Position statement on Digital I&C Software Reliability

NRC Meeting, February 1, 2011

Dr. Gerard J. Holzmann
Laboratory for Reliable Software
Jet Propulsion Laboratory / California Institute of Technology
Pasadena, CA 91109

Background

Software controls are being included in virtually every type of system build today, including those that are safety-critical. This includes engine-controls in cars, flight-controls in commercial airplanes, and the standard operation of an increasing number of medical devices. In many of these cases the size and complexity of the software controls is growing rapidly.

The growing size of software is in part motivated by an increasing desire for expanded functionality, as well as increased flexibility in operation and maintenance. But the expanded functionality can come at a price. There are currently *no* techniques that can provide strict *guarantees* on the reliability of complex software systems. By careful design, development, testing and verification one can significantly reduce the probability of software failure, but at present there are no known techniques that can provably eliminate the possibility of failure.

Systems View

Software controls, no matter how important, generally define only some of the components in a system. Like any other component (e.g., a bolt or a valve), a software component is not infallible: it can and will occasionally fail. But, the simple fact that the individual components that we use to build larger systems are not perfect does not imply that systems as a whole cannot be reliable. In many engineering disciplines we have learned to construct reliable systems from unreliable parts: it is why our bridges and skyscrapers do not routinely fall over and it is why NASA is able to remotely operate spacecraft even decades after they are launched. How one builds redundancy into software, though, is fundamentally different from how it is traditionally done in hardware. Clearly, duplicating a faulty piece of software does not make it any more reliable. Successful methods are based on the use of self-checking code, strict compartmentalization (software *modularity*), and design diversity (*defense in depth*).

Failure in Complex Systems

We have studied the types of software failures that occur in spacecraft over a roughly forty-year history of the use of software controls on spacecraft used in deep space missions. Based on this, a number of key observations can be made.

- Software triggered failures often follow a common pattern and have relatively few root-causes. This is good news, because it means that our software design and development practices can be adjusted to avoid the known vulnerabilities. This motivates the adoption of targeted coding standards focused on risk-reduction (remarkably many coding standards today do not have this as their primary focus). This can be combined with the use of strong state-of-the-art static source code analysis techniques[1] to verify compliance with the standard. It is commonly observed that without automated means for compliance checking, coding rules have virtually *no*

[1] For a brief overview see http://spinroot.com/static/.

impact on software development. This then leads to a simple litmus test for the quality of a software design and development process: *which coding standards are used and how is compliance with that standard verified*?

- The failure data for spacecraft largely matches observations made by Charles Perrow[2] when writing about failure in complex systems: major failures, often defeating multiple layers of protection, can result from the unintended coupling of sub-systems or system components that were designed and assumed to be independent. The unintended coupling allows small failures to propagate and connect in unforeseen ways. These observations reinforce the importance of self-checking, decoupling, and modularity, but it also raises the bar for a *defense-in-depth* strategy that includes software components. Many of the problems that lead to major failures in larger system can also be caught early in the design cycle through the use of model-based engineering techniques that are integrated with verification methods (e.g., logic model checking[3] techniques). A second litmus test is then: *which verification capabilities exist in the software development process to effectively support early fault detection?*

- Concurrency related defects in software are among the hardest to prevent and predict, and they are among the hardest to identify with conventional software test methods. A standard example of a concurrent software system is the real-time multi-tasking system commonly used in embedded systems. But concurrency problems can also strike seemingly sequentially executing code, as for instance, used in medical devices. Software-based systems interact with their environment through peripheral devices (sensors and actuators), and generally have watchdog timers that can generate asynchronous *interrupts*. The interrupt-handlers define concurrent threads of execution, and their interaction with the main code of an application can have unintended consequences, sometimes leading to significant failure. Also from this perspective, the conclusion is inevitable that the highest standards in software quality control and the use of the strongest design verification techniques are essential for the development of safety-critical systems. Where appropriate, strong *evidence* of the successful application of these techniques should be made available to regulators.

What This Means

In safety-critical software development *any* statement about software reliability, be it as a separate component or as a functional part in a larger system, must be supported by strong supporting *evidence*. Complete and convincing insight should further be provided about the set of assumptions that underpin safety cases. This type of *evidence-based* safety argument should include evidence of a well-controlled software development *process*, evidence of *standards* used, and of *mechanisms* used to secure full *compliance* with these standards. Safety-critical software development requires the use of best-in-class static *source code analysis* tools and model-based design and design-verification techniques. Critical parts of the software that involve concurrency should be *formally verified* with the best available technologies.

Pasadena, 24 January 2011

---

[2] C. Perrow, *Normal accidents: living with high-risk technologies*, Basic Books, NY, 1984.

[3] See, for instance, http://spinroot.com/.