Building Functional Safety into Complex Software Systems, Part I

Chris Hobbs, Kernel Developer
QNX Software Systems
chobbs@qnx.com

Abstract

Traditionally, proofs that software systems meet standards for functional safety have depended on exhaustive testing. This method is adequate for relatively simple, deterministic systems, with single-threaded, run-to-completion processes. It is inadequate, however, for today's multi-threaded systems. The complexity of these systems precludes their being treated as deterministic systems in practice.

In Part I of this whitepaper series we discuss the limits of testing of complex software systems, and some factors that should be weighed when deciding how to build complex software systems that must meet functional safety standards. In Part II, we propose how a combination of procedural rigor, statistical testing, and design verification can be used to increase confidence in complex software systems. In subsequent papers in this series, we will explore specific strategies for building and validating functional safety in complex software systems.

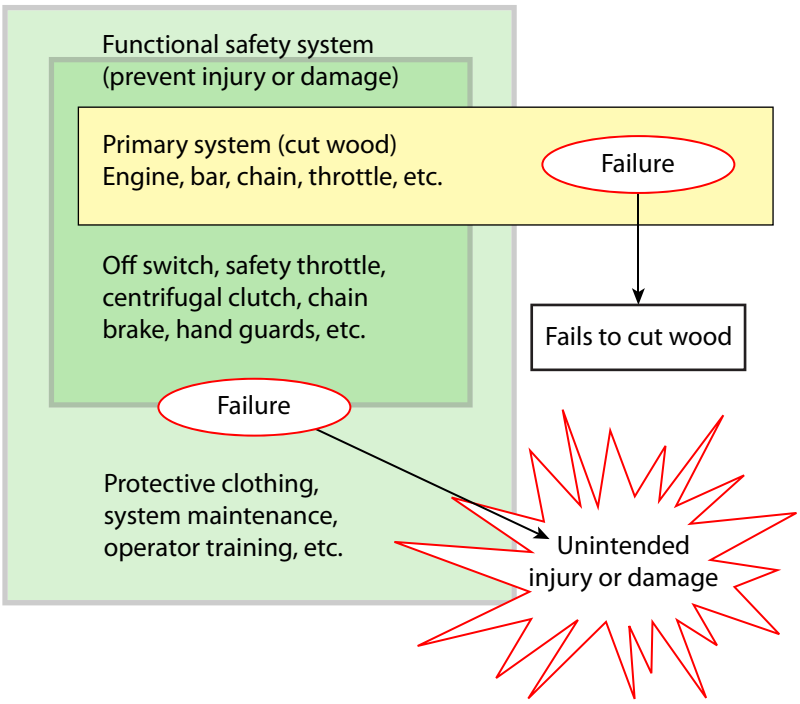

Figure 1. A chainsaw is a safety-related system. Its primary system cuts wood; its secondary, functional safety system is designed to prevent injury and damage.

Safety-Related Systems

In the context of the present discussion, we consider that a safety-related system is a system that could cause unacceptable or unforeseen injury or damage to the health of people, or damage to property or the environment, but that operates in a way that prevents it from doing so. By "unacceptable or unforeseen injury or damage" we mean any injury or damage that:

- the system is not expressly designed to cause — the injury a chainsaw causes to a tree is acceptable; injury to the person operating the saw is unacceptable
- has not been previously identified and deemed acceptable — some oil finding its way into adjacent waters is foreseen and considered acceptable practice during undersea oil extraction; deaths of rig workers, fires, and uncontrolled spills like the 2010 Deepwater Horizon oil spill is unacceptable

In the context of our discussion of functional safety, we can decompose a system into two parts, the primary system and the functional safety system. Figure 1 above presents an abstraction of the primary and functional safety systems for a chainsaw.

Primary system

The primary system performs the primary task; in the case of a chainsaw, it cuts through wood. The components of this system are whatever is required for the chainsaw to cut through wood: the engine, fuel tank, On/Off switch, bar, chain, etc.

Functional safety system

The functional safety system is the system that ensures that during its operation the safety-related system causes no unintended injury or harm; in the case of the chainsaw, it ensures that the saw doesn't cut or otherwise harm the operator or bystanders.

This system includes, not just components that are physically part of the system: the off switch, safety throttle, chain guard, etc., but also components that are not physically part of the system we normally call a chainsaw: protective boots, gloves and clothing, goggles, etc., and less concrete but nevertheless very real components such as system maintenance, operator training, operating instructions and rules (Don't drink and saw. Don't try to stop the chain with your hand!), and so on.

If its primary system fails, the chainsaw doesn't cut wood. If its secondary, functional safety system fails, the chainsaw may or may not cut wood, but it might also cut or otherwise injure the operator, or other persons, animals or property.

Note that the primary system and the functional safety system may share components. For example, in our chainsaw the On/Off switch is shared by both systems. We need it to start the saw so we can use it to cut wood; and we need it so we can instantly switch off the saw in an emergency. For precisely this reason, On/Off switches for chainsaws are designed to be easily flipped to the Off position while wearing heavy work gloves.

Functional Safety

Functional safety is the capacity of a safety-related system to function as it is expected to function. It is the continuous operation of a safety-related system performing its primary tasks while ensuring that persons, property and the environment are free from unacceptable risk or harm.

Functional Safety in Software

Software has been employed in safety-related systems for generations. It has become ubiquitous in contexts ranging from oil refineries to medical devices to automobiles to spacecraft. In every one of these implementations the software systems—like the larger systems in which they operate—have undergone rigorous examination to ensure that they meet the required levels of safety integrity[1] required for certification to standards such as IEC 61508 (electrical/electronic/programmable), IEC 62304 (medical), ISO 26262 (automotive), and the CENELEC EN 5012x series (railway transportation).

Systems are demonstrated to be functionally safe when they have been evaluated by an accredited organization, and been accorded Safety Integrity Level (SIL) certification.

Safety integrity through testing

Until relatively recently, the rigorous examination of software systems to obtain certification relied principally on process evaluation and testing. All possible states and state transitions were identified, and the system was exhaustively tested to demonstrate that at each state and state transition the software behaved as required.

This approach to demonstrating safety integrity and obtaining system certification rests on two premises. First, it assumes that, unlike hardware, software does not wear out. If a software system can once be shown to work correctly in all states and state transitions, it will always work correctly in all states and state transitions. Second, it also assumes that the software system is deterministic; that is, that the system is finite and that all its states and state transitions can be identified and, hence, tested for conformity with required behavior.

Software is not always to blame

Software has been blamed for many costly and highly publicized failures. Among the best known of these failures are the Ariane 5 launch debacle of 1996 (see below), and the Mars Global Orbiter failure in 2007.

Software is not always to blame, however. For instance, when US Airways Flight 1549 lost power in both its engines in January 2009, the flight crew was able to ditch in the Hudson River because the flight control software continued to function correctly and allowed them to control the plane.

Hardware (the engines) failed, but the 155 people on the Airbus survived. They owe their lives to the crew's cool heads and decisive actions—and to a well-designed flight control system.

Software does wear out

No, we are not suggesting that software gradually drops instructions until it becomes threadbare like an old coat, or holes appear where perfectly valid code used to be. It does not wear out with use. Unfortunately, however, in practice software does wear out in the sense that it no longer performs adequately or correctly what it was originally built to do. Without any changes being made to the code, the software may cease to behave as required, just as, without any express changes, a coat may cease to keep its wearer warm in winter because, for instance, it has changed context. Its

[1] EN 50126, for example, defines safety integrity as "the likelihood of a system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time".

owner bought and wore it in, say, London, but has since moved—and taken the coat along—to Helsinki.

Software can wear out or, perhaps more accurately, cease to perform adequately or correctly, when it changes context. Code that works perfectly on one processor might not continue to do so when we move it to another processor. For example, every processor has a long and unique list of errata, and these can affect the way software runs. Software may run correctly on one processor because of a hidden fault with that processor, then fail on a processor without that fault. The instances of software working properly for years then failing are legion, though few are as dramatic, costly or famous as the Ariane 5 incident. (See "What we learned from Ariane 5" on this page.)

The End of Deterministic Systems

More significant than the assumption that software does not wear out is the assumption that a safety-related software system is deterministic, that every state and state transition in the system can be known and tested. This assumption was largely valid for software systems in the past, and remains valid for many safety-related systems in use today.In practice, if we rely on exhaustive testing to prove that a system meets functional safety requirements, the system must be simple. For a software system, this requirement often means that the system is limited to single-threaded, run-to-completion processes. In such a system, rate monotonic scheduling, or something similar may be needed to prove that all processes meet their deadlines, and, if internal states can be pre-set, testing can demonstrate conclusively that processes do meet their deadlines.

What we learned from Ariane 5

Thirty-seven seconds after it was launched on June 4 1996, the European Space Agency's (ESA) new Ariane 5 rocket rained back to earth in pieces. This failure was rather costly: some US $370 million, and a stinging embarrassment for ESA.

It has become one of the best known instances of software that had been exhaustively tested and even field proven — in this case, more accurately, sky-proven — ceasing to function correctly though it had not been changed. What had changed was the context in which the software ran.

The acceleration of the Ariane 5 was greater than that of its predecessor, the Ariane 4, for which the Ariane 5's Inertial Reference System (SRI, Système de Référence Inertiel) had originally been designed and tested. In the new context, though the Ariane SRI itself had not changed, in practice it had worn out; it was no longer able to function as required.

Fortunately, the Ariane 5 incident did not cause any fatalities. Its importance for ensuring functional safety in software systems is far greater than its immediate cost. It provided a dramatic demonstration of the limitations of state-based testing as a means for ensuring functional safety.

This demonstration may in the long run lead indirectly to savings (even perhaps to the ESA) far greater than the US $370 million that it originally cost the ESA, because it led to increased research into other means of proving that a software system meets its functional safety requirements.

Today, these sorts of systems are being increasingly relegated to very specific tasks, such as controlling anti-lock brakes, and are being (or should be) replaced by more complex systems with multi-threaded applications. 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 states 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."[2]

Stranded in an elevator

Figures 2, 3 and 4 below[3] are state diagrams for an elevator and its door in a three storey building. They illustrate just how difficult it is to validate even a simple system through testing.

The building contains a single elevator. On each of the building's three floors there is a door so that people can step in and out of the elevator. A central elevator controller sends signals to the cage to cause it to move up or down.

To keep our example simple, we have our elevator controller ignore requests from the building occupants to come to their floor: if this were a real building the call button by the elevator might, as we've all suspected at some time, light up to make us feel good, but would have no effect on when the elevator arrived. Also to keep things simple, our controller doesn't check if the elevator doors on each floor are open or closed.

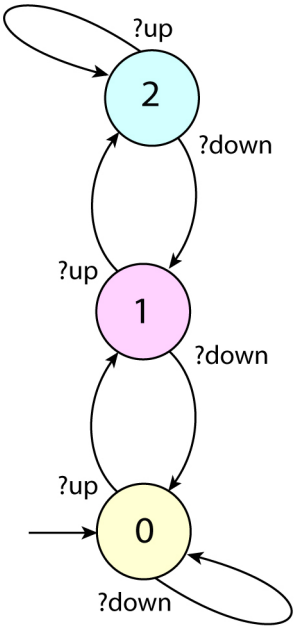

Figure 2. A simple system. The elevator can receive instructions to go up or down to the next floor.

The elevator

Figure 2 shows a state diagram for our elevator. The elevator can receive one of two possible instructions from its controller: go down `(?down)`, or go up `(?up)`.

If the elevator is at the top floor, an `?up` instruction to go up does not change its location; similarly, if it is at the bottom floor, a `?down` instruction has no effect. This system is simple enough to be exhaustively tested and shown to meet its requirements.

The doors

Figure 3 shows a state diagram for our elevator doors. This system is even simpler than the system that controls the elevator's movements. The doors can either open `(?open)` or close `(?close)`. An instruction to change to their present state, for example, `?close` when the doors are closed, has no effect.

The elevator controller

Figure 4 shows a state diagram for the elevator controller, which sends instructions to the elevator and to the elevator doors. This system is still a very simple system, but it requires more careful examination than do the elevator system and the door system. In fact, it contains a fault that may not be immediately apparent.

This system is simple enough that we can uncover the fault through testing—as long as we test the right state transitions; or through design verification—as long as we ask the right questions. A potential condition we might wish to test is that no door should open or remain open unless the elevator is at the floor with that door. This

[2] *Application Note 2: Software and EN 50128.* London: Railway Safety, 2003. p. 3.

[3] Adapted from B. Berard *et al, Systems and Software Verification.* Berlin: Springer, 2001.

condition could, in principle, be tested, but it would be tested only if we have thought of the dangerous condition. If we do not think of this condition, we cannot test for it, and someone just might find an open door and fall down the elevator shaft.

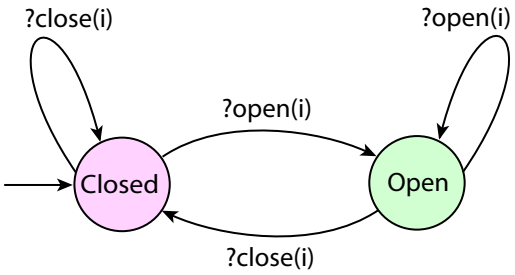

Figure 3. A simple system. The elevator doors can receive instructions to either open or close.

However, even if we design the system so that doors do not open without the elevator at the apporiate floor, the system fails to ensure that someone does not get stuck in the elevator. If, for example, we get on the elevator on the bottom floor, the controller can send us 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 n ups and downs, in much the same way that telecommunications networks drop packets that cannot be delivered after n hops.

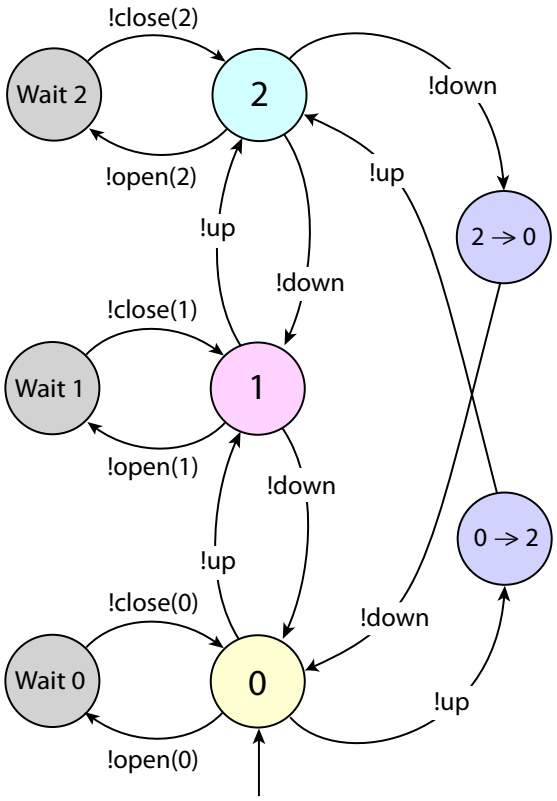

Figure 4. A simple system with a fault. There is no guarantee that the elevator door will ever open.

This simple scenario underlines one of the key challenges we face when we attempt to verify even very simple systems. Testing and design validation can only reveal the presence of faults we have anticipated. Except in the most trivial systems, such as the elevator door system shown in Figures 2 to 4, they cannot confirm the absence of faults. They can only confirm that what we test behaves in the specified way, and that the system does correctly what we asked about the system. If we want to confirm that the elevator doors will not open without the elevator being present, we can test the system and verify its design to ensure that, indeed, it protects people from falling down the elevator shaft.

However, if we forget that people can also become trapped in elevators and we do not test for this problem, or do not ask if this could happen to someone taking our elevator when we verify the design, this rather serious fault can slip unnoticed into our system. We forgot to ask if all elevator rides must end. Of course,

as a system increases in complexity, so will the number of these sorts of faults. As our elevator system shows, even in a very simple system it is easy to miss functional safety requirements that in hindsight are glaringly obvious. We cannot insist too much on the importance of a well-selected set of functional safety requirements.

Testing of complex systems

In theory, a system with multi-threaded processes is deterministic. All its states and state transitions can—again, theoretically—be identified. However, these states and state transitions are so numerous that in practice they cannot be counted, to say nothing of testing them. The number of possible states, state transitions and their effects on the system is so great that, in practice, the system might as well be infinite.

Further, even if these states, state transitions and consequences could be enumerated, their nature and complexity make it impossible to test many transitions, because it would still be impossible to force the starting points from which these transitions move.

Multi-threaded kitchen drawers

To illustrate how a complex system becomes impossible to test exhaustively in practice, we can assume that we somehow got out of our elevator and have gone shopping for furniture. In the furniture store we come across a device which will In this particular case, the device opens and closes a kitchen cutlery drawer repeatedly, recording and displaying the number of times it has performed this movement, proving that the drawer design and manufacturing are of sufficiently well designed and manufactured to last x years.

For example, at the rate of one test every two seconds, in two weeks the test would be repeated 302,400 times. On the assumption that in a real kitchen the drawer would be opened and closed an average of seven times a day, this test would demonstrate that the drawer would last 118 years in a real kitchen—somewhat longer than most modern kitchens go without some sort of makeover.serve well to illustrate our point. The device is used to demonstrate furniture quality by repeating actions that imitate how the furniture would actually be used: “use cases” in the language of software testing.

If, however, the proof must demonstrate that a complex of, say, 1000 drawers is designed to last 118 years, that there are rules stipulating that drawers may or may not be opened depending on what adjacent drawers are doing, that the drawers may decide to move around in the complex based on this adjacency in order to be able to open or close, that if it’s Tuesday blue drawers must open only half way, and red drawers must not close, and, finally, that the operation of some or all drawers depends on the precise location and state of sets of between 14 and 23 drawers, then the proof starts looking more like testing a multi-threaded software system.

The number of possible states and state transitions quickly grows to the point that the drawers may often seem to have minds of their own, and, in practice, it quickly becomes impossible to provide proof that the complex of drawers will indeed function without failure for 118 years, or even a single day. If we remember how easy it was to let an error slip into our very simple elevator controller design because we did not ask the right question (Can people become trapped in the elevator?), we must accept that our slightly more complex system with 1000 drawers will include faults.

Significantly, the Ariane 5’s SRI was not tested adequately because, among other reasons, with one of the two proposed methods of testing “accurate simulation … is

quite expensive", while with the other proposed method "simulation of failure modes is not possible with real equipment, but only with a model", and testing with a model was complex and insufficiently accurate[4]. (See also "What we learned from Ariane 5" on page 4.)

The system was not tested for the specific fault that would cause the failure, because no one asked the right question, and because the system had not been soaked under live conditions. Either of these verification procedures would have revealed the fault. We should remember, though, that every complex software system is like the Ariane 5's SRI: it contains faults no one can imagine before they cause an error or failure, and even for potential faults that have been identified, the systems are incompletely tested.

Functional Safety with SOUP

If we accept, first, that safety-related systems will increasingly require more interaction and computing power than can be provided by single-threaded, run-to-completion systems, and, second, that complex, multi-threaded software systems cannot be validated for functional safety through exhaustive testing, then it becomes essential to map out new, more comprehensive strategies for verifying that a system is functionally safe. In short, projects that develop functionally safe systems must have more in their verification plans than simply "TEST" and "TEST AGAIN".

A witch's brew of dubious ingredients

As with all software design, no strategy for designing a safety-related system is perfect, and the choice of which strategy to use depends very much on the particulars of each project. These design choices include various combinations of:

- software built in-house from scratch
- SOUP (Software Of Uncertain Provenance/Pedigree)
- software with functional safety certification

A comprehensive discussion of the merits of each approach and their various possible combinations would fill a few library shelves, and cannot be entertained here. A few comments may be helpful, however.

Software built from scratch

This approach often appears to be the obvious solution. If the entire system is designed and build from scratch, then the designer and builder control both process and product from start to finish. There are no unknown or dubious components, and control of the process as well as the final product facilitates certification, which usually includes the design and development process as well as an examination of the finished product.

If, however, we remember that "software failure rates do in fact follow the conventional bathtub curve"[5], the do-it-yourself approach to ensuring that a system meets its functional safety requirements begins to looks less attractive.

Experience has shown us that no matter how well-designed, built and verified, a system has a higher failure rate when it is newest. Everyone is familiar with the high

[4] J. L. Lions *et al.*, *Ariane 501 Inquiry Board Report*. Paris: ESA, 1996, p. 8-9.

[5] Chris Hobbs, "Protecting Applications Against Heisenbugs". QNX Software Systems, 2010. www.qnx.com.

failure rate of software when it is first exposed to the field and different usage patterns uncover latent faults. Hence, a system that incorporates components that have stood the test of time may in fact be a better choice than a new system built from scratch — even if some of these are of unknown provenance or pedigree.

To this we must add that for software built from scratch there is clearly no data on which to build a proven-in-use argument: the software has no history; and that building the software from scratch top-to-bottom—or, more accurately, bottom-to-top—is a gargantuan task beyond the capabilities and schedules of most projects.

SOUP and clear SOUP

Software vendors often make the distinction between COTS (Commercial, off-the-shelf) software, and SOUP. COTS software, they say, has a vendor standing behind it, a company that has staked its reputation—and its financial future—on this software functioning as specified, while SOUP has no one standing behind it.

This position is valid in the same way that it may be preferable to buy medication from a reputable pharmacy than from some web site that uses spam to advertise. However, in the context of functional safety, it is mostly irrelevant since for us most COTS is probably SOUP because processes, code, fault histories, and everything else required for certification may not be available to anyone outside the selling organization.

A more useful distinction is between SOUP and clear SOUP. A COTS system, such as a Microsoft Windows operating system, is opaque SOUP because, though it may have a well-documented development process, its source code and failure history are not available for public scrutiny.

In contrast, open source projects such as Apache and Linux are clear SOUP because they make their source code and fault histories freely available. Thanks to years of active service these projects' characteristics are well-known. Like in-house software, they can be scrutinized with code symbolic execution and path coverage analysis, and their long (and freely available) histories make findings from statistical analysis particularly relevant.

An inherent limitation of testing

No matter how simple the system, testing can never prove the absence of errors.

Testing can only reveal the presence of errors. If we ask the right question (devise the appropriate test) about our elevator controller, testing will show that it can trap people in the elevator. Testing cannot confirm that the controller has no other errors, however, because finding each error requires that we devise the test for that error. If we cannot image the error, we cannot test it.

Further, testing requires that we be able to generate starting states from which we can examine specific behaviors, something which we may not be able to do in a complex system.

Finally, testing typically verifies reliability: that responses are correct for each use case tested. Verification of availability: that responses are delivered at all, is only done in passing. If schedules permit, a system may be subjected to stress tests, or left to "soak", but this step is often pushed out to after the system has been deployed on the field.

Despite these attractive characteristics, open source may not be the best solution, however. The difficulty with using open source in functionally safe systems is that open source development is neither clearly defined nor well-documented. We can't know how it was coded or verified. It takes a leap of faith to assume that we know as much as we need to know about them. Add to this that SOUP or COTS 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 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. Ideally, we should work with clear SOUP made with a clear recipe that has a long and well-documented history of success in the field.

Software with functional safety certification

Functional safety certification evaluates the safety integrity level (SIL) of an entire system; when sub-systems and components are evaluated, their dependability is assessed in the context of the system in which they occur. The U.S. Food and Drug Administration, for instance, certifies medical devices as a whole — and quite rightly so. If, for example, a manufacturer changes the battery it uses in its pacemakers, someone relying on this device would likely prefer to know that it has been certified with its new power source, even if nothing else had changed.

That systems must be evaluated in their entirety to acquire functional safety certification in no way diminishes the value of using certified components in these systems. Quite the contrary. Using a component, such as an operating system kernel whose functional safety integrity level has been certified by a reputable agency, can contribute significantly to achieving certification for the entire system. There are four key technical benefits associated with using components with functional safety integrity level certifications: quality, process, documentation, and vendor knowledge.

Quality

Certification of the component confirms its quality. Its claims to dependability have been independently evaluated and found to be true.

QNX Neutrino RTOS Safe Kernel

The QNX Neutrino® RTOS Safe Kernel has been certified to IEC 61508 Safety Integrity Level 3 (SIL 3). It provides a certified platform on which application developers can implement systems that must meet the most stringent functional safety requirements.

Process

In order to be certified the component will have to have been developed and evaluated in conformance with clearly defined, comprehensive processes.

Documentation

The certified component will include user documentation detailing how to use the component in a system requiring functional safety certification.

Vendor knowledge

The vendor of the certified component has seen the component through the certification process from project inception to completion, and in many cases is more than willing to assist customers in obtaining certification of their systems.

This assistance can make the difference that determines a project's success or failure—especially for organizations that have little or no experience with functional safety certification.

These technical benefits translate directly into important business benefits, chiefly a shorter and less expensive certification process, and hence faster time to market, reduced development costs, and increased profits.

Conclusion

We have seen that the functional safety of today's multi-threaded complex software systems cannot be validated by traditional, state-based testing alone. Though these systems are deterministic in theory, due to the number of possible states and state transitions they can present, they might as well be infinite.

It is, nonetheless, not only necessary, but possible to build functionally safe complex software systems. In Part II of this paper we will explore some strategies we can use to design and build these systems, and the methods we can use to verify that they meet their functional safety requirements: procedural rigor, statistical testing, and design verification.

References

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

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

Bouissoe, Marc, and Fabrice Martin and Alain Ourghanlian. (1999) "Assessment of a Safety-Critical System Including Software: A Bayesian Belief Network for Evidence Sources". *Proceedings of the Annual Reliability and Maintainability Symposium.*

EN 50126 1999: Railway applications—The specification and demonstration of reliability, availability, maintainability and safety (incorporating corrigenda May 2006 and May 2010).

ERA Technology Ltd. (2009) "Risk Modelling using Bayesian Networks". http://www.era.co.uk

Havelund, Klaus et al. "Formal Analysis of a Space Craft Controller Using SPIN". Moffet Field: NASA Ames Research Center, n.d.

Helminen, Atte. (2001) *Reliability estimation of safety-critical software-based systems using Bayesian networks.* Helsinki: Säteilyturvakeskus (Finnish Radiation and Nuclear Safety Authority). 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, 2009.

_____. "Protecting Applications Against Heisenbugs". QNX Software Systems, 2010.

Jackson, Daniel, ed. *Software for Dependable Systems: Sufficient Evidence?* Washington: National Acadamies Press, 2007.

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

Littlewood, Bev and Peter Popov, and Lorenzo Strigini. (2001) "Modeling software design diversity—a review". *ACM Comput. Surv.*, 33(2):177-208.

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

Railway Safety: Engineering Safety Management Yellow Book 3: Application Note 2: Software and EN 50128. Issue 1.0. London: Railway Safety, 2003.

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

About QNX Software Systems

QNX Software Systems Limited, a subsidiary of Research In Motion Limited (RIM) (NASDAQ:RIMM; TSX:RIM), is a leading vendor of operating systems, development tools, and professional services for connected embedded systems. Global leaders such as Audi, Cisco, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle infotainment units, network routers, medical devices, industrial automation systems, security and defense systems, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems Limited is headquartered in Ottawa, Canada; its products are distributed in more than 100 countries worldwide. Visit www.qnx.com and facebook.com/QNXSoftwareSystems, and follow @QNX_News on Twitter. For more information on the company's automotive work, visit qnxauto.blogspot.com and follow @QNX_Auto.

www.qnx.com