping the capabilities of the LDRA tool suite to the guidelines of IEC 62304:2006 +AMD1:2015

Sub-clause 5.1 Software Development Planning outlines the first objective in the software development process, which is to plan the tasks needed for development of the software in order to reduce risks and communicate procedures and goals to members of the development team.

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.

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.

Software Requirements Analysis (Sub-clause 5.2) involves deriving and documenting the software requirements based on 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®[11], or Siemens® Polarion® PLM®[12]. 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.

This Bidirectional Traceability of Requirements[13] (Figure 6) would be easily achieved in an ideal world. But most projects suffer from unexpected changes of requirement imposed by a customer. What is then impacted? Which requirements need re-writing? What elements of the code design? What code needs to be revised? And which parts of the software will require re-testing?

[11] http://www-03.ibm.com/software/products/en/ratidoor
[12] https://polarion.plm.automation.siemens.com/
[13] http://www.compaid.com/caiinternet/ezine/westfall-bidirectional.pdf Bidirectional Requirements Traceability, Linda Westfall

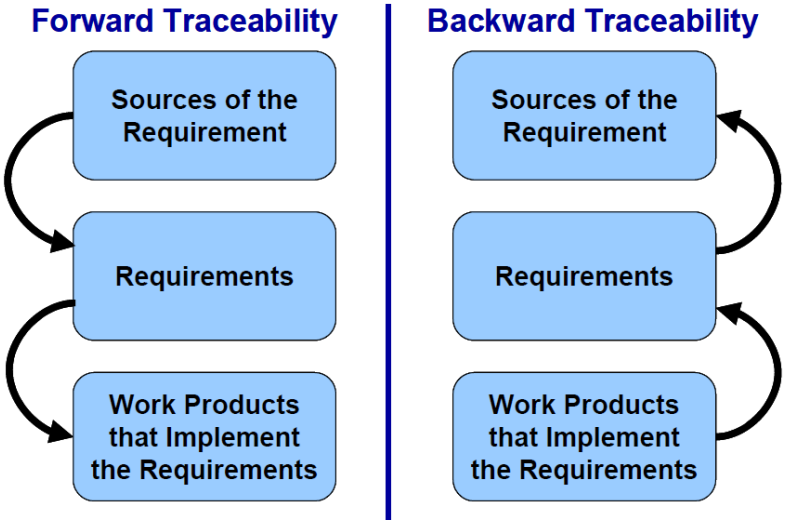


Figure 6: An Illustration of the principles of Bidirectional Traceability

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

A requirements traceability tool alleviates this concern by automatically maintaining the connections between 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 7).

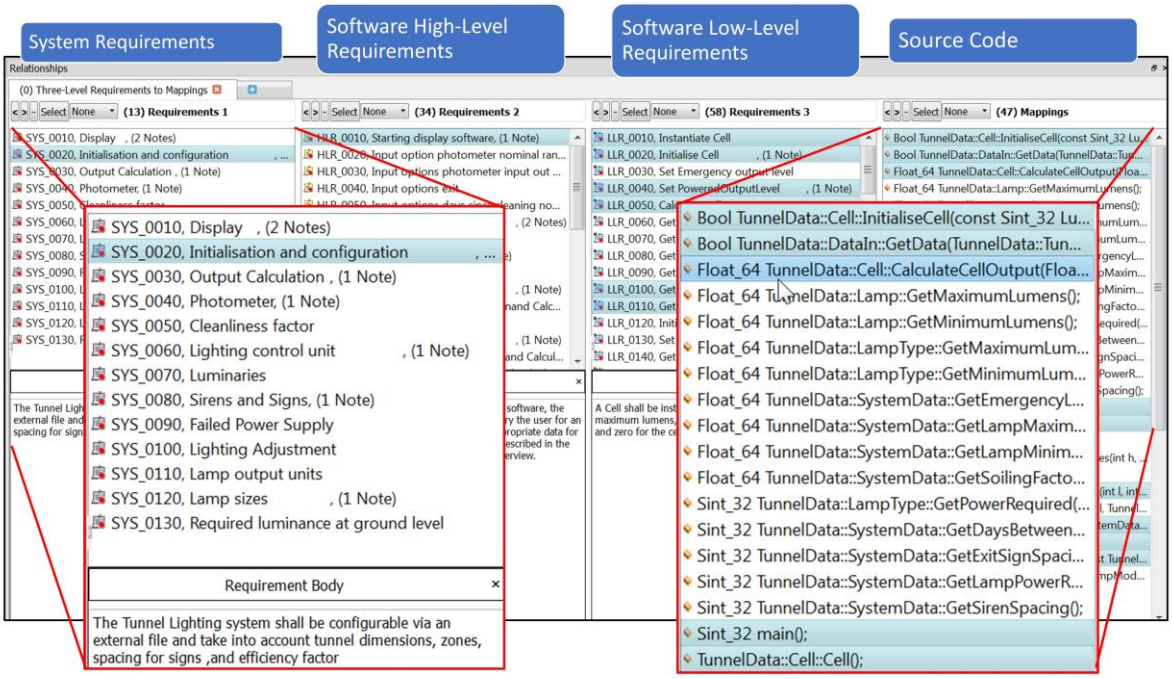


Figure 7: Automating requirements traceability with the TBmanager component of the LDRA tool suite

Software Architectural Design (Sub-clause 5.3) 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.

Developing the architecture means defining the interfaces between the software items that will implement the requirements. Any third-party software integration must be in accordance with **Sub-clause 4.4**, "**Legacy Software**".

If a model-based approach is taken to software architectural design using tools such as MathWorks® Simulink®[14], IBM® Rational® Rhapsody®[15], or ANSYS® SCADE[16], then their integration with test tools will make for seamless analysis of generated code and ensure traceability to the models.

Software Detailed Design (Sub-clause 5.4) involves the specification of algorithms, data representations, and interfaces between different software units and data structures to implement the verified requirements and architecture.

Later in the development cycle, tools 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 8).

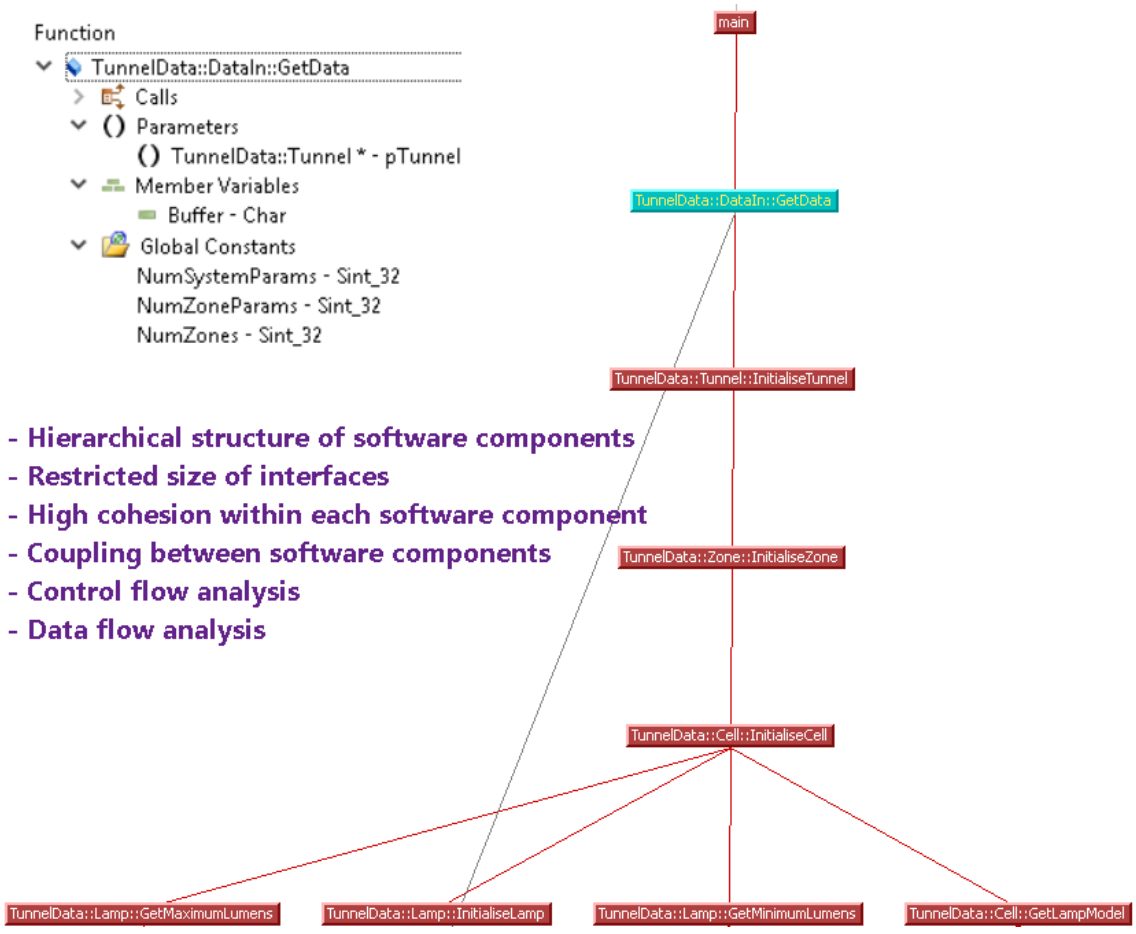


Figure 8: Diagrammatic representations of control and data flow generated from source code by the LDRA tool suite aid verification of software architectural and detailed design

Software Unit Implementation and Verification (Sub-clause 5.5) involves the translation of 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.

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

Verification tools 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, and will also support the creation of, and adherence to, in-house standards from both user-defined and industry standard rule sets.

IEC 62304 also requires strategies, methods, and procedures for verifying each software unit. 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 offer a graphical user interface for the specification of requirements-based tests and to present a list of all such defined test cases with appropriate pass/fail status. By extending the process to the automatic generation of test vectors, such 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.

Software Integration and Integration Testing (Sub-clause 5.6) 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.

To show which parts of the code base have been exercised during testing, the LDRA tool suite has the capability to perform dynamic structural coverage analysis, both at system test level and at unit test level. Mechanisms for structural coverage such as statement, branch, condition, procedure/function call, and data flow coverage vary in intensity, and so are specified by the standard depending on classification.

A common approach is to operate unit and system test 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 such as defensive code. It is advisable to re-run (or "regression test") these test cases as a matter of course and perhaps automatically, to ensure that any changed code has not affected proven functionality elsewhere.

Software System Testing (Sub-clause 5.7) requires the manufacturer to verify that the requirements for the software have been successfully implemented in the system as it will be deployed, and that the performance of the program is as specified.

Clause 6. Software Maintenance PROCESS

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.

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.

Automatically maintaining the connections between the requirements, development, and testing artefacts and activities helps alleviate this concern – not just during development, but onwards into deployment and the maintenance phase.

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.compaid.com/caiinternet/ezine/westfall-bidirectional.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