Practical Statistical Evaluation of Critical Software

Peter Bernard Ladkin

University of Bielefeld and Causalis Limited

Bev Littlewood

City University London

Abstract *In 2010, Rolf Spiker approached one of us with a query from a client concerning advisory material in IEC 61508 on the statistical evaluation of software. We realised that there is a dearth of practical guidance for those who wish to evaluate critical software statistically. We believe statistical evaluation of software is an increasingly important assurance technique. We commence with a brief introduction to some of the simpler statistics and then consider discursively the issues which arise during evaluation.*

1 Introduction

It is sometimes said that "software failures are 'systematic' and therefore it does not make sense to talk of software reliability in probabilistic terms". It is true that software fails systematically, in that, if a program fails in certain circumstances, it will always fail when those circumstances are exactly repeated. Where then, it is asked, lies the uncertainty that requires the use of probabilistic models and measures of reliability?

There are two main sources of uncertainty. First, there is uncertainty about which inputs, of the many possible inputs the software could receive, will result in failure (of the software to fulfil its intended purpose) when executed. Second, there is uncertainty about which inputs the software will in fact receive in the future as it executes: these inputs will depend upon the external operating environment, about which there will be uncertainty.

It follows from these two sources that there is uncertainty about when a program will receive an input that will cause it to fail. Failures thus form a stochastic

process (a random process) as time progresses during execution of the software. There are some simple probabilistic models for such failure processes (as well as some complicated ones). We describe these briefly and show how they can be used to obtain quantitative probabilistic measures of software reliability.

Because software failures occur randomly, it follows that many of the classic measures of reliability that have been used for decades in hardware reliability are also appropriate for software: examples include failure rate (for continuously operating systems, such as nuclear reactor control systems); probability of failure on demand (pfd) (for demand-based systems, such as nuclear reactor protection systems); mean time to failure; and so on. This commonality of measures of reliability between software and hardware is important, since practical interest will centre upon the reliability of systems comprising both. However, the mechanism of failure of software differs from that of hardware, and we need to understand this in order to carry out reliability evaluation.

2 Simple probability models of the software failure process

In this section we outline two simple probability models that describe two common types of failure processes: a discrete-time (counting) model for on-demand systems, and a continuous-time model for continuously operating systems, respectively. Many software-based systems fall into one of these two classes, although there are, of course, exceptions: see discussion in Section 3.

2.1 On-demand software based systems

Consider a nuclear reactor protection system (or "safety system", as it is called; NRPS). An idealized view of the NRPS is that its role is to act only when the reactor enters a hazardous state (the "demand"), whereupon its function is to shut down the reactivity and keep the reactor in a safe state. Such demands upon the NRPS might arise because of the failure of a wider system – e.g. the continuously operating control system – and in a well-designed reactor they could be expected to be quite rare, say about once a year. A dangerous failure of the NRPS would be the system not responding to a legitimate demand. Part of the wider safety case for the reactor would contain a requirement that the probability of such a failure on demand (pfd) of the protection system be adequately small[1].

1 For example, in the case of the UK Sizewell B reactor, this figure was 10^{-7} (of which 10^{-3} was allocated to the software-based Primary Protection System (PPS), and 10^{-4} to the hardware-only Secondary Protection System (SPS), in this 1-out-of-2 configuration).

How could such a figure be claimed with high confidence? First of all we need a simple model for testing and operational use of this kind of on-demand software-based system.

We observe that there is a (probably very large) set of possible demands. Label these 1, 2, 3, … Selection of successive demands by the operational environment (i.e. the wider reactor and its environment in our example) occurs randomly and, we claim, independently, with $P_i=Pr$ (demand i is selected) forming a probability distribution over all demands. Note that selection is not generally equi-probable (indeed this is usually unlikely). Each demand either results in failure, or does not. Define the variable:

$X_i = 1$ if demand i results in failure

$X_i = 0$ otherwise

It is easy to see that the probability of failure of a randomly selected demand is:

$$pfd = \sum_i P_i \times X_i \qquad (1)$$

In practice, we would not know the distribution $\{P_i\}$ completely; nor would we know which demands cause failure, so that the $\{X_i\}$ will also be unknown. It follows that (1) cannot be used to calculate *pfd*.

Instead, *pfd* can be estimated statistically from the results of operational testing, i.e. testing that selects the demands in exactly the same way they would be selected in operational use. Such testing is often based on simulation that uses an understanding of the physical world in which the computer-based system operates. In the example of a reactor protection system, this would require knowledge of the physics and engineering of the reactor, and of the reactor's operational environment.

In such testing we observe a sequence of trials, each of which will result in either success or failure. If the trials are statistically independent, and the probability of failure has the constant value *pfd* for each trial, they are called Bernoulli trials. A sequence of such trials forms a particularly simple stochastic process, called a Bernoulli process.

There are two random variables of interest in such a process. Firstly, the number of failures in a given number, n, of successive demands. This has a Binomial distribution:

$$\Pr(r \text{ failures occur in } n \text{ demands}) = \binom{r}{n} pfd^r (1-pfd)^{n-r} \qquad (2)$$

Secondly, the number of demands until the next failure has a geometric distribution:

$$\Pr(\text{number of demands up to and including next failure} = r) = (1 - pfd)^{r-1}\, pfd \quad (3)$$

Notice that this is true regardless of whether we count starting from a failed demand, or not: the Bernoulli process is said to be *memory-less*.

If we observe r failures in n trials it is straightforward to compute estimates of *pfd* as a function of r and n: details can be found in any introductory text-book on stochastic processes, e.g. (Siegrist 2014). In particular, if we see *no* failures[2] (i.e. r=0), confidence bounds for *pfd* can be obtained as in the Table 1. Table 1 includes numbers related the IEC 61508 SIL levels, taken "one-sided". They arise from the mathematics of the Binomial distribution in Equation (3).

Table 1: Numbers of failure-free demands required to obtain confidence in different pfd levels

SIL level	Acceptable probability of failure	Number of failure-free demands for 95% confidence	Number of failure-free demands for 99% confidence
SIL 1 or greater	$<10^{-1}$	$3\text{x}10^{1}$	$4.6\text{x}10^{1}$
SIL 2 or greater	$<10^{-2}$	$3\text{x}10^{2}$	$4.6\text{x}10^{2}$
SIL 3 or greater	$<10^{-3}$	$3\text{x}10^{3}$	$4.6\text{x}10^{3}$
SIL 4	$<10^{-4}$	$3\text{x}10^{4}$	$4.6\text{x}10^{4}$

Table 1 is simply for illustration. Generally, 95% confidence can be placed in a claim that the *pfd* is smaller than 10^{-x} if $3\text{x}10^{x}$ failure-free demands have been observed, and so on.

These results are based on two important assumptions, and a user needs to be confident that these are satisfied for hisher particular application.

First, the statistical properties of the test case selection need to be exactly the same as those of demand selection in operation. If the distribution of selection probabilities of the test cases was $\{P_i^*\}$, different from $\{P_i\}$, then the probability of failure on demand *in test* will be

$$pfd = \sum_i P_i \times X_i \quad (4)$$

which will not be the same as *pfd*, (1), the probability of failure on demand in operation. In such a case, estimates of the former will not be accurate estimates of the latter.

Second, successive demands must be independent, with constant probability of failure. In our illustrative example of a reactor protection system this may be a

2 In some safety-critical industries, regulators will accept *only* evidence of failure-free working in support of *pfd* claims.

plausible assumption since the demands will be far separated in calendar time. It seems reasonable to assume that today's demand is not affected by the nature of a demand that occurred last year: i.e. knowing whether or not last year's demand failed will not affect the probability that *this* demand will fail, which is just the constant *pfd*.

2.2 Continuously operating software-based systems

Many software-based systems operate in continuous time. Common examples include those that control complex hardware: e.g. automobile engine control systems, fly-by-wire airplane flight control systems, nuclear reactor control systems. In such examples, the state of the system under control will be determined by the elements of a many-dimensional vector of inputs – for example, in the case of a reactor control system: temperatures, pressures, coolant flow rates, etc.

For continuously-operating software-based systems, the vector of inputs forms an evolving *trajectory*, or path, in the multi-dimensional input space as (continuous) time passes. With this way of looking at things, software failures can be identified with regions of the input space. Call these *fault regions*. When the execution trajectory enters a fault region, a software failure occurs.

There are two sources of uncertainty, as in Section 2.1. First, there will be uncertainty about the nature ("shape") and location of the fault regions in the input space. Secondly, there will be uncertainty about the future direction an execution trajectory will take. Thus as time passes the occurrence of failures – points on the time axis – is random: it forms a *continuous-time* stochastic point process.

The simplest such process is called a Poisson process, and this will often be an accurate model of the failure process of continuously operating software-based systems. A Poisson process is characterised by a single parameter, λ, its *failure rate*, measured for example in failures per hour. As in the case of on-demand systems discussed in Section 2.1, there are two random variables of interest. First, the number of failures in a given interval, $(0,t)$, of elapsed time has a Poisson distribution:

$$\Pr(r \text{ failures occur in } (0,t)) = \frac{(\lambda t)^r e^{-\lambda t}}{r!} \tag{5}$$

Second, the time to the next failure has an exponential distribution with probability density function

$$\lambda e^{-\lambda t} \tag{6}$$

Notice that, as in the Bernoulli process, this is true regardless of whether we measure the time from a failure or not: the Poisson process is memory-less.

We can use test data to estimate λ. If r failures have been observed in elapsed time t it is a simple matter to estimate λ, calculate confidence bounds, etc. Many textbooks give the simple details, e.g. (Siegrist 2014). As before for on-demand systems, a particularly interesting case for safety-critical applications is where r=0. In Table 2 are some examples of confidence bounds based on IEC 61508 SIL levels.

Table 2: Numbers of failure-free hours required to obtain confidence in different failure-rate levels

SIL level	Acceptable probability of failure per hour	Number of failure-free hours for 95% confidence	Number of failure-free hours for 99% confidence
SIL 1 or greater	$<10^{-5}$	$3x10^5$	$4.6x10^5$
SIL 2 or greater	$<10^{-6}$	$3x10^6$	$4.6x10^6$
SIL 3 or greater	$<10^{-7}$	$3x10^7$	$4.6x10^7$
SIL 4	$<10^{-8}$	$3x10^8$	$4.6x10^8$

Again, these numbers are just illustrative. Generally, if it is required to claim a failure rate better than 10^{-x}, with 95% confidence, then $3x10^x$ hours or more of failure-free working need to be observed; and so on.

3 Some Observations on Applicability

The advantages of using these stochastic processes for interpreting software behaviour in situ are threefold. First, the pertinent mathematics of these stochastic processes are simple, clear, and well-understood - for Bernoulli processes for some 300 years! (Bernoulli 1713). Line engineers tasked with assessing software could be routinely expected to develop the pertinent mathematical skills. Second, the key parameters are few and clear, so that it is often a straightforward matter to identify these key parameters in system operation and be reasonably assured one has them right. Third, interpretations as Bernoulli resp. Poisson processes are indeed often feasible in software operation. However, whilst many systems fit into one of these classes – on-demand systems operating in discrete time, or continuously operating systems in continuous time – there are also many exceptions.

Indeed, the choice of which of the two interpretations to use can sometimes be a matter of convenience. Consider, for example, a safety-critical flight control system in a civil airplane: this is obviously a continuous time system. But it may be convenient sometimes to treat it as a discrete time system where the measure of interest is probability of failure per flight. Here, a “demand” is a “flight”. A count of the number of demands – i.e. take-offs and landings – may better reflect the exposure of the system to possible failure than calendar time, which includes hours spent in straight and level flight.

In the example of a protection system used above, only failures to respond to a (genuine) demand were considered, and these naturally form a discrete time stochastic process in terms of the sequence of successive demands. Such failures are sometimes called "Type 1", in contrast to "Type 2" failures in which the protection system incorrectly shuts down the reactor when the latter is not in fact in a hazardous state. Type 2 failures, in contrast to Type 1 failures, form a continuous-time stochastic process of events in real time (i.e. clock, or calendar time). Type 2 failures are generally less serious than Type 1 failures, and may not impinge on system safety, but they certainly affect system *reliability*. It would be reasonable to have probabilistic requirements for both types, necessitating the use of both of the probability models described in Section 2.

We have endeavoured to make clear in the examples above that the discreteness or continuity of time concerns the world outside the system, and not the system itself. Whilst it is true that computer systems themselves can be thought to operate in discrete time – clock cycle time – this discreteness is entirely distinct from the worldly discreteness in a Bernoulli process which concerns successive demands upon a computer-based system. There is to our knowledge no simple way (indeed at time of writing we do not know of *any* reasonable way) to relate processor clock cycles to the demands in a legitimate Bernoulli process model.

Of course, not all computer system failures can be modelled by a Bernoulli or a Poisson process. For example, the assumption of constancy of *pfd* (or failure rate) will be violated if fault fixes are made (or attempted) when failures occur, because the code has changed. One would not be measuring the selfsame object after such a change. In such cases, it might be expected that there will be reliability growth, at least in the long run[3]. More complex reliability growth models (RGMs) are needed to represent such situations and there is now a large scientific literature on problems of this kind.

However, it is questionable whether such models are appropriate for safety-critical systems. They require assumptions about the efficacy of fault-fixing that are difficult to justify, and thus may not produce conservative results. The simpler models described here, in contrast, require that no changes are made to the system as failures occur and are thus guaranteed to be conservative in this respect. In fact, as has been remarked earlier, in many safety-critical applications there will be a requirement that *no* failures are observed.

4 Determining Success and Failure

Talking about failure relies on having some notion of successful execution and non-successful (that is, failed) execution of software. There are generally two notions in common use when speaking about software execution.

3 Some fix attempts may not succeed. Some may introduce novel faults. But in the long run it might be expected that reliability will increase in spite of such reversals of fortune.

First is when something happens which does not conform with the expectations of a user of the software. What is meant here by "user" is also a fluid notion. I can use software without having any defined stake in its evaluation or ability to report on its operation, for example if I use third-party web-application software to perform a transaction. The term "stakeholder" might be more appropriate. When using WWW software to perform a transaction, I certainly have a stake in its (to me) correct operation, but it may still do things I don't wish – and the other party to the transaction may wish it so. There is nothing prima facie to say who is right about whether the software is operating correctly. And some software may be designed to force a third party to use it in certain ways uncomfortable for them. It follows that this notion of correctness is a social construct.

Second is when there is a rigorous specification of software behaviour. A failure can be defined as a behaviour (or the outcome of a behaviour) which does not conform with the specification.

There are notions which transgress the boundaries of these conceptions. Say I have a specification, but the specification is in retrospect not quite right (which is often the case). There may be behaviours which may be what I want, but which do not conform with the inaccurate specification.

There is not space here to investigate the notion of failure of software in any detail. However, the statistical evaluation of software does depend on a coherent, deterministic notion of failure of software. One must be able to say in any given circumstance whether the software has failed or has not failed. In the absence of such a clear notion in a specific case, software cannot effectively be evaluated statistically using the methods we have indicated above.

5 Some Tricky Issues

None of the above says that interpreting software operation in situ as a Bernoulli or Poisson process is a straightforward matter. Indeed, there is a case to be made that the key skill for an engineer wishing statistically to evaluate critical software is interpretive rather than mathematical. The hard question is: are you sure your process really is legitimately Bernoulli, respectively Poisson?

5.1 "Easter Egg"-Type Behaviour

A major issue is that there is no useful constraining relation between the behaviour of the software on one set of inputs and its behaviour on another, closely related set of inputs.

In certain consumer software of the past (and present), programmers would occasionally include so-called "Easter Eggs" (Wikipedia 2015). When a specific

combination of inputs was given, the software would cease functioning as intended and it would play a tune, or display a cute picture or greeting, or some such. The chosen trigger combinations were such as to be deemed to be extremely unlikely in normal operation, so only people who "knew" could usually evoke the Easter Egg behaviour.

Some software used in critical applications has a "debug" or "maintenance" mode (DMM) which allows a user access to internal data structures in the software. Giving the software input while in DMM results in output of interest to the maintainer, which will rarely be values appropriate for the critical function of the software. Thus this critical function will routinely fail when the software is in DMM. The software is switched into DMM by a specific combination of input values known to the developers/maintainers ("maintainer"), but not necessarily by the engineer wishing to use the software in a critical application and evaluating its use statistically ("client"). The maintainer knows about the quasi-Easter-Egg, the client not.

Suppose the software has been statistically evaluated on typical in-service inputs. Suppose future inputs are identical to those past inputs, with the sole exception that occasionally the DMM trigger input is seen. The software will fail (to fulfil its intended function) each time this trigger input occurs. The software failure behaviour in the future application will be decisively different (worse) than has been seen in the evaluation. But the difference in inputs from evaluation inputs to future inputs is just a single one of the input values! This shows clearly that the condition that future inputs must be the same, and occur with the same relative frequency, as in the evaluation, must be taken rigorously for predictions from the evaluation to be realised in the future use.

5.2 Masked Dependencies

Sometimes the behaviour of software is dependent upon input parameters which have not been explicitly recognised. If the behaviour of these parameters is different in the future use from that in the evaluation, then the software behaviour might well be different, even when the behaviour of the explicitly-recognised parameters stays the same.

A colleague tells of assessing a system for dependence on GPS. The developer assured the assessors that the software was not at all dependent on GPS signals: it had no function that would require location information; no such dependency had been deliberately implemented; indeed, an attempt had been made explicitly to avoid it. The software did not use library or other external functions that were known to rely on GPS.

The assessors brought in a GPS jammer and activated it. The software soon ceased to operate as intended because of the jamming. This, apparently, is not an uncommon occurrence (Thomas 2011, RAEng 2011).

5.3 Version Deviations

It is commonplace that minor changes to software may result in major changes in behaviour. Apple's "*goto fail*" bug in its TLS/SSL verification software, noted in February 2014 (Ducklin 2014), ensured that all WWW-site certificates were validated, no matter what their actual status as genuine or spoofed. That is a radical failure of intended function. However, the source code responsible was one line containing 11 ASCII characters that seems to have been spurious (an exact duplicate of the preceding line).

Since there is no estimable correlation between behaviour of software and source-code changes, there is no way of reliably estimating the failure behaviour of new versions of software based on the failure behaviour of previous versions and the nature of the changes. If, say, SW Version 1.2 has been evaluated, and a minor change has been made resulting in Version 1.3, then the failure behaviour of Version 1.3 cannot in general be reliably estimated from the evaluation of Version 1.2.

It may be possible to evaluate the behaviour of Version 1.3 if an impact analysis can demonstrate reliably that the changes made to Version 1.2 cannot affect the pertinent behaviour of the software. Such analyses move outside the realm of statistical evaluation and, to be performed reliably, likely involve the use of rigorous formal methods.

5.4 Failure Masking

Failure masking is a phenomenon often desired in fault-tolerant systems. Large parts of computer science have been devoted towards devising algorithms and techniques to tolerate failures, often but not always involving component redundancy. Failures so tolerated may not be apparent to the user; that is, may be "masked".

This is not the phenomenon usually meant when the term "failure masking" is used in statistical evaluation of software. Failure masking relevant to software evaluation occurs when a software component S fails or is imminently about to fail, but this failure is not registered because a larger or a different component C fails: the failure is registered as a failure of this second component C, and the state of software S is not registered. If software S is about to fail, or has failed unnoticed, then this would count for statistical evaluation as a failure of software S on the existing input. However, it is not registered. But is a precondition for successful statistical evaluation of S that all failures are registered.

One of the most well-known examples of failure masking concerns the meltdown of a Babcock and Wilcox 900 Pressurised Water Reactor (PWR) at the Three Mile Island Generating Station in Pennsylvania, USA, in March, 1979

(IAEA 2002). The primary coolant surrounding the reactors is itself cooled by a separate secondary cooling system, also water-based, and a heat-transfer mechanism. The secondary cooling system had stopped circulation, so the primary coolant was heating up. A relief valve (called an "electromatic relief valve" by the manufacturers, Dresser Industries (Perrow 1984) but "pressurizer relief valve" in (IAEA 2002), allows overheated primary coolant to overflow into a sink, relieving pressure in the primary containment (the pressure vessel holding the reactor core) due to the overheating. Enough primary coolant should remain after venting to continue to function, so the relief valve must close when pressure has reduced appropriately. The valve, however, failed to reseat and coolant continued to drain out; ultimately a third of it escaped through the valve. The valve position indicator itself had a fault and indicated to plant controllers that the relief valve was closed, when it wasn't. The failure of the indicator masked the failure of the relief valve. An animated image of the sequence of events is included in the U.S. Nuclear Regulatory Commission "backgrounder" (USNRC 2014).

Software failure masking occurred in the incident to Malaysian Airlines Boeing 777-200 9M-MRG in August 2005 (ATSB 2007). The software was fault tolerant, and, before and during the accident flight, operation of the software masked a previous failure of an air-data unit, whose erroneous values were treated as veridical by the primary flight control computer system, which then commanded significant and untoward deviations in pitch. The failure masking is considered in detail in (Johnson and Holloway 2007), which illustrates the difficulties that may arise in registering failures (of software or of hardware) accurately.

It is beyond scope here to consider failure masking in detail. Suffice it to say that considerable attention must be paid to its possibility where software is to be statistically evaluated.

5.5 Deviations from the Model

The conditions of memorylessness mentioned above are strong conditions on evaluation. An example is given in (Ladkin 2015) of non-memoryless behaviour in software with one failure condition. The particular interest of that example is that some people think that complex software such as real-time operating systems (RTOS) have "proved their worth" over sometimes millions of hours of "successful" operation and on this basis are appropriately dependable in new critical applications. There are many problems with such assertions, in particular the possibility of failure masking and version control issues mentioned above. A particular problem arises, though, if attempting to construe RTOS operation as a Poisson process for the purposes of statistical evaluation.

Suppose the RTOS has at least one failure mode. Then there is some short period of time, microseconds or milliseconds, just before such a failure when that failure becomes inevitable (for example, at the last instruction before a HALT). Let

such a short time period be ε. Then in that period ε the probability of failure of the RTOS is effectively 1. However, consider a time period of ε from start of boot-up. The probability of failure in this time for any reasonably well-used RTOS is effectively 0. So, for some time intervals of length ε in operation the probability of failure is effectively 1 and for other time intervals of the same length it is effectively 0. But the memoryless property of a Poisson process requires the probability of failure in any time interval of length ε to be exactly the same, no matter where the interval occurs during an execution. It follows that the operation of an RTOS with at least one failure mode cannot be considered simpliciter as a Poisson process[4]. This argument and associated considerations is presented in more detail in (Ladkin 2015).

6 Inappropriate Evaluation Attempts

The international standard for functional safety concerning systems which include electrical, electronic or programmable electronic (E/E/PE) components, IEC 61508, includes a short guide to statistical evaluation in Part 7, Annex D. The second sentence of this Annex suggests that the methods, the construal of software operation as a Bernoulli or Poisson process as above, can be used to evaluate software libraries, compilers, even operating systems.

We have heard anecdotes from industrial assessors of people trying to do just that. For example, a client C comes to an assessor. C proposes to use a real-time version of an operating system to run critical software with OS+Software having a safety requirement of SIL 3. C claims that the operating system has more than enough hours without failure, for a particular safety function, to satisfy the reliability conditions for SIL 3 for that function. In particular, the function is continuous (rather than on-demand) and C has detailed logs of the order of 10^8 failure-free (for this function) operating hours on the software, way more than required (see Table 2 above).

From the discussion above, besides the logs, C will have to show accurate registration of all failures in previous operation (and lack of such), with particular consideration given to possible failure masking in operation of such complex software. C will have to address the issue of versions: are all instances of the OS, whose behaviour has been registered, exactly the same version? Or are there "slight" divergences amongst them? And, finally, C must address the issue of whether the software operation is indeed memoryless in the required sense. These are all tricky issues, but only when they have been satisfactorily addressed can C "plug in the numbers" from Table 2 and draw the conclusion that OS operation fulfils the safety requirement. Then it is incumbent upon C to argue, and to ensure,

4 We note, though, that there are more complex ways to consider such software, some of which relax some of the assumptions of a Poisson process.

that inputs to the future system have the same statistical properties as the recorded inputs from the past.

It follows that "plugging in the numbers" is not at all easy.

7 Extending Evaluation Techniques

The question arises whether there are more subtle, and/or more widely applicable methods of evaluating software behaviour statistically than the straight conceptions as Bernoulli or Poisson processes. The answer is yes. However, their application is currently a matter on which expert statistical advice is needed.

In many of the more sophisticated methods, the software architecture plays a key role. Individual behaviours of individual components of the architecture are statistically assessed, and the results are combined into an assessment of the whole architecture. In many cases, the criteria for statistical assessment of the individual components may be relaxed, but the combined assessment still enables the key Bernoulli or Poisson mathematics to be used.

The second author has devised and used such techniques, see for example (Bedford 2001, Chapter 12). Colleagues have recently communicated the successful use of such techniques in evaluating critical software for rail applications (Schäbe 2015). There is a recent method for assessment of two heterogeneous channels, of which one may be "possibly perfect" (Littlewood 2012).

8 Conclusions: Why Use Statistical Evaluation?

In light of the discussion above, the reader may well wonder why anyone would bother with statistical assessment of software proposed for use in critical systems. There are many reasons. We give some.

Suppose a particular software-based system component has an adequate record of past use. Suppose, indeed, it appears informally to be the "best kit for the [new] task". The safety requirements for each critical system or component are individual, special to the specific system. The kit has been used before successfully to execute a specific function, and this function may be required for the proposed new use. However, it may well be that the safety requirements for previous use differ considerably from the safety requirement for the proposed new use. It may indeed be that documentation of the kit does not exist sufficient to justify its inclusion "as new" in the proposed new use. This may well be the case if the kit was not originally intended for safety-critical use, but has established its dependability through experience. It may also be the case that the assessment requirements in previous uses were less stringent than those for the proposed new use. This could occur for

two reasons: one, that safety standards have changed; two, as mentioned above, the safety requirement may be different.

If the kit is indeed the "best kit for the task", then there is good reason to use it. And there is good reason to be able to use statistical evaluation of previous use to make the case for its new use, if the statistics are available and adequate. A recent example of this industrial need has been communicated to the first author, but specific details are not available at time of writing (Kindermann 2015). We may speculate that such cases will arise more frequently, as more and more examples of relatively simple and reliable E/E/PE system components for specific critical functions come onto the market with time.

Acknowledgments We particularly thank Peter Bishop, Jens Braband, Wolfgang Ehrenberger, Rainer Faller, Andreas Hildebrandt, Bertrand Ricque and Hendrik Schäbe for detailed discussion of some of the issues arising here, and Bernd Sieker for valued help with the formatting.

References

ATSB (2007) In-flight upset, Boeing 777-200, 9M-MRG, 240 km NW Perth, WA. Investigation Number 200503722, 2007. Australian Transport Safety Board. https://www.atsb.gov.au/publications/investigation_reports/2005/aair/aair200503722.aspx , accessed 2015-02-03.

Bedford T, Cooke R (2001), Probabilistic Risk Analysis: Foundations and Methods, Cambridge University Press, 2001.

Bernoulli J (1713) Ars Conjectandi, Basel, 1713.

Ducklin P (2014) Anatomy of a "goto fail" - Apple's SSL bug explained, plus and unofficial patch for OS X! nakedsecurity blog, 2014-02-24. https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/ , accessed 2015-11-03.

IAEA (2002) Tutorial on the Accident at Three Mile Island, 2002. International Atomic Energy Authority. https://www.iaea.org/ns/tutorials/regcontrol/assess/assess3233.htm , accessed 2015-11-03.

IEC (2010) IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1-7. International Electrotechnical Commission.

Johnson C W, Holloway C M (2007) The Dangers of Failure Masking in Fault-Tolerant Software: Aspects of a Recent In-Flight Upset Event, in Proceedings of the 2nd IET International Conference on System Safety, IET 2007. Availablemfrom http://www.dcs.gla.ac.uk/~johnson/papers/IET_2007/Chris_Michael_Upset.pdf , accessed 2015-11-03.

Kindermann M (2015) personal communication.

Ladkin P B (2015) Some Practical Issues in Statistically Evaluating Critical Software, in System Safety and Cyber Security 2015, ISBN 978-1-78561-092-9 e-ISBN 978-1-78561-093-6, ISSN 0537-9989 Reference PEP....U, IET, 2015. http://www.rvs.uni-bielefeld.de/publications/ .

Littlewood B, Rushby J (2012) Reasoning about the Reliability of Diverse Two-Channel Systems in Which One Channel Is "Possibly Perfect", IEEE Trans. Software Engineering 38(5):1178-1194, 2012

Perrow C (1984) Normal Accidents: Living with High-Risk Technologies, Basic Books, 1984.

RAEng (2011) Global Navigation Space Systems: reliance and vulnerabilities, 2011. Royal Academy of Engineering, http://www.raeng.org.uk/publications/reports/global-navigation-space-systems , accessed 2015-06-25.

Schäbe H, Braband J (2015) . Basic requirements for proven-in-use arguments, preprint 2015. Available from http://arxiv.org/pdf/1511.01839v1.pdf , accessed 2015-11-06.

Siegrist K (2014) Virtual Laboratories in Probability and Statistics, University of Alabama at Huntsville, 1997-2014. http://www.math.uah.edu/stat/ , accessed 2015-06-25.

Thomas M (2011) personal communication.

U.S. Nuclear Regulatory Commission (2014) Backgrounder on the Three Mile Island Accident, 2014. http://www.nrc.gov/reading-rm/doc-collections/fact-sheets/3mile-isle.html accessed 2015-11-17.

Wikipedia (2015), Easter Eggs https://en.wikipedia.org/wiki/Easter_egg_(media) accessed 2015-11-17.