BlackBerry | QNX

Ten Principles for Building Safe Embedded Software Systems

Obtaining safety certifications and pre-market approvals for safety-related systems is arduous, costly, and prone to failure. And yet such certifications and approvals are integral to the sale and market acceptance of software for a wide range of products. For example, software for a medical device must obtain FDA Class III pre-market approval (IEC 62304), a train control system must meet requirements set out in EN 50126 and EN 50657, automotive system components must meet differing ISO 26262 safety integrity levels depending on functionality, and an industrial automation system is required to be IEC 61508 SIL-rated.

Further, safe system design continues to evolve as embedded systems become more autonomous, connected, and shared. Autonomous systems pose new challenges for safety engineers. For example, autonomous cars need to be run through a multitude of simulations, thoroughly trained, and demonstrated safe. Then there is the issue of updating software: When connected systems receive updates on the fly, it can interfere with a safety system.

From these and other challenges arise new functional safety concepts, such as the Safety of the Intended Functionality (SOTIF) in ANSI/UL 4600; an awareness of the infeasibility of complete testing due to the long-tail distribution of road hazards as described by Dr. Philip Koopman; and the concept of dynamic safety cases, a through-life safety assurance approach described by NASA.

Safety must be embedded in the practices, processes, and culture of every organization building safety-critical systems. If safety-critical products are to succeed, manufacturers must look beyond strictly technical challenges to embrace the following principles for building and certifying safety-critical software systems.

01 / Create a Company-Wide Safety Culture

Without a company-wide safety culture, it is unlikely that a safe software product can be built.

A safety culture is not only a culture in which engineers are permitted to raise safety questions, but a culture in which they are encouraged by company leaders to do so. A programmer might think, "I could code this message exchange using either technique A or B, and I am not sure how to balance the better performance of one against the higher dependability of the other." In a safety culture, that engineer knows with whom to discuss that decision. A culture that encourages the programmer even to consider such a question must be nurtured.

Examples in ISO 26262-2, Road Vehicles — Functional Safety — Part 2: Management of Functional Safety, contrast poor safety culture and good safety culture to show what a successful safety culture looks like:

- In a poor safety culture, "accountability is not traceable" but in a good safety culture it is.
- "Cost and schedule take precedence" in a poor safety culture, but "safety is the highest priority" in a good safety culture.
- A dissenter is "not a team player" in a poor safety culture, but in a good safety culture "intellectual diversity is sought, valued and integrated in all processes."

Cost and schedule take precedence in a poor safety culture, but safety is the highest priority in a good safety culture.

Every phase of the software development lifecycle is affected. In a safety culture, "safety and quality issues are discovered and resolved from the earliest stage in the product lifecycle."

02 / Engage Safety Experts

Building safe systems requires safety expertise and specialized training to define what a safe system must do and to verify that it meets its safety goals.

Safe systems must be simple. And the creation of a simple system is the most difficult challenge for any engineer.

The creation of a simple system is the most difficult challenge for any engineer.

Ultimately, experts (e.g., domain experts, system architects, software designers, programmers, process specialists, verification specialists) need to set the requirements, select appropriate design patterns, and build and validate the system to ensure safety.

You need safety expertise to identify risks and correctly calculate probabilities of failure.

Engineers working on a certified product need expertise in software development, in the focus industry (e.g., rail, automotive, avionics) and in the principles of safety. Safety expertise is expensive because it is based on experience. Few university undergraduate courses cover embedded software development, and even fewer teach the elements of how to create embedded systems with sufficient dependability to achieve the foundation for a safe system.

Software design patterns and techniques have evolved significantly since the early 2000s, but many designers have not been exposed to changes, such as Safety of the Intended Functionality (SOTIF), as we describe in section 5.

Companies that lack safety expertise can engage functional safety consulting services, training, and safety-certified software from safety experts, such as BlackBerry® QNX®.

Sufficient Dependability

No system is absolutely dependable, and engineers and company management must understand what the system needs to be *sufficiently dependable*.

Without an understanding of what level of dependability is sufficient, engineers are likely to produce a system that is excessively complex, and hence fault-ridden and prone to failure.

The acceptance of sufficient dependability reduces development costs and provides the measures against which to validate safety claims.

03 / Use Targeted Processes

It is no accident that most safety standards, such as IEC 62304 and ISO 26262, cover the development process. However, the more recent goal-based standards, such as UL 4600, deliberately do not cover development procedures.

Good processes are a measurable substitute for something that is difficult to assess. For example, while you can easily measure whether a process has been followed, it is much more difficult to assess whether good-quality design and code are being produced. While no one claims that a good process guarantees a good product, a good product is unlikely to result from a poor process.

You need good processes to develop a safe system, not because they guarantee the production of a safe product, but because they provide the:

- environment within which development parameters can be assessed. For example, a good test process allows statistical claims to be made about test coverage.
- structure for documenting the safety case within the chain of evidence throughout development. Producing a safety case retrospectively is possible but expensive and regenerating lost evidence duplicates effort.

04 / Demonstrate Explicit Claims

Safety claims must explicitly state dependability levels and the limits of the claims.

Good processes aren't enough; you also need to demonstrate product-specific safety. Providing the argument and evidence that a software system meets its safety claims is part of the role of the safety case. The purpose of a high-quality process is not so much to guarantee a high-quality product as to foster an environment that encourages analysis of the evidence that supports safety claims.

At the heart of every safety case are claims like "This system will do A with level of dependability B under conditions C and, if it is unable to do A, it will move to its design safe state with probability P." These claims and their caveats are laid out in the system's safety manual so they can be incorporated into the safety case of a higher-level system.

The safety case states the system's dependability claims and provides evidence. The limits of the dependability claims are as important as the claims themselves. For example, an industrial robot may be designed to meet IEC 61508 SIL 3 requirements for continuous operation not in excess of 20 hours, at which time the system must be reset. As long as the system is not used in a robot that runs for more than 20 hours, including a good margin of error, all is well. In fact, the limit allows design and validation efforts to focus on ensuring greater dependability for 20 hours—rather than trying to extend the number of hours the system can be used and remain dependable.

Currently, BlackBerry QNX is deeply involved in standards work for the structure of safety case arguments. In particular, we are driving the addition of doubt into the safety case argument.

Dependability, Availability, and Reliability

A dependable system is a system that responds correctly when, and for as long as, it is required. That is, dependability is a combination of the system's *availability and reliability*:

- Availability: How often the system responds to requests in a timely manner
- Reliability: How often the responses are correct

05 / Expect System Failures

No system is immune to bugs, especially Heisenbugs—mysterious bugs that appear, and then disappear when we look for them. Failures occur, so build a system that recovers or moves to a safe state. Additionally, expect hazardous situations to arise due to unforeseen events.

Faults, Errors, and Failures

- Fault: A mistake in the code, which may or may not cause undesired behavior
- Error: Undesired behavior caused by a fault in the code
- Failure: An inability to perform the normal or expected function or cessation of normal or expected function caused by an uncontained error

Faults are sometimes inserted into code. Even great programmers introduce faults, and some of those faults escape detection. EN 50128 states: "There is no known way to prove the absence of faults in reasonably complex safety-related software." Although it is impossible to prove there are no faults, the company can provide evidence that the system is as safe and dependable as claimed.

"There is no known way to prove the absence of faults in reasonably complex software."

Since all systems contain faults, and those faults may lead to errors that lead to failures, a safe system has multiple lines of defense, such as:

- **Isolate safety-critical processes:** Identify safety-critical components and design the system so that these components cannot be compromised by other components.
- **Stop faults from becoming errors:** While it is best to identify and remove faults from the code, this is impractical. Beware the Heisenbug and design the system so that faults are caught and encased before they become errors in the field. In the fault -> error -> failure chain, errors often result in detectable anomalies. If these anomalies are detected, it is often possible to avoid the failure.

- **Stop errors from becoming failures:** Techniques such as replication and diversification are less suitable to software than to hardware but can be valuable if used carefully.
- **Detect and recover from failures:** In many systems it is acceptable to move to the pre-defined design safe state and leave recovery to a higher-level system, such as a human. In some systems this is not practical and either recovery or restart is needed. A crash-only model followed by a fast reset may be the best way to recover in an ill-defined environment.

Hazardous events also arise under unforeseen conditions, even without a fault. A function may behave exactly as it was designed and yet through either operator misuse, the failure of the designer to anticipate a situation, or the failure of sensors or an algorithm to accurately identify a situation, the system may behave in a hazardous way.

Autonomous systems are likely to encounter unforeseen conditions. Two newer safety standards provide a guide for systems in which the situational awareness of autonomous systems is critical to safety:

- UL 4600 "Standard for Safety for the Evaluation of Autonomous Vehicles and Other Products"
- ISO 21448 "Road Vehicles – Safety of the Intended Functionality"

Safety of the Intended Functionality (SOTIF)

Consider a scenario in which designers program an autonomous car to avoid harm to a human, even if it means striking a non-human animal, such as a dog. Here's what could happen. An autonomous car travels quickly, with a human-driven car following close behind. A child on a skateboard speeds down a hill such that she will enter the road in front of the autonomous car. The vehicle's camera captures an image of the skateboarder, and the first level of interpretation software identifies it as a child. The parallax system determines that the object is traveling at 22 km/hr (14 miles/hr).

At this point, the feasibility software rejects the identification as a child, because the program states that children do not travel so fast (unless they're on a bicycle, which this one is not). Therefore, the autonomous system classifies the child as an animal, probably a dog. Given the choice of hitting the dog or braking hard to avoid it, causing injury to the humans in the car behind, the decision is made to run over the "dog."

In this scenario, every system operates as intended, yet a child is harmed:

- The camera system detects an object, and it is correctly identified as a child.
- The parallax system determines the object's speed correctly.
- The feasibility system correctly determines that the object could not be a child—because the idea of a child on a skateboard speeding downhill hadn't been thought of during application design.
- The decision software correctly balances the tradeoff between harming a human and killing a dog and makes the intended decision.

Figure 1: Safety of the intended functionality, an example of obstacle recognition by an autonomous car

06 / Validate and Verify

Testing is insufficient to prove dependability. Other methods are required, such as formal design verification, statistical analysis, and retrospective design validation.

Testing detects failures caused by a fault, but not the fault itself. And the designer cannot fix a failure, only a fault. The tracing of failure back to the fault can often be extremely difficult. Testing is of most use in the detection and isolation of Bohrbugs, which are consistently reproducible faults. But testing is of less use when faced with Heisenbugs—faults that manifest inconsistently, with a different error on each occurrence.

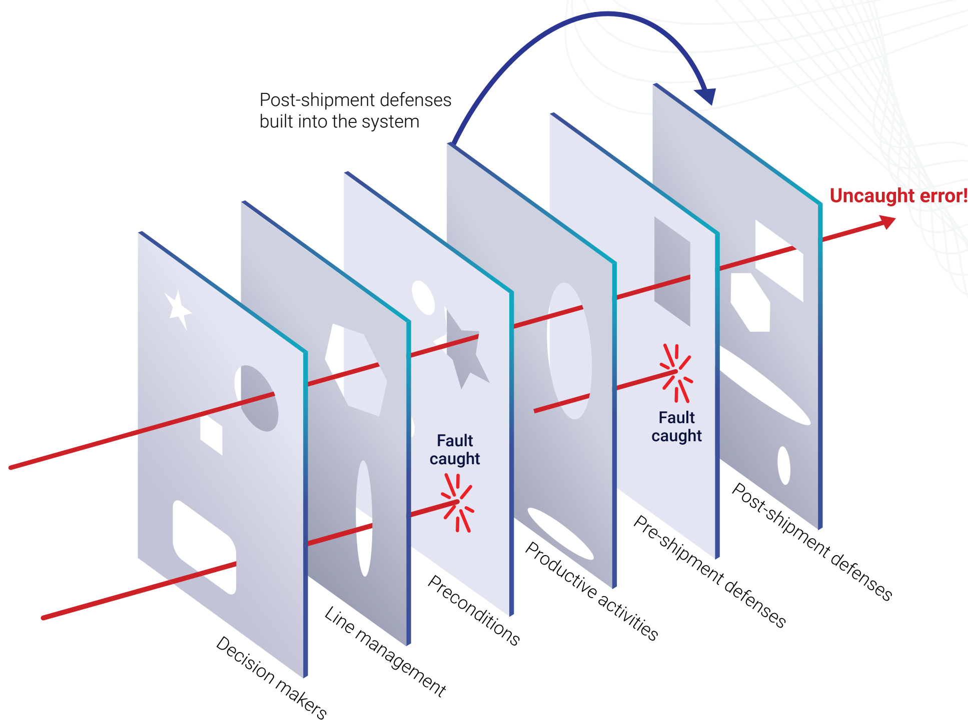

Figure 2: How faults become failures. James Reason's model (adapted) applied to software design and development.

Beyond testing, other techniques can help demonstrate that a system meets its safety claims, such as:

Technique	Description
Formal and semi-formal design verification	Verification confirms that the system works as intended by proving mathematically that it will not do what it should not and will do what it is supposed to do. Traditionally performed before the design is implemented, this form of verification can also be performed retrospectively, essentially reverse engineering the design from the implementation. Formal verification uses a formal language (mathematics) and a formal syntax, and semi-formal verification uses a formal syntax with informal semantics. Many software tools are available that surmount the complexity of the analysis, hiding much of the mathematics.
Static analysis	Invaluable for locating suspect code, static analysis includes techniques such as syntax checking against coding standards, fault probability estimation, correctness proofs against contracts in the code, and symbolic execution (static/dynamic hybrid).
Fault injection	The deliberate introduction of faults can test code designed to handle error detection and estimate the number of remaining faults. As with the analysis of the results of random tests, the results of fault injections require careful statistical analysis.
Proven-in-use and prior-use data	Essential for building dependability claims, in-use hours and failures resulting from use should be gathered throughout the product life cycle, because the larger the sample size, the greater the confidence in the claims.

Validation uncovers situations in which the software and the operator behave as planned and yet still create a dangerous situation. However, in extremely complex, variable environments—such encountered by autonomous cars—validation is impractical, even with a simulator. Dr. Philip Koopman explores the "long tail" effect of hazardous events that no human driver (or autonomous car) is likely to meet during a lifetime. And yet in almost all situations, a human driver knows how to respond, but an autonomous car may not.

A self-driving car is unlikely to ever encounter a person dressed in a chicken costume standing in the road—but what if it does? How could that (or similar) scenarios be included in a testing plan?

Figure 3: Example of a hazardous event that no human driver (or autonomous car) is likely to meet during a lifetime

Koopman estimates that a trillion miles of road testing might be required to encounter enough long-tailed road hazards before you could validate that an autonomous car is safe.

07 / Provide Evidence for COTS and SOUP

It is permissible to use commercial off-the-shelf (COTS) components, and even software of unknown provenance (SOUP), if these components come with sufficient evidence or are incorporated in such a way as to support the overall system's safety case. Some vendors have a well-entrenched safety culture and can provide this documentation.

Building everything from scratch is not the best way to build a safe software system because that entails more risk. Because you require specialized knowledge to build the OS, network stacks and databases, a COTS solution may have the advantage of tens of millions of hours of in-use history. That said, COTS software is usually SOUP and should, therefore, be treated with caution.

Both IEC 61508 for industrial systems and IEC 62304 for medical systems assume that SOUP is used.[1] EN 50128 for rail assumes the same and stipulates that if COTS software is used in systems requiring SIL 3 or SIL 4, "a strategy shall be defined to detect failures of the COTS software and to protect the system from these failures."[2]

Sufficient documented evidence must be available to quantify the effect of the SOUP on the safety goals of the overall system.

This evidence includes proven-in-use data, fault histories, and other historical data. The vendor should also make source code and verification plans available, so the system integrator can scrutinize the software with static code analysis tools. If source code is not available, binary code can be inspected with a binary scanning and software composition analysis tool, such as BlackBerry® Jarvis™.

The vendor should also detail the processes used to build the software, or provide a statement from an external auditor that confirms that the processes used in the design and validation of the COTS software were suitable for the safety and regulatory requirements of the device in which the software is used.

1 EC 61508-4, 3.2.8 and IEC 62304, 5.1.1

2 EN 50128:2001, clause 9.4.5

08 / Select Certified Components

Components with safety certifications, such as a pre-certified OS, can speed development and validation and facilitate the approval of safety claims.

Pre-certified components provide an advantage. Regulatory agencies generally certify or approve the entire system or device for market, not the components (and a jurisdiction could make the use of pre-certified components mandatory). Safety certification is simplest when you are using commercial off-the-shelf (COTS) software that is pre-certified for safety by an external auditing firm.

For a component to receive certification:

- It must be developed in an environment with appropriate processes and quality management.
- It must undergo the proper verification.
- The vendor must provide all the necessary artifacts, which in turn support the approval case for the final device.

09 / Befriend the External Assessor

The auditors are a part of your team. Engage them early on.

Many functional safety standards allow participants to self-assess a product's compliance. In tightly regulated industries—such as avionics, industrial automation, rail and power generation—an external auditor may be a mandatory requirement. Using a third-party auditor provides a higher level of confidence in a product's suitability for use in a safety-critical system.

The earlier the auditors are engaged, the less there is to revise, and the more efficient the development cycle.

Auditors are a valuable part of the manufacturer's team. They understand how to establish good processes to obtain the certifications, and they can help structure the safety case.

It is useful for you to explore the proposed structure of the safety case argument with the auditor before adding evidence. For example, if a notation such as goal structuring notation (GSN) or a Bayesian belief network (BBN) is used to separate the structure of the argument from the evidence, it is helpful to ask the auditor: "If we present the evidence for this argument, would you be satisfied?" Such a proactive approach reduces the likelihood of a surprise during the audit.

10 / Continue to Ensure Safety After Deployment

Responsibility for a safe system does not end when the product is released. It continues until the last device and the last system retire.

You may introduce or discover faults, or the environment in which the system operates may change long after a system enters the market. To maintain safety, the processes used to ensure that software meets its safety goals must span the entire life cycle of the system. In safety-critical industries this can result in an over-the-air software update or a product recall. Automakers recalled more than 6.3 million vehicles for software-related issues in the US in 2016[3].

Automakers recalled more than 6.3 million vehicles for software-related issues in the US in 2016.

Not so long ago, safety critical devices were locked in a box. Any modification occurred in strictly controlled environments during maintenance.

Now, an over-the-air software update can be sent to a car during the morning commute, and the upgrade completed before the driver heads home. The vendor must ensure the update does not compromise the safe operation of the car.

Sometimes a system may become dangerous much later. For example, the Boeing 737 MCAS system was incorrectly classified and didn't receive the attention it should have during certification. The software system led to a dangerous situation long after certification and shipment.

In response to changing situations, dynamic safety cases are an area of research. At this time, a safety case is created before the product ships and is effectively frozen in time. However, a product with a long lifecycle is likely to meet conditions in the field that were not foreseen by the safety case. A dynamic safety case is updated in the field when a certain situation is met.

For example, the safety case for a drone may assume there will never be more than 10 aircraft within a radius of 20 miles. One day the drone encounters 11 aircraft. Is the drone safe or not? With a formal, mathematically expressed safety case argument, calculations run on the fly could tell. If the result shows it is still adequately safe, the drone flies on. If not, it lands immediately.

3 Strategy Analytics, 2019, p. 4, Automotive OTA Updates: 2019 Market Status and Solution Providers.

Conclusion

A product development culture focused on safety can't guarantee that the software will meet its dependability requirements, nor is it certain the system will receive the necessary safety certifications or pre-market approvals. However, a product developed and validated in a culture in which everyone understands these ten principles has a far better likelihood of success than an organization for which safety is an afterthought. Plus, the resulting system will likely cost less to develop, validate, and maintain.

Get Help with Your Next Safety Project

BlackBerry QNX Safety Services complement and enhance your company's strengths in functional safety for embedded systems. We can help you hit your production deadlines and ensure your safety-critical products meet applicable industry standards. We offer training, workshops, consulting and custom safety software development. We can help you reduce risk and streamline the development of your safety-certified products. Learn more about BlackBerry QNX Safety Services.

Streamline Certification with Pre-certified Software

Reduce the time and effort involved in achieving safety certifications and compliance with embedded software solutions from BlackBerry QNX. We help you streamline the process of certifying your products by pre-certifying our real-time operating system and hypervisor to the highest automotive, industrial and medical standards and compliance. Our pre-certified microkernel foundation, and C and C++ toolchains qualified to ISO 26262, IEC 61508 TCL3 and T3 requirements help you reach your goals much faster. Learn more about BlackBerry QNX certified software.

About BlackBerry QNX

BlackBerry QNX is a trusted supplier of safe and secure operating systems, hypervisors, frameworks and development tools, and provides expert support and services for building the world's most critical embedded systems.
The company's technology is trusted in more than 195 million vehicles and is deployed in embedded systems around the world, across a range of industries including automotive, medical devices, industrial controls, transportation, heavy machinery and robotics. Founded in 1980, BlackBerry QNX is headquartered in Ottawa, Canada and was acquired by BlackBerry in 2010.

© 2021 BlackBerry Limited. All rights reserved. QNX, Momentics, Neutrino, and Jarvis are trademarks of BlackBerry Limited, which are registered and/or used in certain jurisdictions, and used under license by BlackBerry QNX. All other trademarks belong to their respective owners.