QNX SOFTWARE SYSTEMS

Clear SOUP and COTS Software for Medical Device Development

Chris Hobbs, Senior Developer, Safe Systems
QNX Software Systems Limited
chobbs@qnx.com

Abstract

In many industries, manufacturers have reduced development times by using COTS (commercial-off-the-shelf) software and hardware in their products. Pressures to bring new, feature-rich products to market quickly affect medical device manufacturers as much as anyone, but the industry may be reluctant to follow suit due to well-justified concerns that COTS implies SOUP (software of uncertain provenance), and thus may compromise device safety and pre-market approval by the FDA and other regulatory agencies.

While we should indeed exercise diligence and caution when considering COTS software for medical devices, neither the IEC 62304 "software for medical devices" standard, nor the demands of functional safety preclude its use. In fact, COTS software may be perfectly acceptable, given stringent selection criteria, and appropriate and equally stringent validation of the completed systems and devices. If we make the fine but critical distinction between *opaque* SOUP[1] (which should be avoided) and *clear* SOUP, that is, SOUP for which source code, fault histories and long in-use histories are available, we will find that COTS software may be the optimal choice for many safety-related medical devices.

The Usual Challenges

Medical device manufactures face the same challenges as everyone building complex systems: time, quality, size (number and complexity of features), and cost. To this we must add functional safety, and pre-market approval by the FDA, MDD, MHRA, Health Canada and their counterparts in every jurisdiction where the device will be used. This approval is the *sine qua non* of medical devices. Without it, a device might as well not exist.

That said, approval does not fundamentally change the challenges we face when designing a software system for a medical device; nor does it fundamentally change our choice of solutions. These are, essentially, two: a) design and build everything ourselves (sometimes called "roll-your-own"), from the OS or even the BSP[2] up; and b) use available software components where possible and advantageous, integrating them with our own components into the system we design.

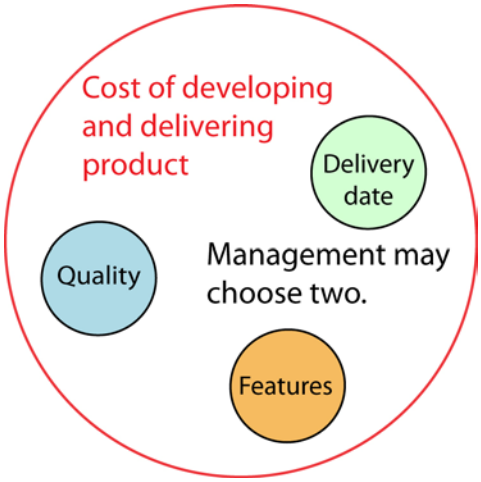

Figure 1. Development is a series of compromises between feature sets, delivery dates and quality. A change to any one of these affects the others. For example, shortening the delivery schedule reduces the feature set or the quality, or both. The total cost of developing and delivering a product is a function of these compromises.

Roll-your-own is probably only a viable choice for a simple system, whose functionality is so limited that it doesn't even require a full OS. For anything more complex, attempting to do everything ourselves would likely take more time and effort, and entail far

[1] Opaque SOUP is sometimes, jokingly, called "pea SOUP" in contrast to clear SOUP, which has not yet been dubbed "broth".

[2] Board Support Package: board-specific software for a specific OS, containing the minimal device support required to load the OS, including a bootloader and drivers for the devices on the board.

more risk than building a system with carefully selected components. The trick, of course, is to determine what available software can be integrated into our medical system without compromising its functional safety and approval requirements, then to demonstrate that the completed system meets these requirements.

Deterministic and Non-deterministic Systems

In theory, any software system is deterministic; that is, every state and state change in the system can be known, documented and tested. In *practice*, however, software systems have become so complex that they should be treated as non-deterministic systems.[3] In practice, we cannot know or predict all their possible states and state changes.

A crucial implication of this practical fact is that it is no longer possible to rely wholly on testing when validating a software system. A single truth underlines why testing is insufficient validation in a system where all states and state changes may not be known and documented: testing can only demonstrate the presence of faults; it cannot prove that there are no faults. Medical device software is no exception. In fact, in *Medical device software—Part 1: Guidance on the application of ISO 14971 to medical device software*, the AAMI cautions that a pitfall to avoid is "Depending on testing as a RISK CONTROL measure—even though 100% testing is impossible"[4].

A simple example of an elevator controller illustrates well the limits of testing. Figure 2 shows a state diagram for such a controller, which sends instructions to an elevator and to the elevator doors in a building. This controller is very simple, but it contains a fault that may not be immediately apparent. If, for example, we think of the danger that someone might fall into the elevator shaft if a door opens without the elevator present and design the system to prevent this failure, we have not necessarily ensured that someone does not get stuck in the elevator.

With the controller shown in Figure 2, we can get on the elevator on the bottom floor, then find ourselves on an endless journey from floor to floor. The elevator doors never need to open. To save ourselves, we would have to find a way to get someone outside the system to either inject an `!open` instruction when the elevator reaches a floor and before the controller issues another `!up` or `!down` instruction, or to force the controller to issue an `!open` instruction after a specified number of ups and downs, in much the same way that telecommunications networks drop undeliverable packets.

This simple scenario underlines one of the key challenges we face when we attempt to verify even very simple systems. We forgot to ask if all elevator rides must end, and designed a system with a very serious fault. Of course, as a system increases in complexity, so will the number of these sorts of faults.

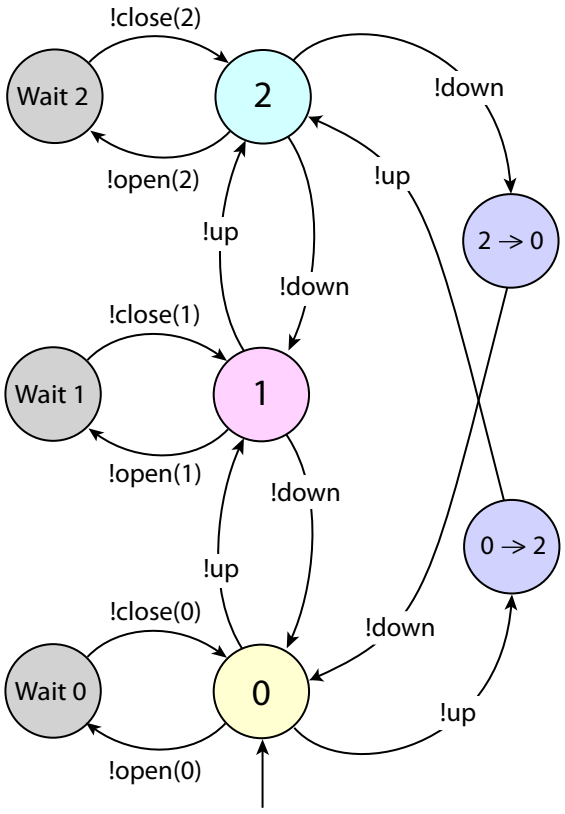

Figure 2. A simple elevator system with a fault: there is no guarantee that the elevator cage doors open at a floor before the elevator is called to another floor.[5]

[3] The decreasing cost of multicore processors supporting ever-greater and sophisticated feature sets has meant a rapid increase in software complexity.

[4] AAMI, *Medical device software—Part 1: Guidance on the application of ISO 14971 to medical device software*, 2009. p. 55..

[5] Adapted from Gerard J. Holzmann, *The SPIN Model Checker*. Boston: Addison-Wesley, 2004.

In short, if ever it was possible to use testing to demonstrate that a system did not contain faults, this was so only because the system was so simple that it was deterministic, in both theory and in practice[6]. We could know and test all possible states and state changes; we could know and ask every required question. Using formal model proving to augment testing, we can show the correctness of designs but, as the simple elevator example makes clear, we can do so only if we ask the right questions. We cannot abandon testing, of course, but we must treat its results statistically and complement it with other methods, such as stringent process controls, design validation, and statistical analysis of fault histories.

Functional Safety

Malfunctions in medical devices don't usually make the same sort of headlines as airplane or train accidents. For the patient, however, the consequences of an error in a medical device system may be just as dramatic or tragic, and the medical device industry is morally, legally and financially bound to ensure as much as is humanly possible that its products do no harm.

Despite the enormous effort invested in validating the functional safety of medical devices, faults, errors and failures continue to appear. For example, the FDA recorded that there were 200,000 pacemaker recalls due to software in the U.S. between 1990 and 2000, and that between 1985 and 2005 there were 30,000 deaths and 600,000 injuries from medical devices (1985-2005), of which some eight percent were attributable to software.[7]

Functional safety refers to the capacity of a safety-related system to function as it needs to function to maintain the safety of the system: it is the continuous operation of a safety-related system performing its primary task while ensuring that persons, property and the environment are free from unacceptable risk or harm[8]. In short, a functionally safe system does what it is designed to do and doesn't unintentionally harm anyone or anything. A radiation therapy unit, for instance, is functionally safe if it does not inflict unacceptable harm to the operator, other persons or the environment, or healthy cells in the patient. Its functional safety is not compromised by the harm it may do to cancerous cells, because this is its intended use.

What Is *Functional* Safety? An Example

Consider a medical device that emits potentially hazardous electro-magnetic radiation (e.g., X-rays). If a technician were to remove the protective shield while the radiation was turned on it could potentially create "unacceptable injury or harm" to her.

Having identified this hazard, we could address it in several ways. We could build the equipment so that it would be physically impossible for the shield to be removed when the switch was in the "on" position: the switch in that position physically covered the shield. This solution would be a safe one, but it would not be one of functional safety, because safety is intrinsic and passive: it does not rely on the continued functioning of any subsystem.

Alternatively, we could create an active subsystem that detected the removal of the shield and shut off the radiation before it became a hazard. This subsystem would constitute functional safety: the continued safety of the overall system depends on the subsystem continuing to function correctly.

Thus, building a safe system can rely on both passive safety measures (such as a design that makes removing the shield impossible) and on active measures (a system that detects shield removal and shuts down the radiation). In practice, most complex systems use a combination of both. In the context of our discussion of COTS, the assumption is that the safe operation of the medical device requires some sort of active system, hence functional safety.

Multiple standards exist specifying both what constitutes a functionally safe system in a given context, and the processes and activities that must be scrupulously followed throughout a product or

[6] The Engineering Safety Management Yellow Book 3, *Application Note 2: Software and EN 50128*, published by Railway Safety on behalf of the UK railway industry even suggests that "if a device has few enough internal stored states that it is practical to cover them all in testing, it may be better to regard it as hardware." p. 3.

[7] Daniel Jackson *et al.*, eds. *Software for Dependable Systems: Sufficient Evidence?* Washington: National Academies Press, 2007. p. 23.

[8] Chris Hobbs *et al.*, "Building Functional Safety into Complex Software Systems, Part I". QNX Software Systems, 2011.

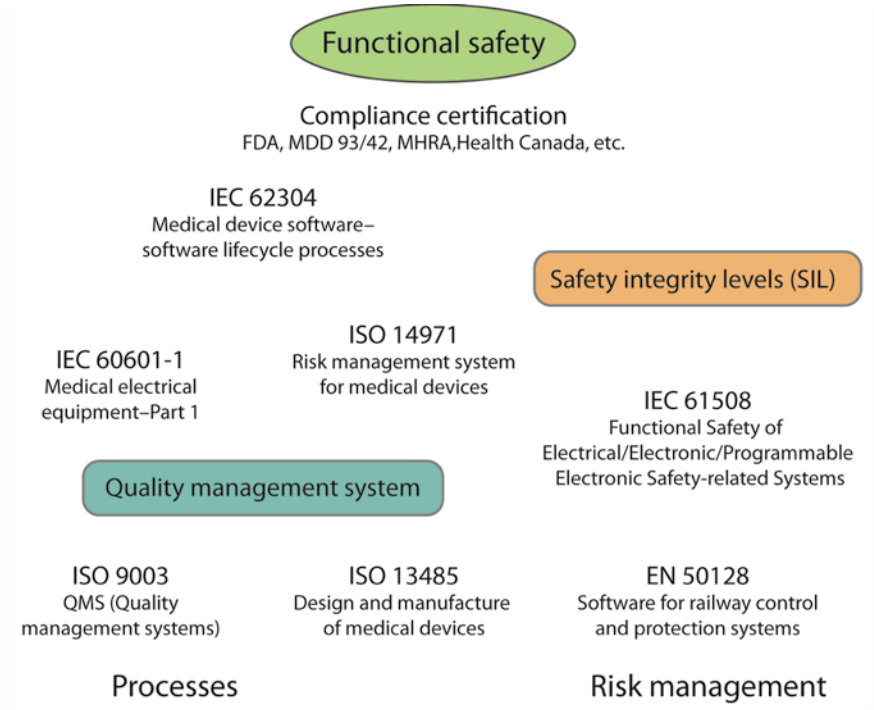

Figure 3. Some of the standards contributing to functional safety in medical devices.

system lifecycle in order to ensure the product's or system's functional safety. Among the best known are IEC 61508 (electrical, electronic, programmable), ISO 26262 (automotive) and the CENLEC 5012*x* series (rail transportation), which define functional safety and target safety integrity levels (SIL) for systems in their respective areas, and specify the processes and activities for demonstrating these.

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) [9]. Unlike IEC 61508 and EN 50128, for example, IEC 62304 does not define common numerical values for acceptable failure rates; conformity to IEC 62304 doesn't imply a SIL rating as does, for instance, conformity to IEC 61508, which is meaningless without one (e.g. IEC 61508 SIL3).

IEC 62304 is limited to the "framework of life cycle PROCESSES with ACTIVITIES and TASKS necessary for the safe design and maintenance of MEDICAL DEVICE SOFTWARE"[10]. It makes two assumptions about quality management and risk management: a) that the software for the medical device is "developed and maintained within a quality management system" (i.e. ISO 13485 or ISO 90003); and b) that risk management conforms to ISO 14971, plus additional minor requirements for software addressed in IEC 62304, Clause 7.

Significantly, IEC 62304 does not prescribe how its requirements are to be met; that is, it does not specify a software development model or particular documents that must be submitted to support claims of conformity. IEC 62304 calls up ISO 14971, which sets the requirements for a medical devices risk management system. Neither of these standards specifies risk levels, or prescribes or proposes a method for determining the probability of software failure based on traditional statistical methods[11]. We are free to choose the development model most appropriate to our requirements, as we are free to choose the methods we ultimately use to validate the functional safety claims we make about the systems we build.

IEC 62304 does, however, 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. It also 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).

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

[10] International Electrotechnical Commission, *IEC 62304: Medical Device Software—Software Lifecycle Processes*. First edition, 2005-2006. Geneva: International Electrotechnical Commission, 2006. Introduction.

[11] Gerber, slide 19.

Finally, and of particular importance in the context of this discussion, IEC 62304 explicitly mentions SOUP, which it defines as a

> SOFTWARE ITEM that is already developed and generally available and that has not been developed for the purpose of being incorporated into the MEDICAL DEVICE (also known as "off-the-shelf software") or software previously developed for which adequate records of the development PROCESSES are not available.[12]

It is important to note that in the above, IEC 62304 a) assumes that off-the-shelf software (commercial or otherwise) will be used, and b) offers two definitions of SOUP, which can be either (or both of) software not developed for the medical device in question, or software with unavailable or inadequate records of its development processes.

IEC 62304 does not prohibit using SOUP in a medical device, and in fact several clauses in the standard are written with the assumption that SOUP will in fact be used. Section 5.1.1 "Software development plans", for instance, states that "The plan shall address … software configuration and change management, including SOUP CONFIGURATION ITEMS"[13], and SOUP is the explicit subject of sections such as 5.3.4 "Specify SYSTEM hardware and software required by SOUP item".

The question, then, is not whether it is permissible to use COTS software and/or SOUP in medical device software, but a) how to decide whether a particular COTS software or SOUP item is appropriate for the medical device in question, and b) how to validate that this COTS item or SOUP item supports the functional safety requirements for this medical device. To answer this question we should start by attempting to add some precision to our definition of SOUP, and to the relationship between SOUP and COTS software.

SOUP and Clear SOUP

Some software vendors make a rather simplistic—and incorrect—distinction between COTS and SOUP. COTS, they say, has a vendor standing behind it, a company that has staked its reputation—and, not incidentally, its financial future—on this software functioning as specified, while SOUP has no one standing behind it.

This distinction is valid in the same way that it may be preferable to buy medication from a reputable pharmacy rather than from some web site that uses spam to advertise. However, it is also largely irrelevant, since for us most COTS software is quite likely also SOUP; the processes the vendor followed (or failed to follow!), the source code, fault histories, and indeed everything else we would have available if we were developing the product ourselves may not be available to us or to anyone else outside the vendor's organization.

A more useful distinction is between (opaque) SOUP and *clear* SOUP. This distinction is not based on any commercial criteria (commercial or not commercial). It is founded in the artefacts available to support a safety case for the software. These artefacts are needed to support our claims about the risks and safety levels of the systems we build with the SOUP.

Quality and Approval

Pre-market approval by regulatory agencies is inseparable from quality, but the two are not interchangeable. First, it is quite easy to image a device or tool, such as a syringe, that performs its primary function (puncture a vein and draw blood) extremely well, but that could never receive approval because the process used to manufacture it includes no sterilization process; or, the syringe could meet all approval requirements: be sharp, sterile, and safe and easy to use, but have a tendency to disintegrate during initial sterilization and thus, though it poses no danger to patients or medical practitioners, it does not meet required standards for quality.

A COTS system, such as a Microsoft Windows OS, may have a well-documented development process, its vendor presumable adheres to this well-defined and documented development process and is in possession of the source code which it can readily examine, and has tracked and documented the software's failure history. However, since this information is not available for public scrutiny, as far as we are concerned the Windows OS is (opaque) SOUP.[14]

[12] IEC 62304 3.29 SOUP.

[13] IEC 62304 5.1.1 Software development plans.

[14] Here the analogy with a medication bought from a reputable pharmacy or through a spamming web site falls apart. The medication sold by the pharmacy is not like the opaque software: every ingredient (including the "inactive" ones), every process used to create or extract

OS Architecture

The OS on which the COTS software runs must support the vendor's functional safety claims. We must, therefore, evaluate the OS, and its architecture in particular, since the OS architecture is critical to system dependability. Important characteristics to look for are:

Pre-emptible kernel operations — to ensure that the system meets realtime commitments, the RTOS must allow kernel operations to be preempted, and time windows during which preemption may not occur should be brief.

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.

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.

In contrast, open source projects such as Apache and Linux have their source code and fault histories freely available to anyone who cares to examine them. Thanks to years of active service, this software's characteristics are well known. Like in-house software, this software, though it is, literally, "of uncertain provenance", can be scrutinized with code symbolic execution and path coverage analysis, and the software's long (and freely available) history make findings from statistical analysis particularly relevant.

Hence, we can consider software developed in these open source projects *clear* SOUP; that is, SOUP that we can examine, verify and validate as though we had written it ourselves.

Despite these attractive characteristics, open source software may not be the best solution for medical systems, however. The difficulty with using open source software in functionally safe systems is that the processes for open source development are neither clearly defined nor well documented—and this is precisely what concerns IEC 62304. We can't know how the software was designed, coded or verified, and validating functional safety claims without this knowledge is an improbable endeavour. Add to this that SOUP or COTS software may include more functionality than is needed, which leaves dead code in the system, a practice that functional safety standards, such as IEC 61508, expressly discourage.

Of course, if a COTS software vendor makes available its product's source code and fault history, it clarifies its SOUP. Some vendors choose to go one better and provide, not just clear SOUP, but a clear *recipe* for the SOUP. That is, they release to their customers the detailed processes they use to build their software, along with its complete development history—essentially an informal audit trail that we can use to help substantiate claims about the software's reliability and availability. Some vendors may even go a step further and make available for scrutiny the evidence they presented in order to obtain certification (e.g. IEC 61508 SIL3) for their product.

In addition to its importance for the medical device's initial approval, the COTS software recipe for clear SOUP (documented development and validation artefacts and histories, and documented processes) can prove invaluable for subsequent validation and approval following product upgrades.[15] It is worth noting, for example, that

> In a study the FDA conducted between 1992 and 1998, 242 out of 3,140 device recalls (7.7 percent) were found to be due to faulty software. Of these, 192—almost 80 percent—were caused by defects introduced during software maintenance.

these ingredients, and the finished medication must be available for regulatory scrutiny. This is why pharmaceutical and biotechnology companies rely so heavily on their patents; they may not keep trade secrets, and hence patents are their only protection.

[15] See Anil Kumar, "Easing the IEC 62304 Compliance Journey for Developers to Certify Medical Devices" *Medical Electronic Device Solutions*, 20 June 2011.

In other words, the faults were introduced after the devices had gone to market: the devices worked, and then someone either broke them or uncovered previously undiscovered faults. Ideally, then, when developing *and maintaining and upgrading* software systems for medical devices where functional safety is an issue (IEC 62304 B and C class devices) we should work with clear SOUP made with a clear recipe that has a long and well-documented history of success in the field.

Shopping for COTS Software

At the highest level, the question we must ask of any COTS software we are considering for a medical device is "What proofs does the software vendor provide that his product is what we need?" In addition to all the standard questions about functionality, features, cost, support and so on, this question must include "Will this COTS software support us in getting approval for our medical device?"

If we assume that, since we did not develop it ourselves, all COTS software is SOUP of some sort, then we must find out what sort of SOUP it is. If it proves to be "software previously developed for which adequate records of the development PROCESSES are not available" we may have difficulty justifying its use in our system. First, validating functional safety claims for systems with such software requires substantial additional effort and expense. Second, this software cannot meet the IEC 62304 requirement for well-documented and rigorously followed software lifecycle processes.

If, on the other hand, the COTS software is clear SOUP, our task may be significantly less arduous. That is, if the software was not "developed for the purpose of being incorporated into the MEDICAL DEVICE" but adequate records of its development processes are available, it may be a good candidate for our medical device.

COTS Checklist

The following can be used as a high-level checklist to help us determine if a specific COTS component is a good candidate for integration into our medical device software system—essentially, if the COTS software is clear SOUP. Our decisions concerning the COTS software will necessarily depend on how well it supports the functional safety and compliance requirements we have specified for our complete system.

Functional Safety Claims

We can begin by examining the functional safety claims the COTS software vendor makes about his software.

Does the vendor make any functional safety claims?	
Do these claims meet the functional safety requirements for our project?	
Are the context and limits of the claims specified? For instance, are these claims for continuous operation or for on-demand operation?	
Do the COTS software functional safety claims specify the probability of dangerous failure? Or, inversely, what claims does the vendor make about the software's dependability?	
Does the vendor define "sufficient dependability", and how does he quantify his dependability claims? For example, is the quantification of the (essentially meaningless) "five-nines" type, or does it provide meaningful information about availability and reliability in relevant contexts?	
Does the vendor quantify the COTS software claims of: • Availability: How often does the system respond to events in a timely manner? • Reliability: How often are these responses correct?	

Process

A defined and documented process covering the entire software lifecycle is a *sine qua non* requirement; without this, we need not go further.

Has the COTS software vendor implemented a quality management system (QMS)?	
Does this system meet the requirements of one of: • The ISO 9000 family of QMS standards? • ISO 15504 (Software Process Improvement Capability Determination (SPICE)? • Capability Maturity Model Integration (CMMI)?	

What processes does the vendor use for source control, including revisions and versions?	
How does the vendor document, track and resolve defects, including those found through verification and validation, and in the field?	
Does the vendor classify defects and follow up with fault analysis?	

Fault-tree Analysis

Fault-tree analysis, using a method such as Bayesian believe networks is an essential tool both for discovering and resolving design errors, and for estimating system dependability. It also provides artefacts that can be reviewed by auditors and agencies that must approve the medical device for market.[16]

Was the COTS software evaluated with fault-tree analysis?	
Did the analysis use both *a priori* (cause to effect) and *a posteriori* (effect to cause) evidence?	
Are the results of the fault-tree analysis available to us?	

Static Analysis

Static analysis is invaluable for locating suspect code, and its use has been recommended by agencies such as the FDA,[17] which is "investigating various static analysis techniques, e.g., symbolic execution, abstract interpretation, and reverse engineering, and applying these to analyze software in medical devices…." The goal of this undertaking is to improve the ability of the FDA's Center for Devices and Radiological Health (CDRH) "to assess software quality, both during pre-market and post-market reviews" and to "improve the state-of-the-art in static analysis technology by improving precision and efficiency of static analysis tools, specifically applied to medical device software".[18]

[16] See Chris Hobbs, "Fault Tree Analysis with Bayesian Belief Networks for Safety-Critical Software", 2010.

[17] For example, "static analyses provide a very effective means to detect errors before execution of the code…" FDA, *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*. 11 Jan. 2002.

[18] FDA, Research Project: Static Analysis of Medical Device Software, updated 11 Feb. 2011.

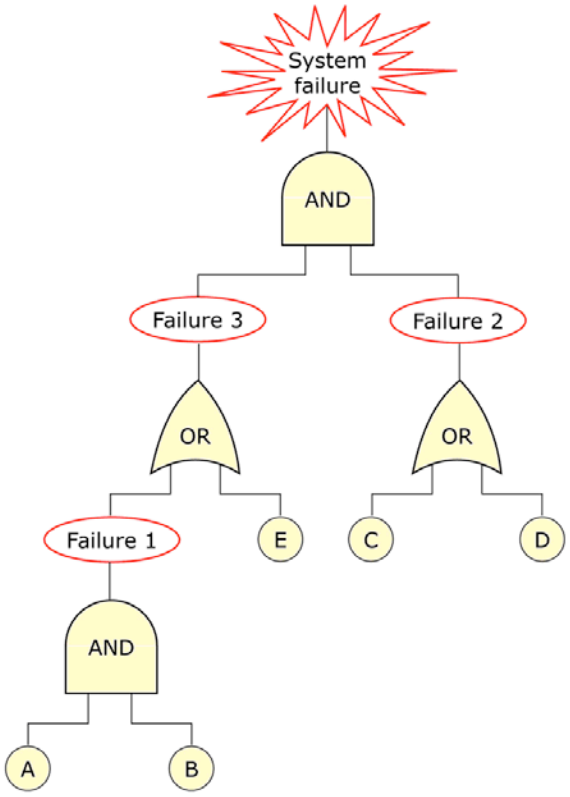

Figure 3. A very simple fault tree. Failures are numbered (1,2, etc.), while letters identify leaves (A, B, etc.).

Does the COTS software vendor use static analysis to identify potential problems in his product?	
What static analysis techniques does the vendor use? • Syntax checking against published coding standards? • Fault probability estimates? • Correctness proofs (e.g. assertions in the code)? • Symbolic execution (static analysis-hybrid)?	
What artefacts does the COTS software vendor provide to support the findings from his static analyses?	

Proven-In-Use Data

Proven-in-use data is invaluable when reviewing COTS software dependability claims, and for building claims. Anyone building a system for which

one day (even in the very distant and nebulous future) he may be required to show proof of dependability should build gathering in-use data into his business model.

Can the COTS software vendor provide proven-in-use data?	
How far back does the data go?	
How comprehensive is the data? • What is the sample size for which data is available? • Does this sample represent a small or large percentage of the vendor's runtimes? • How does the vendor gather this data?	
Does the vendor provide fault analysis results with the proven-in-use data, or just usage data?	

Design Artefacts

Design and validation artefacts are one of the key differences between SOUP and clear SOUP. If the COTS software vendor cannot provide an extensive set of artefacts, there is little reason to select his wares over open source software.

What design artefacts does the COTS software vendor provide with his software?	
Does the vendor provide: • Architectural design documents? • Detailed design documents?	
What are the test plans and methods for the COTS software, and does the vendor publish the detailed results?	
What other validation methods does the COTS software vendor use (see other sections above), and are the methods and detailed results available?	
Does the vendor maintain and make available a traceability matrix, from requirements to deliver, and is it available for scrutiny?	
What records does the vendor keep of the software life cycle, including: • Changes? • Issues and their resolutions?	

The Safety Manual

The Safety Manual is another *sine qua non* requirement. If the COTS software does not include a Safety Manual, return the product to the shelf and try another vendor.

Does the Safety Manual state the functional safety claims for the COTS software?	
Does the Safety Manual define the context and constraints for the COTS software functional safety claims? These should include the environment and the usage where the functional safety claims are valid. For example: • "This list of processor architectures is exhaustive." • "Floating point operations SHALL NOT be performed in a signal handler." • "Critical budgets are limited to the window size."	
Does the vendor provide training on the safe application of the product?	

Certified Components

Even if all the above recommendations have been followed, and the COTS software meets all the requirements for clear SOUP, there are no guarantees that approval of the final product will proceed according to plan and on schedule. Further advantage can be gained from working with a COTS software vendor that has experience with approvals, and from employing components from that have received relevant approvals.

Though agencies, such as the FDA, MHRA, Health Canada and their counterparts in other jurisdictions approve, not the components, but the entire system or device for market, components that have received certifications, such as IEC 61508 or IEC 62304 can streamline the approval process and reduce time to market.

To begin with, in order to have received certification, these components will have to have been developed in an environment with appropriate processes and quality management, they will have had to undergo the proper testing and validation, and the COTS software vendor will have all the necessary artefacts, which will in turn support the approval case for the final device. Finally, a vendor that has experience

with certifications will likely be able to offer invaluable advice and support to his customers.

Conclusion

Neither the IEC 62304 "software for medical devices" standard, nor the demands of functional safety preclude the use of COTS software in medical devices. We must exercise diligence and caution, but COTS software may be a perfectly acceptable choice, given stringent selection criteria, and appropriate and equally stringent validation of the completed medical systems and devices. In fact, if we make the fine but critical distinction between *opaque* SOUP[19] (which should be avoided) and *clear* SOUP, that is, SOUP for which source code, fault histories and long in-use histories are available, we will find that COTS software may be the optimal choice for many safety-related medical devices.

Bibliography

AAMI. Medical device software—Part 1: Guidance on the application of ISO 14971 to medical device software. Association for the Advancement of Medical Instrumentation, 2009.

Berard, B. et al. *Systems and Software Verification*. Berlin: Springer, 2001.

Birman, Kenneth P. and Thomas A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems". Cornell University. February, 1987.

Birman, Kenneth P. Building Secure and Reliable Network Applications. Greenwich: Manning, 1996.

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

FDA, *Research Project: Static Analysis of Medical Device Software*, updated 11 Feb. 2011. <www.fda.gov/MedicalDevices/ScienceandResearch/ucm243156.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.

Helminen, Atte. *Reliability estimation of safety-critical software-based systems using Bayesian networks*. Helsinki: Säteilyturvakeskus (Finnish Radiation and Nuclear Safety Authority), 2001. http://www.stuk.fi/julkaisut/tr/stuk-yto-tr178.pdf

Hobbs, Chris. "Fault Tree Analysis with Bayesian Belief Networks for Safety-Critical Software". QNX Software Systems, 2010. www.qnx.com.

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.

Holzmann, Gerard J., *The SPIN Model Checker*. Boston: Addison-Wesley, 2004.

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>

Lions, J. L. et al. *Ariane 501 Inquiry Board Report*. Paris: ESA, 1996.

NASA. Agency Risk Management Procedural Requirements (NP4 8000.4A). NASA, 16 Dec. 2008.

QNX Neutrino RTOS Safe Kernel 1.0: Safety Manual: QMS0054 1.0. QNX Software Systems, 2010. www.qnx.com.

Reason, James. *Human Error*. Cambridge: Cambridge UP, 1990.

[19] Opaque SOUP is sometimes, jokingly, called "pea SOUP" in contrast to clear SOUP, which has not yet been dubbed "broth".

About QNX Software Systems

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com