Practical Statistical Evaluation of Critical Software

Peter Bernard Ladkin, University of Bielefeld and Causalis Limited
Bev Littlewood, CSR, City University London

Version 4 of 2015-03-01

Abstract: In 2010, Rolf Spiker approached one of us with a query concerning the application of IEC 61508-7:2010 Annex D, on the statistical evaluation of software, which derived from a client [LL10]. We realised that Annex D gives sparse and sometimes misleading information to those who wish to evaluate critical software statistically, and embarked on a project to substitute Annex D with more helpful material. Through the work of one of us in standardisation committees, and the other in research into statistical evaluation of software, we encountered common assessment scenarios and their quandaries. We discuss them and the application of statistical methods, and conclude with a list of prerequisites to the application of Bernoulli/Poisson mathematics.

Section 0: Introduction

IEC 61508-7:2010 Annex D provides what it calls "*initial guidelines*" on a statistical approach (it says "*probabilistic approach*") to "*determining software safety integrity for pre-developed software based on operational experience*." It relies on the theory of Bernoulli and Poisson processes to derive the amount of operational experience needed to determine the reliability of software to some pre-determined confidence level. It says "*the techniques should be used only by those who are competent in statistical analysis.*" However, the information included is much less than basic course material [Sie97], which those "*competent in statistical analysis*" could be expected to know.

Anecdotal evidence supplied to the first author by assessor colleagues suggest there to be a sizeable number of system developers who wish to just "put in the numbers" that they have gathered on operational experience and draw a conclusion that their software is, say, "good for" Software Integrity Level (SIL) 2, SIL 3 or SIL 4 (defined in Tables 2 and 3 of IEC 61508-1:2010). This approach generally won't work, because the conditions for application of the Bernoulli or Poisson mathematics do/did not pertain. However, we do not believe that Annex 4 lays out clearly what tasks are necessary in order correctly to draw the desired conclusions.

The Notion of Software Safety Integrity

There appears to us to be some confusion concerning the notion of software safety integrity as used in IEC 61508:2010. It is defined to be "*part of the safety integrity of a safety-related system relating to systematic failures in a dangerous mode of failure that are attributable to software*" [IEC 61508-4:2010, definition 3.5.5]. Safety integrity is defined as "*probability of an E/E/PE safety-related system satisfactorily performing the specified safety functions under all the stated conditions within a stated period of time* ." [IEC 61508-4:2010, definition 3.5.4].

Although it is "safety" of which is being spoken, in fact the criteria for "safety integrity" in the IEC 61508 sense are criteria for the *reliability* of a safety function which is or may be implemented by software. The safety integrity of such software would thereby be the "*probability of [the software] satisfactorily performing the [....] safety functions [which it implements] under the stated [conditions and time]*". However, the definition of "software safety integrity" introduces the further notion of "*dangerous mode of failure*." It is perfectly possible for a safety function to fail to execute properly without the failure being "dangerous"; consider that the safety function could be triggered by a transient, which resolves itself benignly before the safety function has finished executing. For example, a sensed spike in temperature of a chemical reactor vessel might originate in an

anomalous transient of the sensor rather than any physical property of the reaction. A temperature-limiting safety function is executed successfully (say, a relief valve opened and emergency cooling activated) – or not, if the safety function fails. The failure or not of the safety function in this instantiation is not related to any "mode" of anything which is "dangerous", indeed everything was proceeding normally. Thus (a failure of) the safety function execution would not count towards assessing so-called "software safety integrity", but it would certainly count towards assessing the safety integrity of the software!

Furthermore, software by itself does not execute safety functions in their entirety, except possibly to execute in such a way as to keep the chip temperature within safe bounds. Software calculates values which are passed by chip-external communication channels to system actuators which execute some physical function which is the desired action. A safety function in the sense of IEC 61508 thus *always* essentially involves a combination of hardware, even if the success of the safety function is largely causally determined by the software operation.

Strictly speaking, it is not the operation of the safety function directly that is being evaluated when one speaks in IEC 61508 of "software safety integrity". It is the operation of that part of the safety function which is implemented in the software, and as noted above this will always be a strict part of the safety function. The safety function will have been assigned a SIL, and the part of the software which is causally involved in the execution of the safety function will inherit that SIL, because that is how software is assigned a SIL according to IEC 61508:2010. If the part of the software causally involved in the execution of the safety function is not sufficiently separable from the rest of the software (that is, the software is not sufficient modular), then that rest will also be assigned the SIL. Basically, the smallest demonstrably-modular unit of the software containing the relevant executive for the safety function is assigned the SIL of the safety function.

One apparent oddity of IEC 61508 is that, although a SIL is assigned to a safety function implemented by software/hardware units, and this comes with a reliability requirement, there is no requirement that the derived reliability shall be demonstrated of the software component of the safety function. Instead, various software development methods are recommended (or highly recommended) to be used. However, it has been shown by the first author that the software necessarily inherits a reliability requirement [Lad13.1]. In order to demonstrate, then, that the safety function attains its SIL, the involved software must be shown to satisfy that derived reliability requirement. The authors are aware of only very few instances in which this has been performed and adequately demonstrated to assessors, although we believe it should be routine.

Statistical evaluation, then, applies to any software (partially) implementing a safety function. But it is considered in Annex D only for pre-existing software. The mathematics is identical. What is special about pre-existing software is its history. It may not have been developed according to the precepts of IEC 61508; indeed, the first author is aware of process control software which is widely used in Europe but which has not been so developed. There is currently an IEC project, IEC/TS 61508-3-1, convened to defined requirements for establishing software as "proven in use" [IEC14]. The proposed requirements reference Annex D.

Is "Plugging in the Numbers" Acceptable?

One common assessment scenario is as follows. Suppose a client C comes to an assessor. C proposes to use a real-time version of an operating system to run critical software with 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 below). A similar

situation is considered in [Lad13.2].

The most obvious point is that the "numbers" as proposed are derived from the conception of the RTOS operation as a renewal process, a Poisson process. The only plausible construal of the RTOS as such a process is that one "run" of the Poisson process is defined by the boot-up of the RTOS and concluded with its shut-down. In between those times, all input – *all* input – to the RTOS must have been logged, and that sequence of consolidated inputs constitutes *one mathematical input* to the single run of the Poisson process. That will usually be a very large amount of time-stamped data.

The distribution of that input data must be determined. If the history of this RTOS is like the history of most RTOSs of the authors' acquaintance, the distribution will effectively be a subset of the entire possible input space: each possible (mathematical) input value will have been exhibited just once, or not exhibited at all - it is very unlikely that an exact sequence of inputs from boot-up to shut-down, as well as the relative-timing relations for time-dependent functions, will have occurred more than once. It is also combinatorially impossible that all possible inputs to the RTOS will have been observed; indeed, the subset will be sparse.

Relative timing of various inputs will be important. However, times in a log are likely to be actual clock times, the unique times at which input events took place. Such unique times would trivially rule out any identity to events in the future, thereby making the mathematics trivially inapplicable. Such real time notation must therefore be replaced with some sort of relative-time notation which can also occur in future use. How to do this is beyond the scope of this note.

The "numbers" for the Poisson process are valid only for the case in which the proposed future use of the RTOS has an *identical* input distribution to that cited in the operational history from which the numbers are derived. That is, exactly this sparse subset of all possible inputs must occur in the future. Quite how this could validly be demonstrated in any practical case remains a mystery to the authors – given the combinatorics it seems implausible.

A nice twist was recently recounted to the first author by Bernd Sieker [Sie15]. He observed that some portion of the working memory of the Linux kernel is not initialised on boot-up, in order to serve as a seed to the random-number generator (RNG). There is a story that someone observed that some part of working memory was not initialised, and wrote code to zero it out. The RNG stopped working effectively and thereby also certain important kernel functions. So the seed must be there. Since the (correct) OS thereby does not start from a defined initial state, a run from boot-up to shut-down cannot count as a run of a renewal process, since there is "memory" (both literally and figuratively) left over from a previous run. The mathematics of Poisson processes are prima facie inapplicable: the "numbers" don't work here.

Previous Work

Although there is substantial published work on the statistical evaluation of software, there is limited work on operational histories used in a "black box" fashion to estimate the future reliability of software in a new use. The two "canonical" papers are [BF93] and [LS93]. See [LS11] for the authors' assessment of progress (or not) over the subsequent two decades. [B02] and [BB03] derive some worst-case figures for the reliability of software given its previous operational history. [LW97] give some stopping rules for the case in which the operational history is largely controlled (e.g., it consists of extensive testing). [KHM01] considers the special case of statistical testing of Programmable Logic Systems (PLS) such as used widely in process control. [K05] uses Bayesian hypothesis testing to determine failure upper bounds from testing and also prior beliefs. She includes tables with representative numbers for the number of tests/amount of history required to draw the conclusions considered.

Some work combines operational history, including statistical testing, with other information to obtain more practical estimates of failure behavior than obtainable without such additional information. [B13] shows that "extraneous" information such as external mitigations to prevent an accident and the fact that software is corrected once failures are detected in operation can be used to derive an upper bound on the number of expected failures and accidents. [SP13] combines operational history (or test results) with evidence of fault-freeness to obtain more practical, but conservative, reliability predictions.

[BMGF06] gives some "graphical methods", namely BBNs, for assessing the reliability of safety functions implemented in software, but some access to the software architecture is required to use these methods; they are not strictly "black box". However, given its authorship the work can be taken to represent an insight into what IEC 61508-3 intends.

[BHS09] considers the concept of probability of failure on demand (pfd, see below) and compares with probability of failure per time period (hour) (pfh, see below), identifies some difficulties with using PFD as well as suggesting resolution of the difficulties through an alternative approach.

Section 1: General

This note provides some guidelines on the statistical evaluation of software safety integrity in the sense of IEC 61508:2010 for existing software which has a documented operational history. It has been found that statistical evaluation alone is rarely sufficient to establish adequate confidence in software safety integrity [LS93,BF93]. Usually, knowledge about the internal architecture (design) of the software and additional analysis of this architecture is also necessary, as well as knowledge about the requirements which the software was designed and implemented to meet, and any knowledge about how well the software meets these requirements. However, such knowledge about the internal workings of the software is rarely if ever sufficient to establish the required software safety integrity; and when extensive documented operational history (including tests) is available, statistical inferences may in principle be drawn.

The consensus statistical evaluation is that it may be used for software which performs functions which over time can be construed as a Bernoulli process or a Poisson process (renewal process). The mathematics of these processes may be found in standard texts, for example [Sie97]. Software which may be statistically evaluated by accepted methods is thus restricted to software whose operation constitutes one of these processes. Before software is so evaluated, it must be established that

- the operation of the software constitutes a Bernoulli process, respectively a Poisson process;
- failure detection is known to be perfect; all failures must be detected and there should be no false positives;
- the conditions for deriving conclusions about the reliability of the software through construing it as such a process, namely the information contained in the operational history along with information about its intended further use (see below), are rigorously fulfilled.

These three conditions result in documentation requirements additional to the operational history alone. A safety case including a statistical evaluation should therefore include documentation establishing the validity of the three conditions. Establishing that the conditions hold might not be easy, but it is necessary.

Statistical evaluation does not provide certainty about software properties. It provides a likelihood that certain properties pertain. This likelihood is stated as a level of confidence, or confidence level.

For example, statistical evaluation can lead to a level of confidence X that the software satisfies a safety integrity condition C. Typical levels of confidence are 95% and 99%[1], which represent an assessment that there is a more than 19-in-20, respectively 99-in-100 chance that C pertains, and equally a less than 1-in-20, respectively 1-in-100, chance that C does not pertain. If there is a (derived) safety requirement that C shall pertain, then statistical evaluation alone is insufficient, since certainty is not achieved.

Statistical evaluation could also be used to increase the confidence in failure-free operation of software over a suitably lengthy operational history. Such an operational history may be combined with statistical testing. There are significant obstacles to drawing effective conclusions from statistical testing alone for software expected to perform its safety function reliably in a SIL 2, SIL 3 or SIL 4 element[2] [LS93] [BF93].

Suppose we have software whose operation we have established constitutes a Bernoulli, resp. Poisson, process. We wish to ascertain that this operation attains a given reliability. IEC 61508:2010 assesses such reliability in terms of probabilities of failure per demand, *pfd*, as the appropriate measure for software which operates on demand; respectively failure per hour, *pfh,* as the appropriate measure for software with continuous function. But see [BHS09].

We consider the number of tests required to draw a conclusion, at a reasonable confidence level X, say X is 95% or X is 99%, that, on the given distribution of inputs, the software is reliable to a pfh of 10^{-x} or above. This number is many multiples of 10^x hours in the case that x=6 or higher; and mutatis mutandis for on-demand functions. This is illustrated in the tables below. This phenomenon was considered in depth by [BF93] and [LS93]. See also [LS11]. The table entries correspond to the SIL reliability requirements for elements (constituting typically hardware with operating software) from Tables 2 and 3 in IEC 61508-1:2010.

Acceptable probability of failure to perform design function on demand	Number of observed demands without failure for a confidence level of 99%	Number of observed demands without failure for a confidence level of 95%
$< 10^{-1}$	4.6×10^1	3×10^1
$< 10^{-2}$	4.6×10^2	3×10^2
$< 10^{-3}$	4.6×10^3	3×10^3
$< 10^{-4}$	4.6×10^4	3×10^4

Table 1: Example Operational-History Requirements for On-Demand Functions

Acceptable probability of failure to perform design function per hour of operation	Number of observed hours of operation without failure for a confidence level of 99%	Number of observed hours of operation without failure for a confidence level of 95%
$< 10^{-5}$	4.6×10^5	3×10^5
$< 10^{-6}$	4.6×10^6	3×10^6
$< 10^{-7}$	4.6×10^7	3×10^7
$< 10^{-8}$	4.6×10^8	3×10^8

Table 2: Example Operational-History Requirements for Continuously-Operational Functions

1 Note that we do not endorse these confidence levels generally; others might be appropriate for the case in hand.
2"Element" is a technical term of IEC 61508 whose definition may be found in IEC 61508-4:2010.

A general formula is evident from the tables. For continuously-operational software which must be reliable to a probability of failure per operational hour (pfh) of 10^{-x} , there must be a recorded series of (3×10^{x}) failure-free runs to attain a confidence level of 95% that the software fulfils the reliability condition. To attain a confidence level of 99% that the software fulfils the reliability condition, a recorded series of (4.6×10^{x}) failure-free runs is required.

A Methodological Caution

In order for the conclusions of a statistical evaluation to be valid,

- the conditions under which the mathematics of Bernoulli, resp. Poisson processes is valid must *rigorously* pertain;
- there must be very high assurance in the absence of confounding factors.

These conditions contain traps for the statistically-inexperienced engineer. To be reasonably assured that these requirements pertain usually requires some experience with practical statistical evaluation of very-high-reliability software. We illustrate such traps.

First, consider an example deriving from the mathematical requirements. One requirement (see below) is that the distribution of inputs in the intended future use of the software must be identical to the distribution of inputs in the historical operational records. Not merely similar; but identical. Here is an example of why this is necessary.

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 crucially will rarely be output which is appropriate for the critical function of the software – in other words, this critical function will routinely fail when the software is in DMM. The software is switched into DMM by a specific combination of input which is known to the developer of the software and its maintainers (henceforth "maintainer"), but may well not be known by the engineer who wishes to use the software in a critical application and is evaluating its use statistically (henceforth "client")[3] [Fal14].

The operational history of the software provided by maintainer to client will usually not contain examples of the operation of the software in DMM, for that would not be deemed relevant to the operational use of the software, as a matter of common sense. Suppose that the client can ensure exactly this distribution of inputs in hisher application, with the exception of an infrequent but not rare input *which is exactly the combination to send the software into DMM*. The software will fail, infrequently but not rarely, in the new application. The statistical evaluation will thereby not have been an accurate guide to the future behavior[4]. However, the input distribution will have been very similar to the historical distribution provided for evaluation – indeed identical in all but one point! It follows that "identical" in the mathematical requirements for application of the Bernoulli/Poisson mathematics rigorously means "identical".

Now for an example of the subtlety of confounding factors.

---

3 How to deal effectively with the evaluation and operation of critical software with DMM is beyond the scope of this note. It will necessarily involve both safety and security considerations. See [Lad15].

4Note that IEC 61508-1:2010 Clause 7.4.2.3 requires security to be considered. If the information on DMM is available to a malicious agent with access, say an "insider" with malicious intent, then it is easy to see how such an agent could cause almost-continual loss of intended function, known in computer-security terminology as denial of service (DoS). It is beyond the scope of this note to consider such security issues.

Software for a moderately critical application was being assessed for environmental dependencies. 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 [Tho11] [RAEng11].

These examples show that dealing with the rigor of the mathematical conditions for evaluation and the subtlety of analysis required to determine the absence of confounding factors are significant engineering skills required for statistically evaluating high-reliability software accurately.

Section 2 Mathematical Background to Reliability Evaluation

This section explicates some of the more simple mathematics of Bernoulli and Poisson processes, and the conditions which must pertain to be able to draw conclusions about future behavior from past operational experience.

We consider software as a data-transformation method. A program *P* reads data *d* as input and generates output *outP(d)*. We consider here only deterministic *effectively-stateless* programs (see below) which do not combine some part of their internal state with the input *d* in order to generate their output.

Such programs P are those for which the output *outP(d)* is a pure function of *d* and nothing else: there is only one value(-tuple) *outP(d)* corresponding to input *d*.

- Determining whether a real program, as binary machine code running on electronic hardware, is deterministic and effectively-stateless is a extensive task which requires documented assurance.

We assume that the range of input to *P, D*, is known: that is, *d* \belongsTo *D* necessarily. *D* contains not only input data anticipated by the designers and programmers, but is taken here also to contain anomalous data that might be generated by faults in the larger system and passed by that system on to *P*. (Good system design would lead to attempting to identify such anomalous data in the wider system and trapping it before it gets to P. However, there might be such anomalous data that nevertheless “makes it through”.)

Output *outP(d)* may be correct, according to what *P* is supposed to do, or it may be incorrect. We are concerned here with supposedly highly-reliable programs *P* so we hope that the case of incorrect output will be rare, and we wish to estimate the amount of evidence we need to gather in order to become suitably confident that P is sufficiently reliable.

We can define a new function as follows:

CorrP(d) = 1 if *outP(d)* is correct, where *d* \belongsTo *D*
CorrP(d) = 0 if *outP(d)* is incorrect, where *d* \belongsTo *D*

That the failures of software are systematic is represented here precisely by the fact that *CorrP(d)* is a pure function of d alone. Input values are discrete, represented by some number of bits, and the number of bits that can be used to give input is limited. Hence D is a finite, although usually very, very large, domain.

As P runs, it takes inputs and generates outputs. We assume that the inputs are distributed according

to a distribution *Distr(D)* which specifies how (relatively) frequently a given input *d* is presented to *P* compared with other inputs. The probability of failure on demand (pfd) of the program *P* is the probability that an execution of *P* on an input selected randomly using this distribution is incorrect.

As program *P* runs, taking a succession of inputs d according to a probability given by *Distr(D)* and generates output *outP(d)*, the value of the corresponding function *CorrP* generates what is called a *Bernoulli process*. A Bernoulli trial is an action, such as the toss of a coin, or the random selection of a ball from an urn containing balls of two colors, which can have one of just two outcomes (heads/tails, respectively black/white), and in which the outcome is independent of other similar actions (a previous coin toss; a previous ball selection) [Fel68]. A Bernoulli process is a sequence of Bernoulli trials with fixed outcome probabilities throughout: say, a sequences of tosses of the *same* (fair or biased) coin; a sequence of random selections from the urn, in which each ball is replaced after selection (not replacing the ball would change the selection probability for the next selection!). The key property is that the outcome of a trial in a Bernoulli process is independent of the history of previous trials in the process. The outcomes can be denoted *success/failure,* or *0/1*.

The Bernoulli process associated with program reliability is the sequence of 0's or 1's associated by *CorrP* with the successive inputs as the program *P* runs. The key assumptions here are:

- *outP(d)* is a function solely of *d,* using no other values extraneous to *d*; and
- each input-output combination *d-outP(d)* which occurs is stochastically independent of previous *d-outP(d)* combinations which have already occurred, and
- the probability of failure of *P* is constant.

The random variable of most interest in software reliability is the number of demands to first failure (or, equivalently, the number of demands between successive failures). Let this be S. If S is k, then there are (k-1) successes followed by a failure. Let the probability of failure be p. The trials are by hypothesis all independent events. So we may multiply these probabilities and S has a geometric distribution:

$$Prob(S=k) = (1-p)^{(k-1)}.p$$

This has mean *1/p*, which is the expected number of demands to first failure. Note that this will generally not be an integer despite its name – it should rather be interpreted as the average over many runs of this "experiment".

[Fel68] notes (p.146) the traditional example of Bernoulli processes, arising from Bernoulli trials, namely successive tosses of a coin. A toss of a coin comes up either heads or tails, the two possible outcomes, and the probability of success (say, heads) is the same if the same coin is being used for all the trials (the coin may be biased). The original posthumous manuscript of Bernoulli considers an urn model [Ber1713], also see [Lad15.2]. These two traditional examples of Bernoulli processes are mainly of historical or pedagogic interest.

For continuous-time operation, we need a continuous-time equivalent of the Bernoulli process, which is the Poisson process. The Poisson process has similar properties, but in continuous time, to the Bernoulli process: the number of events in a certain time interval is statistically independent of what happens outside this interval: its properties are homogeneous over time - they do not change as time passes.

Of interest here is the distribution of the (continuous) random variable: time to first failure (equivalently, inter-failure time). It is the exponential distribution with probability density function

$\lambda.e^{-\lambda.t}$ where $t > 0$

The mean value of this function, $1/\lambda$, is the mean time to first failure (MTTF).

The most important property of each of these processes is that they are *memoryless* [RosWei]. Memorylessness means that the probability of a future event (say, the failure of a function) seen from any time point is the same. If the MTTF of a process is *X* hours, then one hour later it will still be *X* hours (rather than, say, *X-1* hours). Memoryless processes are the most extreme examples of renewal processes, since they "renew" themselves at each point (at each demand in the case of Bernoulli; at each time point in the case of Poisson). This means that, for example, in a Poisson process $1/\lambda$ is not just the mean time to first failure, but also the mean time between failures (a similar observation holds for the Bernoulli process). It can easily be shown that these distributions, geometric for Bernoulli, exponential for Poisson, are precisely *the* memoryless functions .

The Poisson process can be shown to be the limiting case of the Bernoulli process when the probability of failure becomes very small and the number of demands very frequent. For our purposes here, it is best just to keep in mind that the Bernoulli process models successes/failures of discrete demands (on-demand functions) and the Poisson process models successes/failures of continuously-operating functions.

The mathematics of Poisson processes and the relation to Bernoulli processes is standard textbook material [Fel68,Sie97,BedCoo01,Bir14]. [BedCoo01] also includes a chapter on software reliability which surveys inter alia the modelling approach of the second author.

On-demand and Continuous Functions

Some safety functions are appropriately characterised as discrete: they are invoked occasionally, such as reactor scram functions or overcurrent-protection functions; or they are invoked regularly, such as garbage-collection to maintain available working-memory space for critical functions.

Some safety functions are appropriately characterised as continuous. Critical-control-system outputs, for example, are often a combination of control function with safety function.

Some simple devices combine both continuous function with on-demand function. Consider a household fuse. It has the continuous function of enabling electric current to flow in a circuit within suitable parameters, and the on-demand function of breaking the circuit when these parameters are exceeded. These two separate functions are implemented in the same device using exactly the same physical mechanism; in other words, the dual functions supervene on the single mechanism. The continuous function is not generally a safety function (unless the circuit energises critical equipment), whereas the on-demand function is a function whose role is safety (although it may not necessarily be a safety function in the sense of IEC 61508-1:2010).

When implemented by software on digital hardware, the hardware-level distinction is one mainly of frequency of assessment. A "continuous" control function will sample parameters and react to them just like an on-demand function, but at a rate of some hundreds of Hz (1 Hz = 1 sample per second), whereas an on-demand function will be invoked at a frequency of small fractions of a Hz. The Bernoulli process is appropriate for representing the reliability of on-demand functions, and the Poisson process more appropriate for representing that of continuous functions, in which the parameter is more conveniently given as a time period rather than the number of trials (there is a functional relationship: the time period is simply the number of trials divided by the sample rate).

A Bernoulli process is approximated by a Poisson process when the time associated with each

"demand" is very small and the probability of a failed demand is very small. For software, it has been pointed out that a "continuous" process is in fact implemented as a discrete process with a very fast sampling rate (typically hundreds of Hz). This is the way in which digital feedback-control systems operate.

Assessment of Failure-Freeness

Suppose one has observed *N* invocations of an on-demand function, or alternatively a time period *T* (*N = T*demand-rate*) of successful operation without failure of a continuous function. One wants to know with a certain degree of confidence that the probability of failure, *p*, of the function is sufficiently small, according to the specification of the SIL level. The question is statistical: one has a certain series of observations, one knows "for certain" that one is observing a Bernoulli, resp. a Poisson process and one wants to know what the likelihoods are that one is observing such a process with an appropriate *p* determined by the SIL level. The mathematics is given by [LS93]. For a derivation from first principles see [LadS11] for a Bayesian approach and [BF93] for a frequentist approach.

Note that this assessment talks about real failures, not observations of failures. As we have said, in order for the mathematics of Bernoulli and Poisson processes to be applied, real failures have to be considered. Thus

- There must be perfect failure detection: all failures of the software must be (have been) observed and recorded;
- In particular, failures may be masked by other failure phenomena. These masked failures must also somehow be observed and recorded.[5]

Section 3. Renewal Processes and Effectively-Stateless Software Functionality

The assumptions that lead to the characterisation of the statistical behavior of software malfunction as described by a Poisson process are

1. That the SW is "stateless", that is, that the behavior of the SW on a given input E is dependent only on the value E and not on the internal state of the SW at the time E is input;
2. That the likelihood of a malfunction remains constant over time

Condition 1 in particular may give rise to the observation that relatively little safety-related software, if any, fulfils it literally. Often, however, the internal state is used to record some history of inputs in order to determine a change which is significant for the function of the software. Consider, for example, a temperature-rise-control function *TC*. *TC* monitors temperature and, if a temperature is rising too fast, executes a mitigating function. The means whereby *TC* determines that the temperature is rising too fast can be through a sequence of timestamped temperatures (delivered in a suitably timely fashion). Internal state (memory) is used in *TC* to record the sequence of temperatures and times for determination of the temperature profile.

TC is not literally "stateless"; indeed internal state is used essentially. Its behavior, however, does exhibit a Bernoulli, respectively Poisson process. Consider factoring the function which *TC* performs into two. The first function *DT* aggregates temperature over time, and determines the rise in temperature over discrete time periods: it is a discrete approximation to the first derivative of temperature with respect to time. When this value exceeds a threshhold, it signals a second, protection/mitigation process *Prot* to start. *Prot* is stateless and thus embodies a Bernoulli process. *TC* consists of the pipeline *DT* » *Prot* and the Bernoulli-process assumptions and mathematics apply

5 A discussion of failure masking and resolution is beyond the scope of this note.

because *Prot* embodies literally a Bernoulli process (resp. a Poisson process if considered over time rather than through demand-invocations).

Other examples of such renewal processes are some protection systems in nuclear power plants. Such a system starts from a given "initial" state and reacts to parameters over a short time period, in order to execute its protection function. The time period over which input is aggregated is shorter than the time period between invocations of the software, and the software is "reset" to the initial state after execution of its function. Different invocations of this function, and thereby failure or success per invocation, are independent. Furthermore, there is some constant probability of failure, dependent on the exact sequence of inputs per invocation and the distribution of those inputs. It is necessary to be considered as a renewal process and be subject to the Bernoulli/Poisson analyses that no vestige of internal state is present from one invocation of the function to the next. For many protection systems, this is indeed the case.

One class of software which fulfils the general conditions is that which implements on-demand functions and which is returned to a defined initial state ("reinitialised") after each invocation, such as protection functions. There are specific verification conditions which must be assured:

- that such SW starts each time in a pre-defined initial state must be proven (if it is always the case), or alternatively
- it must be recognised without exception when the SW is not in the initial state, and such exceptional invocations omitted from the statistics and the statistical evaluation of the software.

Another class of functions which fulfil the general conditions are cyclic functions: software functions which are reinitialised at predetermined times or at other points (say, dependent on internal state). The sequence of inputs from an initial state up to the point at which the software is reinitialised form one mathematical input, namely a sequence, which can be seen as follows.

We generalise from the *DT* » *Prot* example above. Say a cyclic function F receives inputs E_1, E_2, ... E_n during a cycle. Consider a program P which takes as input the sequence $< E_1, E_2, ... E_n >$ and which delivers these elements in sequence to F on demand. Suppose also that it is known that P is correct (say, P has been formally verified). Then the function P » F, which consists the pipeline P to F, is a "stateless" function in the required sense. Its failure behavior is identical to that of F. It follows that F itself may be taken to execute as a renewal process.

Notice that the requirement that P is correct may be relaxed in a similar way to the protection-function example. If

- it is recognised without exception when P fails, and these P-failure occasions are omitted from the statistical evaluation, and
- the failures of P which enter the statistical evaluation are known to be statistically independent of failures of F

then P » F with the statistics filtered by P-failure forms a renewal process and thus does F.

It will not usually literally be the case that failures of P are statistically independent of failures of F, without qualification. There always exist common-cause failures of both, say when the equipment on which they run is obliterated by explosion or other disturbance. The condition that

- common-cause failures of P and F are exceptionlessly recognised and eliminated from the statistical evaluation

allows common-cause failures of P and F to be eliminated from the statistical evaluation, and the remaining failures of P are thereby statistically independent of failures of F and may be filtered out of the statistical evaluation.

Possibly, however, a failure of P may mask a failure of F, of which need to know: evaluation concerns the number of runs *without* a failure of F, in order to draw valid conclusions about the reliability of F. If a run involved a failure of P and was thus filtered out, but the resulting input (or lack of input) to F triggered a failure of F, then that would be a counterexample to a possible reliability claim for F – we must not filter out F-failures. If the failure of P is *transparent*, in the sense that we can reconstruct the input (or lack of it) to F that occurred even though P failed, then the failure of F on that (real) input may be considered part of the statistical evaluation of F. Thus,

- if failures of P are transparent, then the behavior of F on runs in which P failed may be considered part of the statistical evaluation of F;
- however, it should be taken into account that the failures of P, when corrected using the transparency property, yield a distribution of input to F which may be different from the distribution of input to P » F on which F is being evaluated.

This short discussion suffices to indicate that the restriction to "stateless" programs is not such a severe restriction as it appears. This mathematics applies also to "*effectively-stateless*" functions such as *DT* » *Prot* and P » F. It does, however requires some degree of experience to perform accurately such transformations as above.

Section 4 Summary of Evaluation Conditions

In the discussion above, a number of assessment conditions are enumerated. We collect them here.

- The operation of the software must be shown to constitute a Bernoulli process, respectively a Poisson process;
- The conditions under which the mathematics of Bernoulli, resp. Poisson processes is valid must rigorously pertain. That this must be shown is prerequisite for deriving conclusions about the reliability of the software through construing it as such a process, namely the information contained in the operational history along with information about its intended further use (see below), are rigorously fulfilled. These conditions include those mentioned below;
- The real program, as binary machine code running on electronic hardware, must be shown to be deterministic and effectively-stateless;
- The likelihood of a failure must be shown to be constant over time;
- There must be perfect failure detection: all failures of the software must be/have been observed and recorded;
- In particular, failures may be masked by other failure phenomena. These masked failures must also be/have been observed and recorded;
- That such SW starts each time in a pre-defined initial state must be shown (if it is always the case), or
    - Alternatively, it must be recognised without exception when the SW is not in the initial state, and such exceptional invocations omitted from the statistics and the statistical evaluation of the software.
- There must be very high assurance in the absence of confounding factors, such as unremarked environmental parameters, in the operational-history logs.

Acknowledgements

Many thanks to Bernd Sieker for the example in Section 0, and to Rainer Faller and Martyn Thomas for those in Section 1. as also to Michael Kindermann for comments on a draft version of Section 2. Thanks in general to the members of the German committee on "Safe Software", DKE AK 914.0.3, with whom the first author has had many discussions over the previous five years, and whose concerns prompted this study.

References

[BedCoo01] T. Bedford and R. Cooke, Probabilistic Risk Analysis: Foundations and Methods, Cambridge University Press, 2001.

[Ber1713] J. Bernoulli, Ars Conjectandi, Basel, 1713.

[Bir14] A. Birolini, Reliability Engineering: Theory and Practice, Seventh Edition, Springer-Verlag, Heidelberg, 2014.

[B02] P. G. Bishop, Rescaling Reliability Bounds for a New Operational Profile, in Proceedings, International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy, 22-24 July, 2002, ACM Software Engineering Notes 27(4):180-190, ACM 2002.

[B13] P. Bishop, Does Software Have to Be Ultra Reliable in Safety Critical Systems?.in Computer Safety, Reliability, and Security (Safecomp 2013. Springer Berlin Heidelberg, 2013. 118-129.
in F. Bitsch, J. Guiochet, and M. Kaâniche, (eds.), Computer Safety, Reliability, and Security, proceedings of SAFECOMP 2013, Lecture Notes in Computer Science, Vol. 8153, Springer-Verlag, Heidelberg, 2013

[BB03] P. G. Bishop and R. E. Bloomfield, Using a Log-normal Failure Rate Distribution for Worst Case Bound Reliability Prediction, *i*n Proceedings *fourteenth International Symposium on Software Reliability Engineering (ISSRE '03)*, pp. 237-245, 17-20 November, 2003, Denver, Colorado, USA, IEEE, 2003.

[BHS09] J. Braband, R. Vom Hövel and H. Schäbe, Probability of Failure on Demand – the why and the how, in B. Buth, G. Rabe and T. Seyfarth (eds.), Computer Safety, Reliability and Security, proceedings of SAFECOMP 2009, Lecture Notes in Computer Science 5775, Springer-Verlag, Heidelberg, 2009.

[BMGF06] M. Brito, J. May, J. Gallardo and E. Fergus, Use of graphical probabilistic models to build SIL claims based on software safety standards such as IEC 61508-3, in F. Redmill and T. Anderson (eds.), Developments in Risk-based Approache to Safety, Proceedings of the Fourteenth Safety-critical Systems Symposium, Bristol, UK, 7-9 February 2006, Springer-Verlag, London 2006.

[BF93] R. Butler and G. Finelli, The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software, IEEE Transactions on Software Engineering 19(1):3-12, 1993.

[Fal14] R. Faller, personal communication, December 2014.

[Fel68] W. Feller, An introduction to probability theory and its applications, Volume 1, Third edition, John Wiley and Sons, 1968.

[IEC10] International Electrotechnical Commission, IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1-7, 2010.

[K05] M. Kristiansen, Finding upper bounds for software failure probabilities – experiments and results, in R. Winther, B.A. Gran and G. Dahll (eds.), Computer Safety, Reliability and Security, proceedings of SAFECOMP 2005, Lecture Notes in Computer Scienc 3688, Springer-Verlag, Heidelberg, 2005.

[KHS01] S. Kuball, G. Hughes, J. H. R. May, J Gallardo, A. D. John, and R.B. Carter, The effectiveness of statistical testing when applied to logic systems, in U. Voges (ed.), Computer Safety, Reliability and Security, proceedings of SAFECOMP 2001, Lecture Notes in Computer Science 2187, Springer-Verlag, Heidelberg, 2001.

[IEC14] International Electrotechnical Commission, Technical (Sub)committee SC65A, Project IEC/TS 61508-3-1 Ed. 1.0: Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 31: Software requirements - Reuse of pre-existing software elements to implement all or part of a safety function. Details available through http://www.iec.ch .

[Lad08] P. B. Ladkin, An Overview of IEC 61508 on E/E/PE Functional Safety, available at http://www.causalis.com/IEC61508FunctionalSafety.pdf , Causalis Limited, 2008.

[Lad13.1] P. B. Ladkin, IEC 61508 Case Study, RVS White Paper 03, 20 February 2013. Available from http://www.rvs.uni-bielefeld.de/publications/WhitePapers/rvsIEC61508CaseStudy.pdf

[Lad13.2] P. B. Ladkin, Assessing Critical Software as “Proven in Use”: Pitfalls and Possibilities, RVS White Paper 4, 14 June 2013. Available from http://www.rvs.uni-bielefeld.de/publications/WhitePapers/LadkinPiUessay20130614.pdf

[Lad15.1] P. B. Ladkin, An example of a safety-critical element with deliberately unreliable function, RVS White Paper 8, 1 January 2015. Available from http://www.rvs.uni-

bielefeld.de/publications/WhitePapers/LadkinFallerExample20150101.pdf

[Lad15.2] P.B. Ladkin, Software, the Urn Model, and Failure, preprint, RVS Group, University of Bielefeld, February 2015.

[LL10] P. B. Ladkin and B. Littlewood, Discussion on Part 7, Annex D, Working Document DKE 914.0.3_2010-0010, Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE , Frankfurt 2010.

[LadS11] P. B. Ladkin and B. M. Sieker, Safety of Computer-Based Systems, draft textbook, Chapter 16 (in German), RVS 2011. Available from http://www.rvs.uni-bielefeld.de/publications/books/ComputerSafetyBook/index.html

[LPS00] B. Littlewood, P. Popov and L. Strigini, Assessment of the reliability of fault-tolerant software: A Bayesian approach, in F. Koornneed and M. van der Meulen (eds.), Computer Safety, Reliability and Security, proceedings of SAFECOMP 2000, Lecture Notes in Computer Science 1943, Springer-Verlag, Heidelberg, 2000.

[LS93] B. Littlewood and L. Strigini, Validation of ultra-high-dependability for software-based systems, Communications of the ACM 36(11):69-80, 1993.

[LS11] B. Littlewood and L. Strigini, “validation of ultra-high dependability...” - 20 years on, Safety Systems 20(3):6-10.

[LBB11] B. Littlewood, P. Bishop, R, Bloomfield, A. Povyakalo and D. Wright, Towards a formalism for conservative claims about the dependability of software-based systems, IEEE Transactions on Software Engineering 37(5):708-171, 2011.

[LW97] B. Littlewood and D. Wright, Some conservative stopping rules for the operational testing of safety-critical software, IEEE Transactions on Software Engineering 23(1):673-683, 2011.

[RosWei] A.M. Ross and E. W. Weisstein Entry: Memoryless. In *MathWorld*--A Wolfram Web Resource, no date, accessed 2015-02-26. Available at http://mathworld.wolfram.com/Memoryless.html

[RAEng11] Royal Academy of Engineering, Global Navigation Space Systems: reliance and vulnerabilities, March 2011. Available at http://www.raeng.org.uk/publications/reports/global-navigation-space-systems .

[Sie97] K. Siegrist, Virtual Laboratories in Probability and Statistics,University of Alabama at Huntsville, 1997-2014. WWW only, available at http://www.math.uah.edu/stat/ .

[Sie15] B. M. Sieker, personal communication, 12 February, 2015.

[SP13] L. Strigini and A Povyakalo, Software fault-freeness and reliability predictions, .in F. Bitsch, J. Guiochet, and M. Kaâniche, (eds.), Computer Safety, Reliability, and Security, proceedings of SAFECOMP 2013, Lecture Notes in Computer Science, Vol. 8153, Springer-Verlag- Heidelberg, 2013.

[Tho11] M. Thomas, personal communication, 2011.