Reliable Software Systems Design

Gerard J. Holzmann

Laboratory for Reliable Software
NASA/JPL Pasadena, CA 91109, USA

Abstract

The grand challenge that is the focus of this conference targets the development of a practical methodology for software verification: a methodology that can help us to reduce the number of residual defects in software products. Reducing residual defects is of course not in itself the objective of this exercise; the true objective is to reduce the number of *failures* in the use of software products. Or in other words: the objective is the development of a methodology for "reliable software systems design."

It has often been argued that with the right training, discipline, and tools it should be possible to produce zero-defect code. Very few things in life, though, are zero-defect – not even the things that can be considered life critical. If you practice sky-diving, you are probably acutely aware that your main parachute could fail to open, no matter how carefully you check it before each jump. The parachutist would also be wise not to trust a company that tries to sell him a zero-defect parachute. He is more likely to avoid disaster by bringing a spare chute on his jumps. That is: the seasoned parachutist takes the possibility of *component* failure into account in the adoption of a system that has a significantly lower probability of *system* failure. Elevators are another good example. Of course an elevator can fail, for instance when the cable from which it is suspended breaks. But, the elevator system as a whole is designed in such a way that when the cable breaks, the car will not come crashing down. We trust the system, even though we know that none of its components are zero-defect. Note that in the case of the elevator mere redundancy does not solve the problem (i.e., operating multiple elevators in parallel). A reliable system is designed with the possibility of *component* failure in mind, and with remedies in place to significantly reduce the odds of *system* failure.

It is worth contemplating how deeply engrained the discipline of reliable system design really is, outside software engineering. If your kitchen-sink leaks, you can close a valve that stops the flow of water to that sink. The valve is there because experience has shown that sinks do occasionally leak, no matter how carefully they are constructed to prevent just that. If you short-circuit an electrical outlet in your home, a fuse will blow. The fuse is there to prevent greater disaster in case the unimaginable happens. The presence of the fuse and the valve do not signify an implicit acceptance of sloppy workmanship; they are an essential part of reliable *system* design.

Most software today is build without valves and fuses. We try to build perfect parachutes that do not need backup. When software fails, we blame the developer for failing to be perfect. Would it not be wiser to assume from the start that even carefully constructed and verified software components, like all other things in life, may fail in unexpected ways, and use this knowledge to construct assemblies of components that provide independently *verifiable* system reliability?