Opinion – Taking Software Seriously

P. B. Ladkin
University of Bielefeld

Appeared in: *Journal of System Safety 41(3), May-June 2005*

March 29, 2005

Say you are a software house, or a company division, producing safety-critical software for airplanes, or cars, or air traffic control, or power stations. If you are in Europe or the Asia-Pacific region, chances are that you will have to conform with the requirements of the international standard on functional safety of electric/electronic/programmable electronic (E/E/PE, pronounced "eepy") systems. If your clients are trying to certify an aircraft, you will have had to conform to RTCA DO178B (EA12B in Europe). Both these standards require that, for the most critical systems, someone will have had to demonstrate a dangerous failure rate of at most one failure in one hundred million, respectively one billion hours of operation (the fabled ten-to-the-minus-nine rate), for the device running your software.

Can you do it? Can you build your software to those standards? Can you persuade others that you have done so?

There are a number of data points one could use. One is the failure rate one can expect from faults in the delivered software. (What? You deliver software without faults? How do you know? Are you really to be believed? What if you started from faulty requirements? It wouldn't be your fault, but your code still wouldn't do what is needed.) Another is the confidence in the software that one can achieve through testing. (What? You test exhaustively? You must be writing tiny programs. Or underestimating the environment. What if the ROM containing your program gets zapped by

an alpha particle and an opcode bit gets flipped? Does your program still work correctly?) A third is what you can reasonably claim as the level of reliability or safety of your program. (Note that reliability, the ability of the software to perform its function, is not the same as safety, that the software will cause no harmful action to be performed.)

First, what do we know about defect rates or failure rates? A discussion in early 2004 between John McDermid, Martyn Thomas, Peter Amey and myself led to agreement that current good practice achieves a defect rate in delivered software of less than one per delivered KLOC. Les Hatton has achieved a failure rate in C code developed according to his stringent "Safer C" procedures of about 0.24 per KXLOC (XLOX = executable LOC) [Hat05]. The company Praxis High Integrity Systems, which documents its work extensively in the scientific literature, achieved an instrumented defect rate of 0.22 per KSLOC (SLOC = source LOC, including comments and annotations) on the SHOLIS shipboard helicopter landing advisory system in 1997 (size 27 KSLOC), 0.04 defects per KSLOC on the Mondex smart card security project in 1999 (size 100 KSLOC), and comparable or lower defect rates on more recent projects [Am05]. I know of no better results than those of Praxis.

To summarise, if you are down to one defect per 4 KLOC, you are doing well, but not as well as those who are down to one defect per 25 KLOC.

Second, what do we know about the efficacy of testing SW? Quite a lot. There is a hard mathematical limit to practical testing of complex systems with non-continuous failure modes, such as SW. You cannot breach it by testing "smarter". This result was published by Bev Littlewood and Lorenzo Strigini using Bayesian calculations, and Rick Butler and George Finelli using frequentist calculations, in 1993 [LitStr93, ButFin93]. It disturbs me that it is still not generally known.

Littlewood and Strigini noted that if you want to develop confidence, through testing, that your software fails less than once every million hours (the "posterior probability" in Bayesian terms), you already have to start out with that level of confidence before you test (the "prior probability"). How can you possibly attain that prior confidence when you know that the very best practice only gets you down to one fault per few KLOC? Answer: you can't. And that goes also for rates of once-a-billion-hours.

To summarise, the very best statistical-testing regime will find you those faults which lead to failure at a rate equal to or more frequent than one hundred thousand operational hours.

How many failures are those? Edward Adams analysed statistics on design errors in IBM software as reported by customers from 1975 through 1980 [Ad84]. Fully one third of the reports concerned faults that could be expected to result in failures less frequently than once every 60k operational months. Consider that there are 672, 696, 720 or 744 hours in a month, and we see that these faults lie outside the failure-frequency range of those expected to be discovered through testing. The answer to the question is that through statistical testing you can expect to find those faults which cause at most two-thirds of your failures. (What do you do about that remaining one-third?)

Third and finally, what can you reasonably claim as the reliability or safety of your program?

At this point we depart from consensus. Some organisations set a target level of safety (TLS) and then try to demonstrate that they have or will have achieved it. The problem with this approach lies with people such as myself and colleagues who see a TLS setting an accident rate lower or much lower than one per million operational hours, and set about finding mistakes in the arguments, at which we inevitably succeed. And then some organisations develop their systems according to rigorous development processes, such as SEI's Capability Maturity Model and derivatives. The problem with this approach is that no reliable correlation has ever been demonstrated between adherence to a development-process model and the quality of the resulting product, no matter how much we might wish for one. And then some organisations develop their software according to rigorous mathematical-logical procedures, measure what they have accomplished, and provide those measurements as evidence in assessing how safe the product is. This approach demonstrably eliminates whole classes of errors, provided it is efficiently incorporated into development procedures. There is no known technical problem with it. However, like the other two approaches, it cannot support undemonstrable claims of ultra-low dangerous-failure rates. Best simply to give them up.

Acknowledements

Thank you Martyn Thomas and Harold Thimbleby for helpful comments on a draft.

References

Ad84 Edward N. Adams, Optimizing Preventive Service of Software Products, IBM Journal of Research and Development 28(1):2-14, January 1984. Available from www.research.ibm.com/journal/rd/281/ibmrd2801B.pdf

Am05 Peter Amey/Praxis High Integrity Systems, IEC 61508-conformant Software Development with SPARK, the Fifth Bieleschweig Workshop on System Engineering, Munich, Germany, April 2005. Presentation slides available from `www.rvs.uni-bielefeld.de` ⟶ Bieleschweig Workshops ⟶ Fifth Bieleschweig Workshop

ButFin93 R.W. Butler and G.B.Finelli, The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software, IEEE Transactions on Software Engineering, 19(1):3-12, January 1993. Available from `techreports.larc.nasa.gov/ltrs/PDF/ieee-trans-se-19-1.pdf`

Hat05 Les Hatton, personal communication, also in Designing and Implementing Efficient Tests and Test Strategies, AsiaStar 2004, Canberra, Australia 2004. Presentation slides available from `www.leshatton.org`

LitStr93 B. Littlewood and L. Strigini, Validation of Ultra-High Dependability for Software-based Systems. Communications of the ACM, 36(11):69–80, 1993. Available from www.csr.city.ac.uk ⟶ Staff ⟶ Lorenzo Strigini ⟶ Papers and Abstracts ⟶ CACM, November 1993.