Ten truths about building safe embedded software systems

Yi Zheng, Product Manager, Safe and Secure Systems
Chris Hobbs, Senior Developer, Safe Systems
yzheng@qnx.com,chobbs@qnx.com

Obtaining safety certifications and pre-market approvals for safety-related systems and the larger systems, devices, components, machinery, and vehicles in which they reside is an arduous and costly undertaking.

Whether the software is an IEC 62304 medical device that must obtain FDA Class III pre-market approval, an embedded train control system that must meet requirements set out in the EN 5012x series, an automotive system with different components requiring different ISO 26262 automotive safety integrity level (ASIL) certifications, or indeed any IEC 61508 SIL-rated software system, certifications and approvals must be an integral part of the project.

If these projects are to be successful, manufacturers must look beyond the strictly technical challenges, and focus also on the environment and culture needed to develop safe software systems. Specifically, they should consider ten fundamental (but often ignored) truths about building and obtaining certifications and approvals for these software systems.

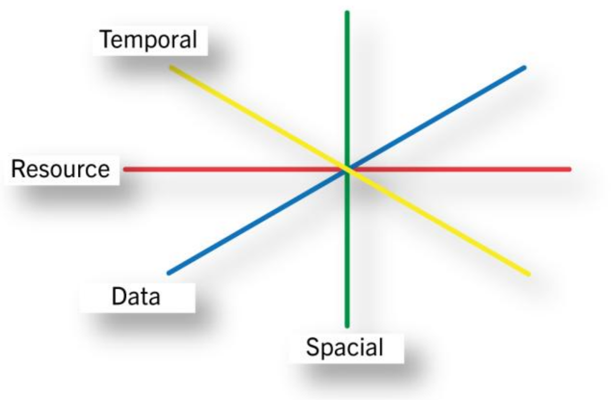


Figure 1. Planning for failure: components in a software system can be isolated from each other along multiple axes to help prevent faults from percolating across the entire system. See "System failures" below.

In short, safety should be no accident. It must be embedded in the practices, processes and culture of every organization building safety-related systems.

1. A 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 questions related to safety, but a culture in which they are encouraged to think of each decision in that light. A programmer might think, "I could code this message exchange using technique A or B and I am not sure how to balance the better performance of A against the higher dependability of B" and know with whom that decision should be discussed. The culture that encourages the programmer even to consider the question must be nurtured.

2. Experts

Safety requires professionals. It takes specialized training and experience to define what a safe system must do and to verify that it meets its safety requirements.

Safe systems must be simple. And creating a simple system is the hardest challenge for any engineer.

Ultimately, it is the relevant experts (domain experts, system architects, software designers, programmers, process specialists, verification specialists, etc.) who determine the requirements, select appropriate design patterns, and build and validate the system.

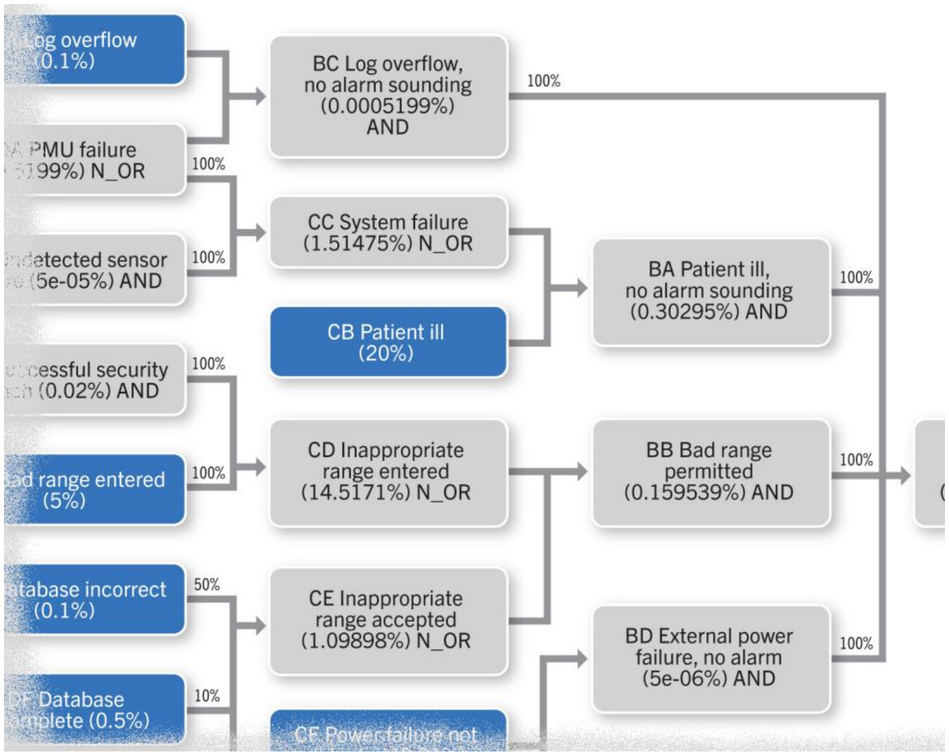

Log overflow (0.1%)
PMU failure 99%) N_OR
etected sensor (5e-05%) AND
cessful security (0.02%) AND
d range entered (5%)
tabase incorrect (0.1%)
Database mplete (0.5%)
BC Log overflow, no alarm sounding (0.0005199%) AND
CC System failure (1.51475%) N_OR
CB Patient ill (20%)
CD Inappropriate range entered (14.5171%) N_OR
CE Inappropriate range accepted (1.09898%) N_OR
CF Powerfailure not
BA Patient ill, no alarm sounding (0.30295%) AND
BB Bad range permitted (0.159539%) AND
BD External power failure, no alarm (5e-06%) AND
100%
100%
100%
100%
100%
50%
10%
100%
100%
100%
100%

Figure 2. Detail from a diagram showing the probability of failure per hour for a medical monitoring device reference design. Great expertise is required to identify risks and correctly calculate probabilities of failure.

Such expertise is expensive because it must be based on experience rather than training: few university undergraduate courses in computer engineering cover embedded software development, and even fewer teach the elements of creating embedded systems with sufficient dependability (see sidebar). Software design patterns and techniques have moved significantly since the mid-1990s but many designers have not been exposed to these changes.

3. Processes

It is no accident that standards such as IEC 62304 are about processes. Without good processes we will never be able to demonstrate that a system meets its safety requirements.

Good processes are a measurable proxy for something that is largely unmeasurable. It is relatively easy to 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 good product, it is generally recognised that good product is unlikely to result from a poor process.

Good processes are need to develop a safe system, not because these guarantee the production of a safe product but because:

1. They provide the environment within which development parameters can be assessed. For example, having a good test process allows statistical claims to be made about test coverage. Without the process, this would be impossible.
2. They provide the structure within which the chain of evidence in the safety case is preserved. Retrospectively producing a safety case is possible but expensive and would almost certainly require the re-generation of evidence that existed during the project development but which was not preserved.

4. Explicit claims

Safety claims must explicitly state dependability levels, and the limits within which these levels are claimed.

The FDA states the case very well: “indirect process data showing that design and production practices are sound”[1] is not adequate to demonstrate that software is safe, and “device assurance practices […] focused on demonstrating product-specific device safety” are also required. In short, process is not enough.

This demonstration that a software system meets its safety requirements, is included in the safety case for the product. It reflects the observation above that the purpose of a high quality

Sufficient dependability

No system is absolutely dependable, and we must understand what our system needs in order to be *sufficiently* dependable.

Accepting sufficient dependability reduces development cost and gives us the measures against which we can validate our safety claims.

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

[1] Charles B. Weinstock and John B. Goodenough, “Towards an Assurance Case Practice for Medical Devices”, Carnegie Mellon University, Software Engineering Institute, October 2009, p. 1. <http://www.sei.cmu.edu>

process is, not so much to guarantee a high quality product, but specifically to provide the environment within which evidence supporting safety claims for the product can be assessed.

Every safety case has at its heart claims of the sort “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.” This claim with its attendant caveats are laid out in the system's safety manual so that they can be incorporated into the safety case of a higher-level system.

A system’s *dependability* is its ability to respond correctly to events in a timely manner, for as long as required; that is, it is a combination of the system’s *availability* (how often the system responds to requests in a timely manner), and its *reliability* (how often these responses are correct). In other words, a dependable system is a system that responds when it is required in the time required, and responds correctly.

The safety case states the system’s dependability claims and provides the evidence that it meets these claims. The limits of the dependability claims are as important as the claims themselves.

For example, a flight control system may be designed to meet IEC 61508 SIL3 requirements for continuous operation not exceeding 20 hours, at which time the system must be reset (rejuvenated). As long as the system in not used in an aircraft that does not fly more than 20 hours, including a good margin of error, this limit will pose no inconvenience. In fact, it allows design and validation efforts to focus on ensuring greater dependability for 20 hours rather than on extending the number of hours the system can be used and remain dependable.

5. System failures

No system is immune to bugs, especially Heisenbugs[2]—mysterious bugs that “appear”, then “disappear” when we look for them. Failures will occur: build a system that will recover or move to its design safe state.

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	A system failure caused by an uncontained error.

Table 1. Faults, errors, and failures

EN 50128, for instance, explicitly states what is known to anyone who has had to design or validate a safety-related software system: “There is no known way to prove the absence of faults in reasonably complex safety-related software”[3]. In other words, “When we build a safe system, we cannot

[2] See Chris Hobbs. “Protecting Applications Against Heisenbugs”, <www.qnx.com/download/feature.html?programid=21289>

[3] BS EN 50128:2001 (incorporating corrigendum), May 2010, Introduction, p. 5.

prove that the system contains no faults"; we can only "provide evidence to support our claims that our system will be as dependable as we say it is."[4]

Accepting that all systems will contain faults, and that faults may lead to failures, a safe system must include multiple lines of defense:

Isolation of safety-critical processes—identify safety-critical components, and design so that they cannot be compromised by other components.

Prevention of faults becoming errors—while the ideal solution is to identify and remove faults from the code, this is impractical. Beware the Heisenbug, and design so that faults are caught and encapsulated before they become errors in the field.

Prevention of errors becoming failures—techniques such as replication and diversification are less suitable to software than to hardware but can still be valuable if used carefully.

Detection and recovery 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 (for instance, a human). In some systems this is not practical and either recovery or restart will be needed. In general, the crash-only model followed by a fast reset may be preferred to an attempt to recover in an ill-defined environment.

6. Validation

Testing is insufficient to prove dependability. Other methods are required: formal design, statistical analysis, retrospective design validation, etc.

Testing can indirectly detect faults in the design or implementation by uncovering the errors and failures that they can cause. Testing is of primary importance in detecting and isolating Bohrbugs—solid, reproducible bugs that remain unchanged even when a debugger is applied—but is of less use when faced with Heisenbugs because the same fault manifests as different errors each time it occurs.

[4] Chris Hobbs, "The Limits of Testing in Safe Systems", *Electronic Design*, 11 Nov. 2011. <electronicdesign.com/article/embedded/the-limits-of-testing-in-safe-systems>

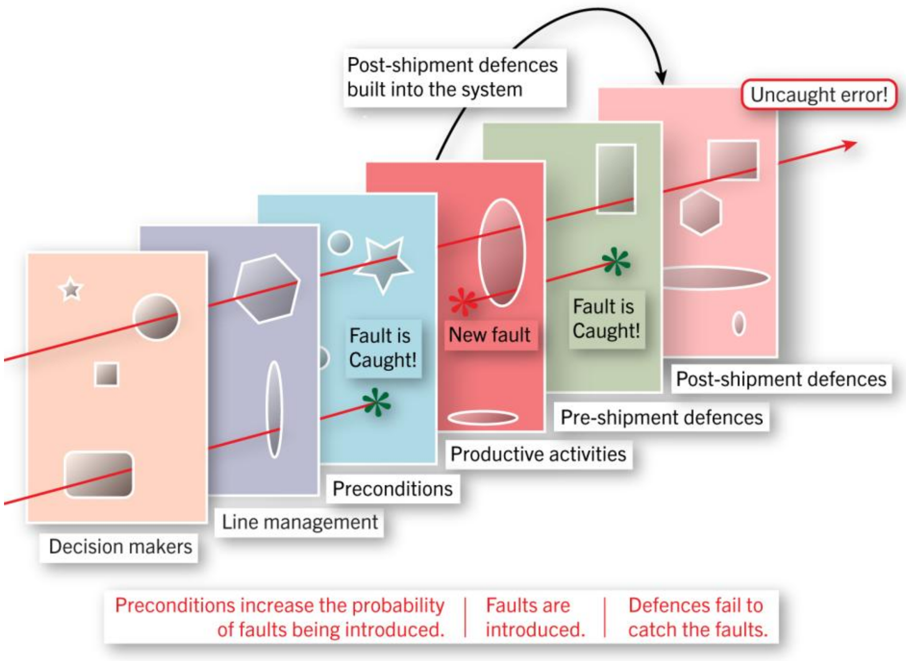


Figure 3. James Reason's model (adapted) of how faults become failures applied to software design and development.

To demonstrate that a system meets its safety claims, we must use testing as just one of many techniques that include:

Static analysis—recommended by agencies such as the FDA and invaluable for locating suspect code.[5] It can include syntax checking against coding standards, fault probability estimation, correctness proofs against assertions in the code, and symbolic execution (static/dynamic hybrid).

Proven-in-use and prior-use data—essential for building dependability claims, the in-use hours and failures resulting from this use should be gathered throughout the product lifecycle: the larger the sample size, the greater the confidence we can place in our claims.

Fault injection—deliberately introducing faults can both test code designed to handle error detection and help estimate the number of remaining faults. As with the analysis of random tests, the results of fault injections require careful statistical analysis.

Formal and semi-formal design verification—traditionally done before implementation; can also be performed retrospectively.

7. COTS and SOUP

It is permissible to use COTS, and even SOUP, if these components come with sufficient evidence to support the overall system's safety case.

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

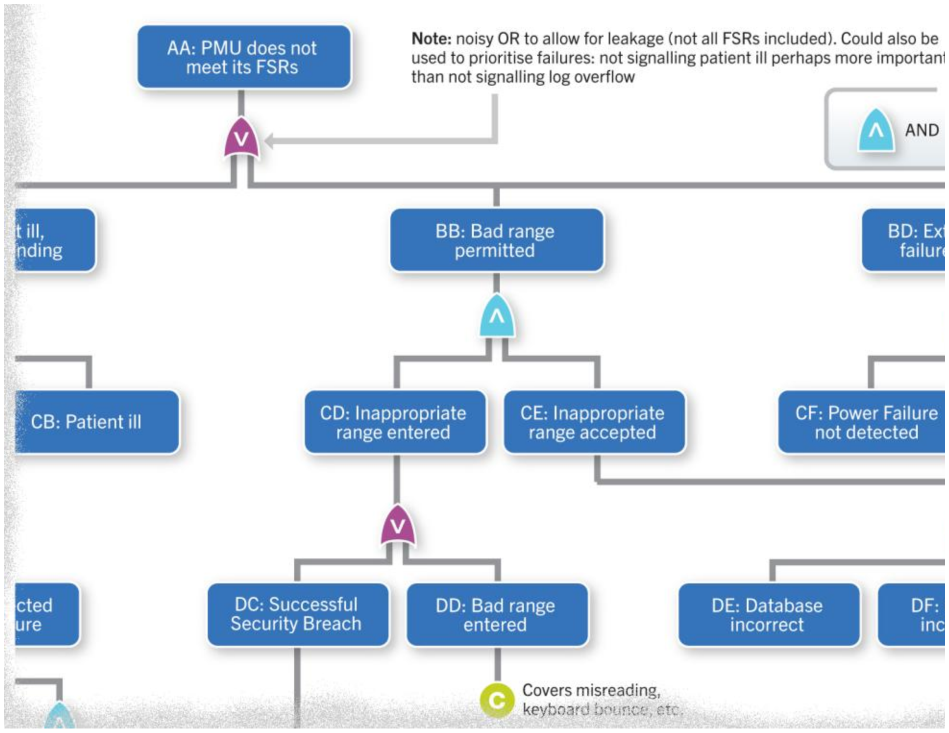
The best way to build a safe software system is usually *not* to build everything oneself as that will entail more risk than building a system with selected COTS (commercial off-the-shelf) components. Building OSs, communications stacks, and databases requires specialized knowledge and the COTS equivalent may have the advantage of tens of millions of hours of in-use history.

That said, COTS software is usually SOUP (software of uncertain provenance) as far as the developer of the medical device is concerned, and should therefore be treated with appropriate caution.

Both IEC 61508 and IEC 62304 assume that SOUP will be used.[6] EN 50128 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"[7] The trick is to ensure that sufficient documented evidence is available to quantify the implications of the SOUP for the system's safety requirements.

This evidence will include proven-in-use data, fault histories, and other historical data. We should request the source code and test plans, so we can scrutinize the software with static code analysis tools.

The vendor should also make available the detailed processes used to build the software, or a statement from an external auditor that confirms that the processes used when designing and validating the COTS software were suitable for the safety and regulatory requirements of the device in which this software will be used.


AA: PMU does not meet its FSRs
Note: noisy OR to allow for leakage (not all FSRs included). Could also be used to prioritise failures: not signalling patient ill perhaps more important than not signalling log overflow
V
AND
t ill, nding
BB: Bad range permitted
BD: Ext failur
CB: Patient ill
CD: Inappropriate range entered
CE: Inappropriate range accepted
CF: Power Failure not detected
cted ure
DC: Successful Security Breach
DD: Bad range entered
DE: Database incorrect
DF: inc
C
Covers misreading, keyboard bounce, etc.

[6] IEC 61508-4, 3.2.8., IEC 62304, 5.1.1.

[7] *Ibid.*, Clause 9.4.5.

Figure 4. Detail from a system-level fault tree for a medical monitoring device. The fault tree uses a Bayesian network, and can be seamlessly integrated into a safety case, if the case is also prepared using Bayesian techniques.

8. Certified components and their vendors

Components with safety certifications, such as an OS certified to IEC 61508, can speed development and validation, and facilitate approvals.

If COTS is used, advantage can be gained by employing components that have received relevant approvals. Regulatory agencies may approve, not the components but the entire system or device for market. (This is certainly the case with the FDA, MHRA, Health Canada and their counterparts in other jurisdictions). Nonetheless, components that have received certifications, such as IEC 61508 can streamline the approval process and reduce time to market.

In order to receive certification, a) these components must be developed in an environment with appropriate processes and quality management, b) they must undergo the proper testing and validation, and c) the COTS software vendor must provide all the necessary artifacts, which in turn support the approval case for the final device.

9. Auditors

The auditors are our friends. Engage them early on.

In the world of safe software development, certification auditors are our friends. They understand how we need to establish our processes to obtain the certifications, and they can help us structure our safety case. The earlier we bring the auditors in to help us, the less we'll have to revise, and the more efficient our development cycle will be.

It is particularly useful to explore the proposed structure of the safety case argument with the auditor before evidence has been added to it. If a notation such as GSN or BBN is used to express the argument, clearly separating the structure of the argument from the evidence, we can ask the auditor: "If we present the evidence for this argument, would you be satisfied?" This reduces the chances of surprise during an audit.

10. It doesn't end with the product release

Our responsibility for a safe system does not end when the product is released. It continues until the last device and the last system are retired.

The numbers below concern medical devices and are a little dated, but they are eloquent: updates to software can compromise its integrity:

> 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.[8]

In other words, the faults were introduced *after* the devices had gone to market. Hence, the processes we use to ensure that our software meets its safety requirements must encompass the entire lifecycle of the software, including fixes and updates.

Conclusion

A product development culture in which safety is fundamental in no way guarantees that software will meet its dependability requirements, much less receive the indispensable certifications and pre-market approvals. However, a product developed and validated in a culture in which everyone from the senior management to the technical editors reviewing the safety manual understands just the ten truths we have noted above has a far better chance of being successful than one for which safety was an afterthought—and it will likely cost a lot less to develop, validate, and maintain.