

# Perspectives on Safety-Critical Software

P.V. Bhansali, PhD

Boeing Commercial Airplane Group (BCAG)

**S**oftware is being increasingly used in safety-critical applications. These include commercial aviation, medical equipment, industrial process control, nuclear power plants, automobiles, and consumer products. Software by itself is harmless. It does not explode or electrocute its users. However, it inherits the system hazards once the system functions are allocated to software. These functions include monitoring, control, or display of safety-critical functions.

Developing software for such systems is both challenging and expensive. These challenges arise during requirements, implementation, verification, and maintenance of software. Each is described in some detail below.

A significant number of errors in software can be traced back to requirements being incomplete or incorrect. The commercial aviation community has recognized this trend and has “redefined” the terms validation and verification. Validation is the process of ensuring that the requirements are correct and complete. (Are we building the “right house?”) Unless we know what the requirements really are, it is impossible to build a system that will function correctly and will be safe. From a practical viewpoint, this is essential since we need to come up with the expected answers when verifying software and system. Verification, on the other hand, is the process of ensuring that the implementation satisfies the requirements. (Are we building the “house right?”) This change in definitions have major ramifications on the way systems are build. We no longer talk in terms of software validation or system validation (it’s too late); only requirements validation. By the same token, we can only refer to software verification and system verification i.e. verification only applies to validated requirements.

Validation of requirements can be accomplished by four process: traceability, reviews and analysis, simulation, and past experience. BCAG spends a significant amount of resources for validating requirements, especially if they are new and safety-critical. For example, the 777 fly-by-wire technology was new to BCAG, and the control laws went through rigorous validation in a 757 **testbed**.

Validation of requirements is expensive but it is also money well-spent. Identifying and eliminating errors early in the life-cycle is cheaper and reduces “risks” later in the life-cycle.

Assuming that the requirements are sufficiently validated, implementing these requirements could be error-prone. The design of complex systems usually involves “derived” requirements in software. If these derived requirements are not validated, the final product may not provide the functionality and safety specified during the system requirements phase. **Examples** of derived requirements are the scheduling of various software processes, using finite precision arithmetic, and timing constraints.

Besides these design aspects, the choice of programming language for safety-critical applications is a source of constant debate. At BCAG, we went through extensive evaluation of Ada during the 1986-1988 period. We standardized on Ada mainly because it was felt that it would reduce the cost. The 777 airplane (developed during 1990-1995) has 70 percent of its software coded in Ada. Some projects suffered because they used complex language features such as tasking and compilers were not sufficiently matured during the initial part of the program. Fortunately, these were not safety-critical systems and the “risk” was only on cost and schedule. Based on these experiences, the BCAG standard was revised (1997) and Ada is required for only Levels A and B (high criticality) in the form of a subset. The “subset” is currently being investigated for Ada 9.5 which is more complex than the original Ada 83. Prof. Tony **Hoare** is right! “Language complexity is here to stay and the only way to redeem ourselves is by identifying a small, safe subset.”

Verification consumes almost 20-50 percent of the resources during the development of safety-critical software. The cost of verifying a Level A software is almost 5 times that of Level D software. Verification is accomplished by 4 techniques: traceability, reviews, analysis, and testing. Level A software requires lot more testing and analysis compared to Level D software.

New techniques for verification are being added assuming that the existing techniques are inadequate.

Some techniques that are suitable for hardware analysis are proposed for software analysis e.g. software FMEA (failure modes and effects analysis.) Others that are suitable for system analysis are also proposed for software analysis e.g. software FTA (fault tree analysis.) This kind of “invention” has caused a lot of confusion in the industry. Also, current research appears to be preoccupied with code-based testing rather than requirements driven testing.

Maintenance of safety-critical software is probably the most challenging and expensive phase of the life-cycle. The challenge arises because it is usually boring and also occurs over a very long period of time. The development of an airplane usually takes about 5 years whereas the life of a typical airplane is 25-30 years. My Chief Engineer enlightened me the other day by pointing out that the 777 software will be maintained by people who are not even born as yet (or conceived, as yet)! This brings out the importance of documentation. Documentation of safety-critical systems is still an art. Quite frequently, a lot of nitty-gritty details are documented, whereas the basic assumptions made or the “big picture” are not sufficiently well documented.

Another aspect of maintenance which makes it challenging and expensive is the amount of reverification required. On several projects I have worked, we have taken the brute-force approach of retesting the whole software. Partitioning of software is one way to minimize this activity but this requires careful thinking during the initial design phase.

Several safety-critical software standards exist or are emerging. The trend is very disturbing. This is because these new standards require the industry to do “more,” and more is considered “better.” This approach is very dangerous since every project has finite resources in terms of cost and schedule. This tends to put pressure on doing some aspects of software development and verification more rigorously at the expense of diluting other, possibly more important activity.

The research community faces significant challenges. It needs to “optimize” the verification

approaches required for safety-critical software. For example, what combination of testing and analysis would be adequate for a certain level of criticality? Two standards such as DO-178B used by commercial airplanes and MoD00-55,56 used by UK Ministry of Defence emphasize different amounts of testing and analysis. The former emphasizes dynamic testing whereas the latter emphasizes static analysis. The research community needs to “quantify” this optimal solution in terms of cost and effectiveness by gathering field data. This is an arduous task but is crucial for establishing a “universal” safety standard.

The safety-critical software community has other challenges that are specific to their applications. These challenges arise because most of these systems are real-time, embedded in nature requiring special analysis techniques. Very little progress has been made and the only weapons known at the present time are “simplicity and isolation.” This raises the basic question: How do we quantify simplicity vis-a-vis complexity?

I am skeptical that the approaches being currently investigated will strengthen safety-critical standards. These include formal methods, software reliability modeling, and code structure-based testing. This skepticism arises from the fact that most serious errors occur during the requirements phase and at hardware-software interfaces. We need better tools and techniques for requirements validation and hardware-software verification.

In summary, there is a considerable amount of work the research community has ahead of it. It was extremely gratifying for me personally when Prof. Tony Hoare complemented the practitioners during his keynote address at the ICSE meeting in Berlin, 1996. He felt that the practitioners had used their “traditional” existing tools wisely to produce safety-critical software. However, Prof. Hoare is only half right! We know how to build safety-critical software but we don’t know how to build it economically. We need to constantly find ways of economically producing software without compromising safety.