Using Dynamic Software Analysis to Support Medical Device Approval

Mark Pitchford, Field Application Engineer, LDRA
Chris Ault, Product Marketing Manager, Medical, QNX Software Systems
mark.pitchford@ldra.com, cault@qnx.com

"Beware of bugs in the above code; I have only proved it correct, not tried it".
—Donald Knuth

Introduction

Manufacturers of medical devices that include software face the same challenges as everyone building complex systems: time, quality, size (number and complexity of features), and cost. To these must be added approval by the FDA, MDD, MHRA, Health Canada and their counterparts in every jurisdiction where the devices will be used.

In this paper, we look at a) how dynamic code analysis can support demonstrations of compliance with safety requirements, and b) key capabilities we should look for in dynamic analysis tools. We then present in appendices, to help with tool selection, tables mapping development activities with requirements in the IEC 62304 standard: and, to help with OS selection, a short description of OS characteristics that can facilitate the design, development and approval of safety-related software.

Expertise and Process

Expertise and good development processes are not guarantees that a system will meet its required level of dependability, or even that the system will be a good one. However, they do vastly improve the chances that this will be the case.

Great expertise is needed to produce a design of the simplicity required for a safety-critical system. A comprehensive understanding of software validation methods, the software being evaluated, and the context in which it is evaluated (including validations of similar systems) is required to demonstrate that the software system in question meets its safety requirements.

It is no accident that IEC 62304 focuses on the development process. Considering this, we would do well, not only to develop our software in an environment that meets the most exacting quality management standards, but also to use tools that both help ensure we maintain these standards and provide evidence of this for auditors and regulatory agencies.

Demonstrating dependability

To ensure that their devices receive regulatory agency approvals, manufacturers must demonstrate that the devices meet their safety specifications. For the device software, this means demonstrating that the software meets required standards of dependability (reliability and availability). Whether reliability or availability is more important depends on how the system will be used. Carefully limited claims and precise dependability requirements provide a defined context and accurate measures in and against which we can validate a software system’s dependability.[1]

Defining acceptable risk

No software system is absolutely dependable, and even if a system were absolutely dependable, we would have no way of proving this. The techniques available to us can never prove that the system will never fail. They can only help us find faults and eliminate them, and estimate the probability of failure. A software system is thus deemed “safe” when its probability of failure is sufficiently low as not to present an unacceptable risk. Precisely what “unacceptable risk” or, rather “acceptable risk” means differs between industries and jurisdictions. Methods include:

ALARP (As Low as Reasonably Practical): the potential hazards and associated risks are identified and classified as a) clearly unacceptable, b) tolerable if the cost of removing them would be prohibitive and c) acceptable. All unacceptable risks must be removed, but the tolerable risks are removed only if the cost and time can be justified.

GAMAB (globalement au moins aussi bon) or GAME (globalement au moins équivalent): the total risk in the new system must not exceed the total risk in comparable existing systems.

MEM (Minimum Endogenous Mortality): the risk from the new system must not exceed one tenth of the natural expected annual human mortality in the area where the device is to be deployed. For example, for people in their mid-20s in western countries this value is about 0.0002.

All these techniques must be then adjusted depending on the number of people that could be simultaneously affected by a dangerous failure of the equipment.

With ALARP, in order to decide which risks are unacceptable, tolerable, and acceptable, we need to determine numerical values for the maximum allowed probability of dangerous failure for each risk. With GAMAB and MEM we will need to determine this numerical value globally.

Techniques for proving software dependability

No single technique available to us is sufficient for proving that a software system meets its dependability requirements. Our demonstration of dependability must, therefore, be built using a complete arsenal that combines strategies and techniques. This arsenal can include but is not be limited to:

- a development environment that complies with IEC 62304 or another comparable standard
- requirements tracing matrices to ensure that all safety-related requirements are addressed

[1] See Chris Hobbs, *et al.* “Building Functional Safety into Complex Software Systems”, Parts I and II. QNX Software Systems, 2011. www.qnx.com.

- formal design methods and tools, which can provide mathematical proofs of design correctness
- fault-tree analysis using methods such as Bayesian Belief Networks
- retrospective design validation, which evaluates system design based on what has actually been built
- static analysis using methods such as model checking and data flow analysis
- testing using direct fault detection techniques such as dynamic analysis to identify faults through the errors and failures they cause

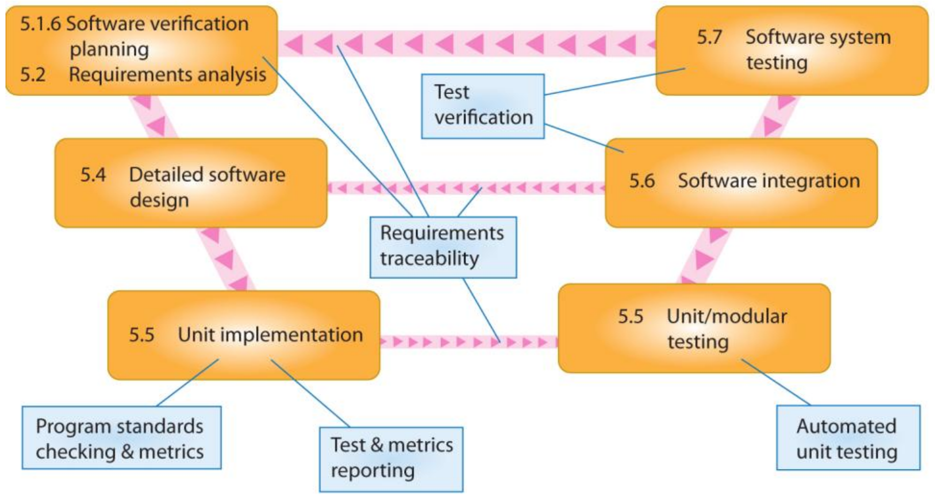
5.1.6 Software verification planning
5.2 Requirements analysis
5.7 Software system testing
Test verification
5.4 Detailed software design
5.6 Software integration
Requirements traceability
5.5 Unit implementation
5.5 Unit/modular testing
Program standards checking & metrics
Test & metrics reporting
Automated unit testing

Figure 1.Different analysis techniques and the relevant sections in IEC 62304 shown over a traditional "V" development model. None of the techniques shown is process dependant. A similar representation could be made for any other development process model: waterfall, iterative, agile, etc.

IEC 62304

IEC 62304 is becoming the *de facto* global standard for medical device software life cycle processes. The FDA has driven its development, and it is being harmonized with EU standard 93/42 EWG (MDD)[2].

Like the other standards shown in Figure 1, IEC 62304 draws on established industry-specific practices to complement the principles of IEC 61508. For example, unlike ISO 26262 or even IEC 61508 itself, IEC 62304 does not define common numerical values for acceptable failure rates (a Safety Integrity Level (SIL) rating). Instead, it defines safety classifications according to the level of harm a failure could cause to a patient, operator or other person. These classifications are analogous to the FDA classifications of medical devices: A (no possible injury or damage to health), B (possibility of non-serious injury or harm) and C (possibility of serious injury or harm, or death).

[2] Cristoph Gerber, ~~-I~~Introduction into software lifecycle for medical devices", Stryker Navigation: Presentation (4 Sept. 2008)

For the most part, the standards derived from IEC 61508 are similar in that they set out the processes (including a risk management process), activities and tasks required throughout the software lifecycle, stipulating that this cycle does not end with product release, but continues through maintenance and problem resolution as long as the software is operational. Ultimately, regardless of how they specify the level of acceptable or unacceptable risk, IEC 62304, IEC 61508 and other like standards provide guides and measures which we must use to demonstrate that our system meets its safety requirements.

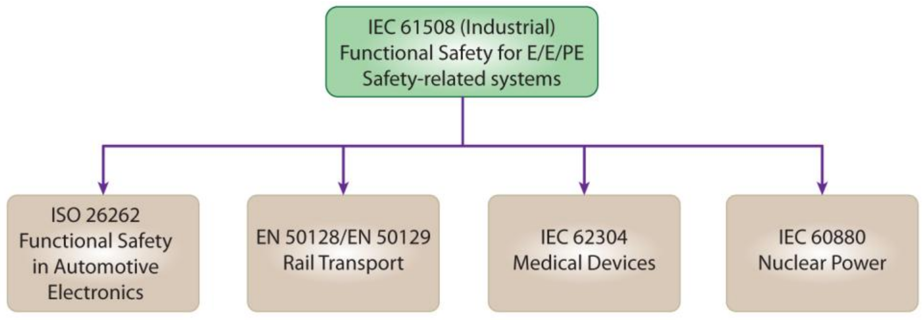

Figure 2.IEC 62304 is derived from IEC 61508 and hence shares its roots with other industry specific standards. Note that IEC 62304 expressly states that it does not depend on IEC 61508, but that IEC 61508 can be consulted for tools and techniques[3].

Dynamic analysis

Dynamic analysis is used to examine execution of the compiled source code, either in its entirety or on a piecemeal basis. Since dynamic analysis executes code, it tests not only the source code, but also the compiler, the linker, the development environment and, potentially, the target hardware. Dynamic analysis generally involves structural (code) coverage analysis and unit testing, which together can provide not only a very effective means for detecting errors in the software, but also evidence showing what software has been exercised and how this software has been exercised.

Structural coverage analysis is fundamental in the aviation industry standard DO-178B/C. While aviation accidents are dramatic and often tragic, and hence tend to make the news more often than do accidents with medical devices, the aviation industry does have an exemplary safety record. Mile for mile, flying is one of the safest modes of transport.

Structural coverage analysis

Dynamic analysis tools use either intrusive probes or non-intrusive probes. An intrusive probe system puts software probes (counts or procedure calls) into the code being analysed (high level language or assembler). These probes record information about the execution process and produce execution histories.

[3] Annex C.

Intrusive and non-intrusive probes

When using intrusive probes, demonstrating that the probes do not change the functionality of the instrumented code is essential to the validity of the analysis. In addition to proving that intrusive probes do not affect the source code, such a demonstration usually requires showing that the probes themselves introduce nothing which would expose weaknesses in the compiler. This can be achieved by using a Compiler Validation Suite (a set of source code artefacts designed to confirm that a compiler performs correct computations) to show that compiler validation is not affected by the instrumentation process.

A non-intrusive system obtains the same or similar information as does an intrusive system, but directly from the processor, and the dynamic analysis tool then relates this low-level information back to the original representation (high-level language or assembler). Unfortunately, for various reasons (such as the effects of compiler optimization) it is not always possible to establish this relationship unambiguously.

Note that, as with all testing, in a complex software system it is never possible to demonstrate with absolute certainty that the probes associated with structural coverage analysis do not affect the behaviour of the code. For instance, by definition Heisenbugs are irreproducible; usually considered to be caused by subtle timing conditions, they may be corrected (or even introduced!) by any changes to the code, including instrumentation.

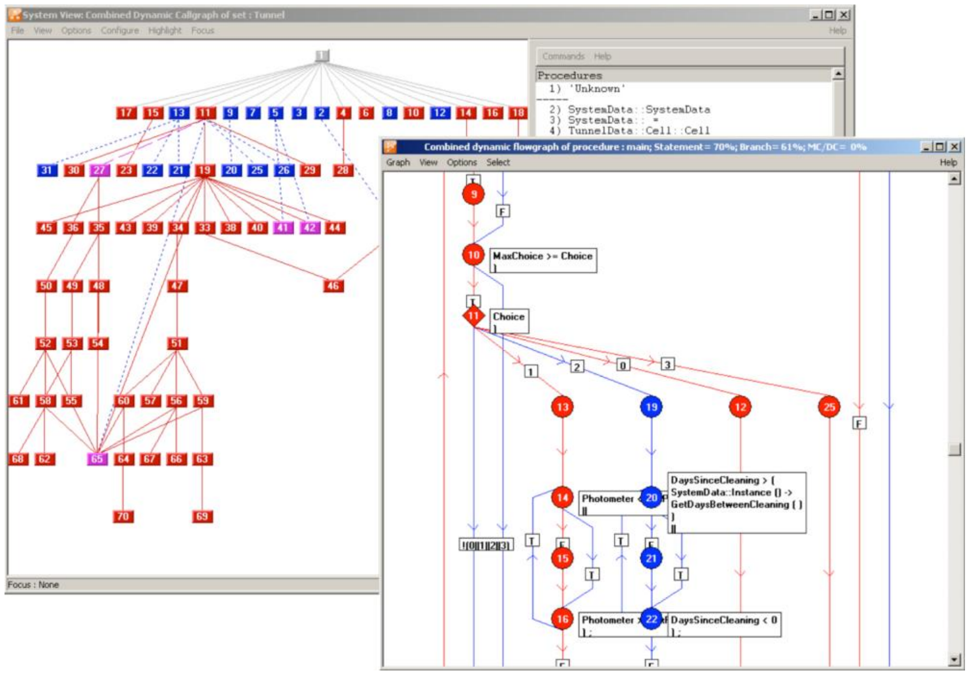
System View: Combined Dynamic Callgraph of set : Tunnel
File View Options Configure Highlight Focus
Help
Commands Help
Procedures
1) 'Unknown'
2) SystemData::SystemData
3) SystemData::=
4) TunnelData::Cell::Cell
Focus : None
Combined dynamic flowgraph of procedure : main; Statement = 70%; Branch = 61%; MC/DC = 0%
Graph View Options Select
Help
MaxChoice >= Choice
Choice
Photometer
DaysSinceCleaning > (SystemData::Instance () -> GetDaysBetweenCleaning ( )
Photometer
DaysSinceCleaning < 0

Figure 3. In screen captures from an LDRA code coverage tool, colour-coded graphical information clearly identifies unexercised code.

Evidence for the dependability estimate

The trick, then, is not to prove the absence of bugs (a formal impossibility), but to gather evidence that we can include in our estimate of the software's dependability. In particular, if we use SOUP (Software Of Unknown Pedigree) in our system, structural coverage analysis can help show that there is no unused or superfluous code, in compliance with, for instance, ANSI/AAMI/IEC TIR80002-1:2009[4], table B.2: ―Use only the SOUP features required: remove all others".

Unit testing

Unit testing verifies small units, making it relatively simple to observe incorrect behaviour and hence to detect faults. With unit testing, procedures or collections of procedures are tested in isolation from the complete system in order to establish that they satisfy specific requirements.

Typically, these requirements are more comprehensive than those of the project, so that, for example, boundary conditions can be tested: a test for rendering a screen display of 750 x 1,000 pixels may test up to, say, 1,200 x 1,600 pixels. The interface to each procedure is tested for input values that may be excluded by higher-level procedures, exploring generality—that procedures always behave as required.

Unit testing provides access for the exploration of housekeeping code, and otherwise infeasible, protective code components can similarly be tested. Some instances of coincidental correctness can be removed; for instance, in the bigger system, a procedure may be called when it should not be or vice versa and yet leave the observer with the impression that all is well. Because we are dealing with a smaller component, it is easier to observe incorrect behavior and hence detect faults,

The issue of how to handle procedures called by the unit under test is dependent of the purpose of the particular test in question. Indeed, unit testing traditionally employs a bottom-up testing strategy (sometimes called module or integration testing), where units are tested then integrated with other test units. Where called functions are excluded from the tests they can be replaced by ―stubs".

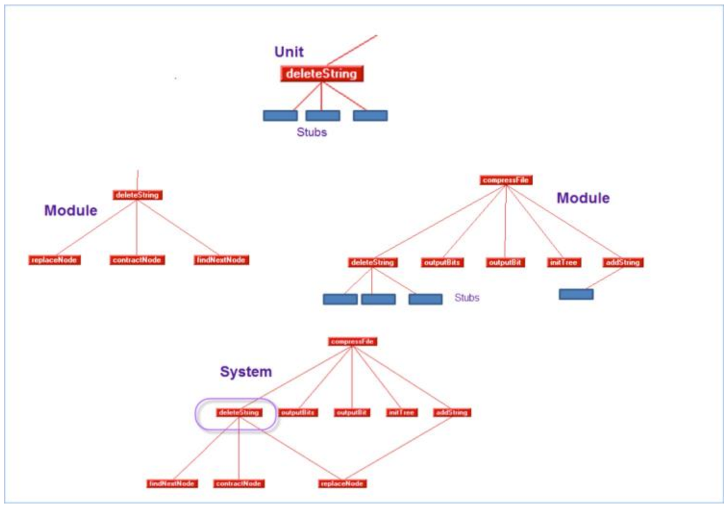

Figure 4.Performing structural coverage analysis on the whole or a subset of the system provides great flexibility.

[4] *Guidance on the application of ISO 14971 to medical device software*

When combined with structural coverage analysis, the flexibility of being able to include as much or as little of a call tree as desired in the tests facilitates achieving the coverage required from the most demanding qualification and certification authorities.

Structural coverage metrics

One of the most difficult steps to take when validating any system is deciding when to stop testing. This decision should be made in the context of the system's dependability requirements, and ultimately it depends on the IEC 62304 and regulatory agency safety classifications of the medical device using our system.

Coverage metrics can help gauge how much has been achieved by dynamic testing, and can be used to inform the decisions about how much testing remains to be done. These metrics include:

- Statement Coverage: the most basic metric, which consists of the proportion of statements in the system that have been executed.
- Branch/Decision Coverage: the proportion of control flow branches covered. On average, each statement and each procedure call is executed twice as often as for statement coverage alone.
- LCSAJ Coverage: a path-related metric, LCSAJ (linear code sequence and jump) coverage is more demanding than branch/decision coverage and is potentially useful in the most critical parts of the application. It is available with the more sophisticated tools on the market.
- Modified Condition/Decision Coverage: full MC/DC coverage is achieved when every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently.

Choosing a software analysis tool

All software tool vendors are keen to sell their wares and, understandably, few vendors are particularly keen on advertising what their tools might not do. The following are a list of key points to consider when evaluating a software analysis tool.

Fault reporting

- Does the tool produce many false positive reports; that is, does it report faults that are not in fact present?
- Does the tool produce false negative reports; that is, does it fail to report defects that are in fact present?

Project compatibility

- Does the tool take too long when viewed against the overall benefits of the information it generates? The time a tool takes to run is not usually an issue, but this should still be a consideration in case it is excessive and hence becomes a problem for the project.

- Does the tool support the preferred dialect for the project where it is to be used? Most compilers implement their own version of the language in which the code to be analyzed is written. It is essential, therefore, to ensure that analysis tools support the language variant used in the project.
- How readily can the tool be incorporated into the development process? A tool is of little use if it requires a disproportionate effort to integrate into the project.

Capabilities and limitations

- Does the tool work across the complete system? This is an important question because some faults can only be detected when the whole system is analyzed.
- Is the tool capable of accommodating inter-procedural recursion? Even in a single file, inter-procedural recursion is important if a procedure can only be analyzed fully once another procedure has been analyzed.
- What are the tool's limitations? All tools have limitations, including the amount of code they can analyze, the depth of blocks they can handle, the bracket nested depth they permit, their symbol table size, etc. These limitations and their implications for the project should be noted and understood.

Conclusion

With complex software systems at the heart of so many medical devices, the success of these devices is increasingly dependent on the manufacturer's ability to demonstrate that these systems meet required levels of dependability. While regulatory agencies such as the FDA, MDD and MHRA approve the entire device and not its parts for market, the evidence presented to demonstrate the dependability of the device software (the software Safety Case) are essential for device approval. Hence, close attention to design and development practices, and a careful choice of validation techniques and the tools used to implement these are essential to the success of any medical device project that involves software.

Appendix A: IEC 62304 and development activities

The tables in this section map paragraphs in IEC 62304 with software design, development and validation activities. Adherence to IEC 62304 doesn't guarantee that the software will meet sufficient dependability requirements or that agency approvals will be forthcoming. However, it will help ensure that the project follows good processes, that requirements are clear at all levels, and that a Safety Case can be built for the completed product.

Legend

+” The method is recommended for this Class.

✔” Software test tools are likely to aid test effectiveness and efficiency.

5.2 Software requirements analysis		**Class**		
		A	**B**	**B**
5.2.1	Define and document software requirements from SYSTEM requirements	+ ✔	+ ✔	+ ✔
5.2.2	Software requirements content	+ ✔	+ ✔	+ ✔
5.2.3	Include RISK CONTROL measures in software requirements		+ ✔	+ ✔
5.2.4	Re-EVALUATE MEDICAL DEVICE RISK ANALYSIS	+ ✔	+ ✔	+ ✔
5.2.5	Update SYSTEM requirements	+ ✔	+ ✔	+ ✔
5.2.6	Verify software requirements	+ ✔	+ ✔	+ ✔

Table A1. Test tool capabilities mapped to IEC 62304 section 5.2 Software requirement analysis

5.5 Software unit implementation and verification		**Class**		
		A	**B**	**C**
5.5.1	Implement each software unit SOFTWARE UNIT	+ ✔	+ ✔	+ ✔
5.5.2	Establish SOFTWARE UNIT VERIFICATION PROCESS		+ ✔	+ ✔
5.5.3	SOFTWARE UNIT acceptance criteria		+ ✔	+ ✔
5.5.4	Additional SOFTWARE UNIT acceptance criteria			+ ✔
5.5.5	SOFTWARE UNIT VERIFICATION		+ ✔	+ ✔

Table A2. Test tool capabilities mapped to IEC 62304 section 5.5 Software unit implementation and verification

5.5.4 Additional SOFTWARE UNIT acceptance criteria	**Class C**
a) proper event sequence	+
b) data and control flow	+ ✓
c) planned resource allocation	+
d) fault handling (error definition, isolation, and recovery)	+
e) initialisation of variables	+ ✓
f) self-diagnostics	+
g) memory management and memory overflows	+ ✓
h) boundary conditions	+ ✓

Table A3. Test tool capabilities mapped to IEC 62304 section 5.5.4 Additional SOFTWARE UNIT acceptance criteria

5.7 Software SYSTEM testing		**Class**		
		A	**B**	**C**
5.7.1	Establish tests for software requirements		+ ✓	+ ✓
5.7.2	Use software problem resolution PROCESS		+ ✓	+ ✓
5.7.3	Retest after changes		+ ✓	+ ✓
5.7.4	Verify SOFTWARE SYSTEM testing		+ ✓	+ ✓
5.7.5	SOFTWARE SYSTEM test record contents		+ ✓	+ ✓

Table A4. Test tool capabilities mapped to IEC 62304 section 5.5.4 software system testing

Appendix B: the operating system

No matter how good the validation tools, ultimately it is the device and its software that must receive approval. In any system that uses software, everything above the silicon depends on the OS. This means that any medical device that includes a software component can only be as dependable as its OS. This OS must be able to support the claims we make about the device's safety.

A comprehensive discussion of requirements for OSs used in safe systems would fill more than a few library shelves. It is, nevertheless, worth noting, at a very high level, some key OS requirements we should look for when selecting the OS for our safe system.

Real-time guarantees

Only a real-time operating system (RTOS) is designed to ensure the timely responses required for the dependability that is fundamental to any safe software system.

Architecture

A failure in a real-time executive or monolithic OS usually requires a device reboot, compromising system availability. With a microkernel RTOS, applications, device drivers, file systems, and networking stacks all reside outside the kernel in separate address spaces, and are thus isolated from both the kernel and each other. A fault in one component will not bring down the entire system.

Memory protection

The OS architecture should separate applications and critical processes in their own memory spaces so that a fault cannot propagate across the system.

Priority inheritance

To protect against priority inversions the RTOS should support assigning, until the blocking task completes, the priority of a blocked higher-priority task to the lower-priority thread doing the blocking.

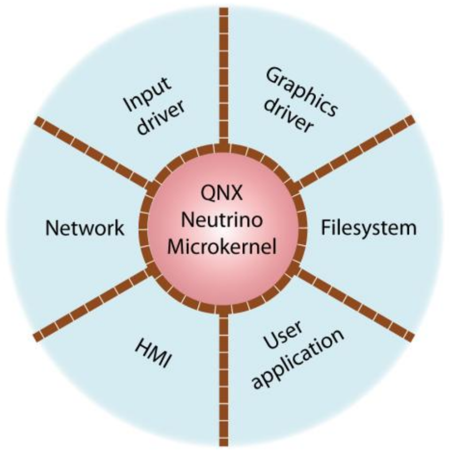

Figure 5. In a microkernel RTOS, system services run as standard, user-space processes. A failure in one user-space is isolated to that space; the microkernel and other user-spaces are protected.

Partitioning

To guarantee availability, the RTOS should support fixed or, preferably, adaptive partitioning, which enforces resource budgets but uses a dynamic scheduling algorithm to reassign CPU cycles from partitions that are not using them to partitions that can benefit from extra processing time.

High availability

A self-starting software watchdog should monitor, stop and, if safety can be assured, restart processes without requiring a system reset. If a restart is not a safe alternative, then the watchdog should set the system to its design safe state.

References

Center for Devices and Radiological Health, U.S. Food and Drug Administration. —Infusion Pump Improvement Initiative”. Washington, April 2010. <http://www.fda.gov/MedicalDevices/Productsand MedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm205424.htm>

Gerber, Cristoph. —Introduction into software lifecycle for medical devices”. Freiburg, Germany: Stryker Navigation. Presentation (4 Sept. 2008).

Green, Blake. —Understanding Software Development from a Regulatory Viewpoint”. *Journal of Medical Device Regulation*, 6:1 (Feb. 2009), pp. 14-23.

Hall, Ken and StypeTriteq. —Developing Medical Device Software to IEC 62304”. *European Medical Device Technology*. 1 June 2010. <http://www.emdt.co.uk/article/developing-medical-device-software-iso-62304>

Heneghan, Carl, *et al.* —Medical-device recalls in the UK and the device-regulation process: retrospective review of safety notices and alerts”. *BMJ Open*, 15 May 2011. <http://bmjopen.bmj.com/content/early/2011/05/12/bmjopen-2011-000155.full>

Hobbs, Chris. —Clear SOUP and COTS Software for Medical Device Development”. QNX, 2011. <http://www.qnx.com/download/feature.html?programid=22793>

Hobbs, Chris, *et al.* —Building Functional Safety into Complex Software Systems, Part I”. QNX Software Systems, 2011. www.qnx.com.

______. —Building Functional Safety into Complex Software Systems, Part II”. QNX Software Systems, 2011. www.qnx.com.

International Electrotechnical Commission. *IEC 62304: Medical Device Software–Software Lifecycle Processes*. First edition, 2005-2006. Geneva: International Electrotechnical Commission, 2006.

Jackson, Daniel et al., eds. *Software for Dependable Systems: Sufficient Evidence?* Washington: National Academies Press, 2007.

Jackson, Daniel et al., eds. *Sufficient Evidence: A Briefing of the National Academies Study Software for Dependable Systems*. <cstb.org/pub_dependable>

Kumar, Anil. —Easing the IEC 62304 Compliance Journey for Developers to Certify Medical Devices”. Medical Electronic Device Solutions. 20 June 2011. <www.medsmagazine.com/articles/view/118>

LDRA. —Implementing IEC 62304 with the LDRA tool suite”. Liverpool: LDRA Ltd., 2011. <www.ldra.com>

McCabe, T. A complexity measure. IEEE Transactions on Software Engineering, 2(4):308–320, 1976.

Moore, Janet. —Study calls for more FDA scrutiny of medical devices”. *Star Tribune.* 16 Feb. 2011. <http://www.startribune.com/business/116203594.html>

U.S. Food and Drug Administration. *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*. 11 Jan. 2002. <www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085281.htm>

U.S. Food and Drug Administration. *Research Project: Static Analysis of Medical Device Software*, updated 11 Feb. 2011. <www.fda.gov/MedicalDevices/ScienceandResearch/ucm243156.htm>

Winstein, Keith J. —Medical Devices Face New Scrutiny from FDA”. *The Wall Street Journal*. 8 April 2009. <http://online.wsj.com/article/SB123920937438601763.html>

Wood, Jonathan, —Medical Devices Under Scrutiny”. *Oxford Science Blog*. University of Oxford. 17 May 2001. <http://www.ox.ac.uk/media/science_blog/110517.html>