HIGH-INTEGRITY SOFTWARE, COMPUTATION AND THE SCIENTIFIC METHOD

Les Hatton*, CISM, Kingston University, UK

Abstract

Computation rightly occupies a central role in modern science. Datasets are enormous and the processing implications of some algorithms are equally staggering. With the continuing difficulties in quantifying the results of complex computations, it is of increasing importance to understand its role in the essentially Popperian scientific method. In this paper, some of the problems with computation, for example the long-term unquantifiable presence of undiscovered defect, problems with programming languages and process issues will be explored with numerous examples. One of the aims of the paper is to understand the implications of trying to produce high-integrity software and the limitations which still exist.

Unfortunately Computer Science itself suffers from an inability to be suitably critical of its practices and has operated in a largely measurement-free vacuum since its earliest days. Within CS itself, this has not been so damaging in that it simply leads to unconstrained creativity and a rapid turnover of new technologies. In the applied sciences however which have to depend on computational results, such unquantifiability significantly undermines trust.

It is time this particular demon was put to rest.

BACKGROUND

High-integrity systems are fundamentally those systems which on failure, have a significant impact on humans, organisations, society or the environment. It is traditional to split these into two categories

- Safety-critical systems. Such systems on failure, have a direct impact on human safety or indirectly by damaging the environment. Examples of these include aircraft active avionics, railway signalling software or medical imaging software.
- Mission-critical systems. Such systems on failure, are typically associated with financial loss. In an organisation this could be failure of its accounting systems, customer information or other enterprise systems. However, in the context of science, such systems on failure would be directly associated with unintentionally misleading results. Indirectly, this may lead to financial loss but the essential issue here is untrustworthiness of the results.

In the context of Accelerator and Large Experimental Physics Control Systems, both categories will be represented. The failure of Control Systems is frequently safety-critical whilst data acquisition and processing is mission-critical in that failure here may well lead to misleading results.

The processes involved in producing Safety-Critical Systems are well-rehearsed, well-documented by numerous standards, both military (DO-178B, DO-254, DEF STAN OO-55/56) and civilian (CENELEC EN 50126, EN 50128, EN 50129, IEC 61508) and strongly associated with legal liability. In this paper, therefore, I will restrict myself to the latter of these issues, viz. the risk of unintentionally misleading results and the potential impact on scientific research.

SOME ISSUES IN HIGH-INTEGRITY SOFTWARE

The Scientific Method

The highly influential 20th century philosopher Karl Popper was instrumental in delineating the scientific method by laying down the notions of deniability. Deniability in an experimental context is fundamentally linked to openness of technique and reproducibility of result. We make progress by ruthlessly eliminating those experimental results which cannot be reproduced within some suitable bound of error when an identical experiment is performed. Such methods have served science well and have come to embody the very essence of the scientific method.

I will now restate deniability in a software context as

- Truth cannot be verified by software testing, it can only be falsified,
- Falsification requires quantification of computational modelling error,
- Deniability is at the heart of progress in scientific modelling. We are always seeking to deny the truth of a result and continued failure simply adds weight to a result but not verification,

In this context, Computer Science has not done well.

Software Defect

The principle enemy of the programmer is software defect. In the absence of a well-defined set of categories, I will define a defect as a fault or mistake in the software which then causes the software to fail. Software failure occurs when the behaviour of a computer program departs from its expected behaviour. Every failure is associated with at least one fault but not all faults fail.

There are many kinds of fault and there are many kinds of failure.

* lesh@oakcomp.co.uk

Fault categories Although by no means exhaustive or independent, these might include

- The use of ambiguous programming language features as described for example by [4],
- Mistakes in logic, for example an incorrect programming branch,
- Errors of omission, (i.e. something important left out such as initialisation statement),
- Errors of commission, (i.e. something put in which should not be there),
- Various abuses of floating-point arithmetic, [11],

Fault occurrence rates have been thoroughly examined by many researchers, for example the seminal contributions of Basili, [2]. Such faults are typically identified as occurring in the range 1-10 per thousand executable lines of code for typical software with arguably the best software around 0.1 on this scale.

Where computer scientists have made little progress is on *quantifying the effects these faults have on the actual output values of the computation, in other words on how they fail.* Let me then consider different kinds of failure.

Failure categories Examples of these might include

- An unintended program halt. These can occur for many reasons in programming languages. For example, an incorrectly dereferenced pointer or a pointer alignment problem in languages like C and C++; a divide by zero which causes a run-time exception; overflows; underflows and so on. Such errors although traumatic for a safety-critical system because service is interrupted perhaps during a critical phase, they tend not to be particularly serious in scientific computation because the scientist is immediately alerted that there is something wrong.
- Wildly incorrect results. These are results which are so far wrong that they cannot be correct as for example the rather surprised Malaysian man who received a $218 trillion dollar phone bill in 2006, (http://www.msnbc.msn.com/id/12247590 /ns/world_news-weird_news/t/think-your-phone-bill-high-try-trillion/, accessed 30-Sep-2011). These are not serious in scientific computing either as they are readily identifiable as spurious. Such a result is extremely unlikely to mislead.
- Subtly incorrect results. These are results which are profoundly dangerous in scientific computing. It may be thought if they are subtle, they cannot mislead but this was profoundly disproved by [7] who showed that the presence of previously undetected software faults in a comparison of 9 seismic data processing packages independently developed to the same mathematical specifications effectively destroyed the level of agreement to the point where the data became highly misleading. This is exemplified in Figure 1. Although some 17 years old, the programming language used is still in widespread use (Fortran), the packages are still in widespread use, the software processes used are effectively unchanged today and scientific programmers still make mistakes, so its relevance is undiminished.

The experiment of [7] is known as an N-version experiment. Although independent versions of software written to the same specifications do not fail independently, [12], [14], they fail sufficiently independently to provide important insight into quantifying the effects of failure.

Another effective way of achieving this is to release all source code so that it can be independently checked. This is very similar to N-version experiments in that source code is subjected to many analysing minds in parallel. In N-version work, this leads to N versions. In open source work, this will usually lead to one version, independently checked by N persons.

There have been many attempts to build models which can predict unreliability based on some statically measurable property of software, for example the cyclomatic complexity, (a graph theoretic measure of decision complexity), first proposed by McCabe [13] but these have not been very successful, [3].

A typical example of a correlation between recorded defects and the cyclomatic number in a study by Hopkins and Hatton [8] illustrates the unfocused relationship between them with no statistically significant patterns.

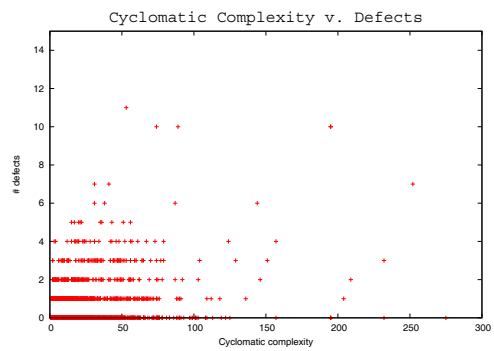

Figure 2: Recorded defects over a period of several years in the NAG Fortran library plotted against the cyclomatic number of the component in which the respective defects were found.

Programming Languages

Problems with ambiguity in programming languages have been reported on many occasions and regularly occur in scientific software in multiple languages, [4]. These include very simple issues like the use of uninitialised variables to far more subtle aspects of computation such as any reliance on a specified order of evaluation, (which is not defined for the majority of programming languages).

Recent research suggests that there are implementation and language independent properties which can be exploited. In [6] I used information theoretic arguments in the

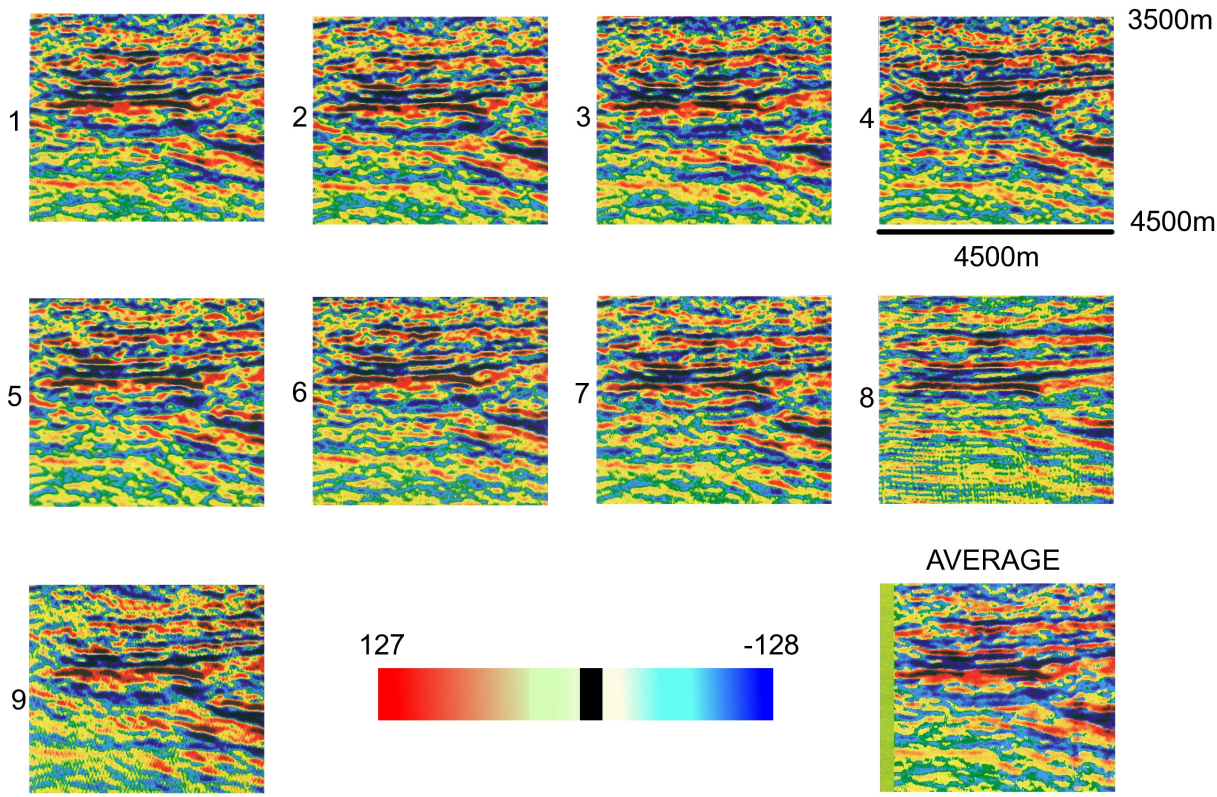

Figure 1: A comparison of nine independently developed packages in the same programming language on the same input seismological data shown by [7]. The y-axis is depth of burial in the earth and the x-axis is distance along the surface of the earth. The outputs vary in the second and sometimes first significant figure. The data needs about three significant figures of accuracy to resolve the geological features (in this case an unconformity trap for a gas field in the North Sea) sufficiently accurately for statistically reliable positioning of a well. The differences were demonstrated to be entirely due to software faults which had been present in the packages in some cases for years. This is by no means unusual as shown by Adams, [1] who demonstrated that a significant number of faults took hundreds and in some cases thousands of execution years to fail for the first time.

context of statistical mechanics to show that the probability p_i of the i^{th} software component in a system containing say, t_i programming tokens in total, obeyed a power-law in the alphabet of *unique* tokens a_i used to build it *independently of the programming language or its application area.* A token here is a keyword, identifier name or operator of a programming language.

$$p_i \approx (a_i)^{-\beta} \tag{1}$$

This in turn leads to a simple prediction that defects will also obey a power-law. One important property of power-law distributions is *clustering* so this gives some theoretical support for the widely-observed phenomenon that defects in software systems do in fact cluster. This can clearly be seen for example in the NAG Fortran scientific library as demonstrated by Hopkins and Hatton [8] as shown in table 1 where all reported defects cluster in only 30% of the executable lines.

Table 1: Defects in NAG Fortran Library

Defects	Components	XLOC
0	2865	179947
1	530	47669
2	129	14963
3	82	13220
4	31	5084
5	10	1195
6	4	1153
7	3	1025
> 7	5	1867

As I pointed out in [6] however, this appears to be a statistical phenomenon with little to be gained by asking the question why are almost 80% of all components defect free. This is akin to asking why somebody has won the lottery, the answer being of course that everybody else didn't.

Clustering is of real value in software testing because if a defect is found, then there is an increased probability of finding another if the same component is re-examined. In other words test effectiveness is improved.

Software Process

Software process in its many forms has become synonymous with High-Integrity developments thanks to initiatives such as the CMMi (Capability Maturity Model Integration) at the Software Engineering Institute of Carnegie-Mellon. This in turn was based on the pioneering work of Humphrey, [9], [10]. The principle notions of this maturity process enabling an organisation to progress through various levels towards full defect management, tracking and prevention have proven to be of great importance.

As with all good things though, there is no question that some implementations of these ideals have become poisoned by bureaucracy and it is all too easy for complacency to creep in, [5]. Nowhere is this more tragically seen than in the events leading up the Afghanistan crash of a RAF Nimrod in 2006, (http://www.official-documents.gov.uk/document/hc0809/hc10/1025/1025.pdf, accessed 01-Oct-2011). In his summing-up, Charles Haddon-Cave QC, author of the independent review stated:

> "Unfortunately, the Nimrod Safety Case was a lamentable job from start to finish. It was riddled with errors. It missed the key dangers. Its production is a story of incompetence, complacency and cynicism."

As important as management of software process is, it should be noted that it makes no guarantees and can shed little light on the effect that residual delivered defects can have on the output of a computation.

Specific Issues in High-Integrity

In the preparation to this paper, I was made aware of a number of issues of relevance in the sphere of Accelerator and Large Experimental Physics Control Systems and I will list these here and make comments on them based on the background above.

Risk versus ingenuity The LHC (Large Hadron Collider) is novel and ingenuity figures large even to get the machine realised. However, risk has always accompanied ingenuity. An ingenious solution is almost by definition new. However new solutions carry risk far more than well-tried solutions because not all of their potential side-effects have been realised. As I stated at the beginning, in terms of dealing with potential safety, this remains a well-rehearsed process with a number of techniques available such as FMECA (Failure Modes, Effects and Criticality Analysis).

Ingenious software solutions are a little different. Ingenuity in software can and often has been associated with obscurity. Obscurity in software is usually a menace because it reduces the power of peer review. Peer review is one of the main weapons in the battle against software defect.

Knowledge transfer and Education This is a vital area. Many of the ways that the problems described earlier in this paper manifest themselves, would be ameliorated with education. Problems with programming languages; the acceptance of basic training in software engineering methods for scientists; engineering technologies such as software redundancy; the imperative for reproducibility; fundamental notions of how to test software and the realisation that software testing is Poppperian in the deniable sense I defined at the beginning of this article, and so on.

In terms of the accuracy of the results of a computation, it is not only the specification, often resident in just a few minds, which is of value. Ultimately, the results depend on the code. The code must therefore be disseminated with the specification to maximise the chance of reproducibility.

At the risk of causing a little upset, I personally have found some scientists highly resistant to the notion that their code is almost inevitably in error. Amongst such scientists, there seems to be the notion that in comparison to scientific research, software implementation is in some sense 'easy'. I have been a practising scientist for much of my career and am a competent mathematician but the formidable difficulties presented by trying to verify the results of a scientific computation are enough to beat humility into anyone who realises just how difficult it is.

Consequently, I would like all people who deal with scientific software to respond to being faced with 100,000 lines of code in the same way, *if we've done a really good job, there will only be around 100 defects in this of which a significant number may well undermine the accuracy of the results.* That is a refreshing, pragmatic and above all, justified approach and I sincerely hope that education will enable this.

Approaches to Design It is commonly thought that High-Integrity systems are specified, built, tested and released. This rather comfortable viewpoint is still frequently taught in universities. In my experience, it could not be further from the truth. In contrast, *every* stage of the implementation of a trustworthy system is accompanied by iteration or prototyping to make sure ideas are feasible before cementing them into a system.

As a very simple recommendation, it is not common for software testers to be included at the design stage. They *must* however be included from the earliest days because testability is an excellent brake on vaulting and frequently unimplementable ambition.

Dealing with technological leaps In High-Integrity systems, the ideal case generally is to ignore them on the grounds that they carry too much risk as described in the section above. However, each must be taken on its own merits recognising that some technological leaps carry great benefit. A very simple example arises from the interoperability and openness of the Internet protocols. The Internet was not designed and no doubt if we had tried, we would have failed. However, from these tiny beginnings of portability and openness allowing cooperating but disparate development, the vast complexity and indeed reliability, of the Internet evolved to everybody's benefit.

CONCLUSIONS

In this essay, I have tried to present some background on why it is so difficult to quantify the inevitable inaccuracy in the results of scientific computation and why the notions of Popperian deniability demand that we make more progress in this vital area.

I have pointed out a few areas in which we are making progress, for example by designing N-versions systems but these are very expensive. Ultimately, there seems no alternative than to open all source code to public scrutiny and hope to garner the same benefits that open source has brought for example to the world of operating systems, where the Linux kernel is now one of the most reliable complex pieces of software the human race has ever built.

Perhaps we can do the same with scientific computation.

REFERENCES

[1] E. Adams. Optimising preventive service of software products. *IBM Journal of Research and Development*, 1(28):2–14, 1984.

[2] B. Boehm, H.D. Romback, and M.V. Zelkwitz. *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer, 1st edition, 2005. ISBN 3-540-24547-2.

[3] N.E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[4] L. Hatton. The T experiments: Errors in scientific software. *IEEE Computational Science and Engineering*, 4(2):27–38, April 1997.

[5] L. Hatton. Bureaucracy, Safety and Software: a potentially lethal cocktail. In C. Dale and T. Anderson, editors, *Making Systems Safer: Proceedings of the 18th Safety-Critical Systems Symposium*, pages p.21–36, London, UK, 2009. Springer.

[6] L. Hatton. Scientific computation and the scientific method: a tentative road map for convergence. In *IFIP / SIAM / NIST Working Conference on Uncertainty Quantification in Scientific Computing*, 2011.

[7] L. Hatton and A. Roberts. How accurate is scientific software ? *IEEE Transactions on Software Engineering*, 20(10), 1994.

[8] T.R. Hopkins and L. Hatton. Defect correlations in a major numerical library. *Submitted for publication*, 2008. Preprint available at http://www.leshatton.org/NAG01_01-08.html.

[9] W. Humphrey. *Managing the Software Process*. Addison-Wesley, 1989. ISBN 0-201-18095-2.

[10] W. Humphrey. *A discipline of software engineering*. Addison-Wesley, 1995. ISBN 0-201-54610-8.

[11] W. Kahan. Desperately needed remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering. In *IFIP / SIAM / NIST Working Conference on Uncertainty Quantification in Scientific Computing*, 2011.

[12] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[13] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[14] Meine van der Meulen and Miguel A. Revilla. The effectiveness of software diversity in a large population of programs. *IEEE Trans. Software Eng.*, 34(6):753–764, 2008.