Software Reliability and the Misuse of Statistics

Dewi Daniels[1], and Nicholas (Nick) Tudor[2]

1. Software Safety Limited, Trowbridge, UK
2. D-RisQ Ltd, Malvern, UK

Abstract

Many papers have been written on software reliability. The claim is made that failures of software-based systems occur randomly and that statistical techniques used to predict random hardware failure rates can also be used to predict software failure rates. This claim has not been challenged in any academic papers, though it is treated with suspicion by many practising engineers. As a result, the applicability of these statistical techniques has been accepted in some standards, such as IEC 61508, but has been rejected in others, such as RTCA/DO-178C. It is more important than ever to understand whether this claim is true. There is strong lobbying from industry to allow software not developed to any standard to be used for safety critical applications provided it has sufficient product service history. The European Union Aviation Safety Agency (EASA) is promoting dissimilar software in the belief that using two or more independent software teams will deliver ultra-high levels of software reliability. Software defects are different from random hardware failures and need to be treated differently. This paper argues that the techniques used for statistical evaluation of software make unwarranted assumptions about software and lead to overly optimistic predictions of "software failure rates". This paper concludes that many software reliability models do not provide results in which confidence can be placed. Instead, this paper proposes an alternative way forward that does provide evidence that software is safe for its intended use before it enters service.

1 Introduction

Many papers have been written on software reliability, including (Littlewood and Verrall 1973) and (Ladkin and Littlewood 2016). The claim is made that the same statistical techniques used to predict random hardware failure rates can also be used to predict software failure rates. While (Ladkin and Littlewood 2016) admits that "*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*", the authors go on to claim that, "*there is uncertainty about when a program will receive an input that will cause it to fai*l". They therefore conclude that software failures form a stochastic (random) process and that probabilistic models can be applied to software failures. A view is that as software is a component of a system and all components contribute to the reliability of the system, that software therefore has to have a failure rate. This reflects the demands of some reliability engineers using techniques such as Failure Mode and Effects Criticality Analysis (FMECA) and therefore also demands to have failure mechanisms for software.

The applicability of these statistical techniques is accepted by some industry standards, such as IEC 61508-3 (IEC 2010a), but rejected by others, such as RTCA/DO-178C (RTCA 2011).

© Software Safety Limited and D-RisQ Ltd 2022.

There is pressure from industry to allow commercial-off-the-shelf (COTS) and open source software to be accepted for use in safety critical systems based on product service history. For example, RTCA SC-240/EUROCAE WG-117 has been directed by RTCA Program Management Committee (PMC) and the EUROCAE Council (EC) to develop guidance on the integration of COTS, open source, and service history into software for airborne systems (RTCA 2021).

Furthermore, the European Union Aviation Safety Agency (EASA) is advocating dissimilar software development in the belief that the use of multiple, independent software teams will prevent common mode software failures (EASA 2014).

2 Hardware Reliability Modelling

The mechanical domain successfully uses statistical models. However, that world is markedly different from the one occupied by software. There are no similar failure mechanisms for software: there is no corrosion, fatigue, or fungus, let alone vagaries in manufacturing, which engineers account for by building in safety factors. It has been shown over many years that the use of modelling and statistics in techniques such as FMECA give well understood and predictable outcomes. These are models built up from knowledge of individual component failure mechanisms, their likelihood, and contribution to the overall system reliability. Looking for the contribution from the software domain, some mechanical engineers wish to have a number for reliability of the software component. Unfortunately, a response that it is either a '1' or a '0', does not tend to satisfy, so seeking a statistical approach to determining the contribution of software to overall system reliability is understandable perhaps, if one thinks of software as a component of a system, like all other components; unfortunately, it is not like other components, and needs to be treated differently, as we shall now discuss.

It should be noted that complex electronic hardware (CEH), on the other hand, can fail in very similar ways to software. Indeed, writing in VHDL (VHSIC [Very High-Speed Integrated Circuits] Hardware Description Language) is very similar in many ways to software programming. The issues raised in this paper could therefore be applied equally to CEH.

3 Software Reliability Modelling

3.1 Software Reliability in the Standards

3.1.1 Software Reliability in RTCA/DO-178C

RTCA/DO-178C (RTCA 2011) warns, "*Development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software reliability rates based on software levels cannot be used by the system safety assessment process in the same way as hardware failure rates*". Concerning software reliability models, it states, "*Many methods for predicting software reliability based on developmental metrics have been published, for example, software structure, defect detection rate, etc. This document does not provide guidance for those types of methods, because at the time of writing, currently available methods did not provide results in which confidence can be placed*".

3.1.2 Software Reliability in IEC 61508

IEC 61508-3 (IEC 2010a) recommends probabilistic testing (highly recommended for software aspects of system safety validation at SIL 4). IEC 61508-7 (IEC 2010b) recognises that it is very difficult to demonstrate ultra-high levels of reliability using these techniques. (Littlewood and Strigini 1993) acknowledge that, where ultra-high dependability is required for software-based systems, it is not possible to confirm that a sufficiently high, numerically expressed dependability has been achieved.

Annex D of (IEC 2010b) describes a probabilistic approach to determining software integrity for pre-developed software. It is noted that Annex D is informative rather than normative. Table 1 below shows the number of failure-free demands experienced or hours of failure-free operation needed to qualify for a particular safety integrity level according to Annex D.

Table 1 ~ Necessary History from IEC 61508-7 Annex D

SIL	Low demand mode of operation	Number of treated demands		High demand or continuous mode of operation	Hours of operation in total	
	(Probability of failure to perform its design function on demand)	$1-\alpha = 0.99$	$1-\alpha = 0.95$	(Probability of a dangerous failure per hour)	$1-\alpha = 0.99$	$1-\alpha = 0.95$
4	$\geq 10^{-5}$ to $< 10^{-4}$	4.6×10^5	3×10^5	$\geq 10^{-9}$ to $< 10^{-8}$	4.6×10^9	3×10^9
3	$\geq 10^{-4}$ to $< 10^{-3}$	4.6×10^4	3×10^4	$\geq 10^{-8}$ to $< 10^{-7}$	4.6×10^8	3×10^8
2	$\geq 10^{-3}$ to $< 10^{-2}$	4.6×10^3	3×10^3	$\geq 10^{-7}$ to $< 10^{-6}$	4.6×10^7	3×10^7
1	$\geq 10^{-2}$ to $< 10^{-1}$	4.6×10^2	3×10^2	$\geq 10^{-6}$ to $< 10^{-5}$	4.6×10^5	3×10^6

NOTE 1 $1-\alpha$ represents the confidence level.

NOTE 2 See [IEC 61508-7] D.2.1 and D.2.3 for prerequisites and details of how this table is derived

3.2 Applicability of Models

While models can be useful in various circumstances, users have to be careful to ensure that the chosen model is applicable and that the limitations of the model are understood when drawing conclusions. (Mandelbrot and Hudson 2004) predicted the stock market crash that occurred in 2008. (Mandelbrot and Hudson 2004) claimed that the mathematical models used were flawed and that it was mistaken to assume that the normal distribution was a useful model for tracking price changes in the stock markets. Most economists responded that independence and normality are just assumptions that help simplify the mathematics. However, the inappropriate application of the normal distribution underestimated the probability that many borrowers would default on their subprime mortgages at the same time.

One must be careful of the "tail" in the models. If the application of the model underestimates the probability of an extreme event, then the consequences of decisions made to defend against such an event could be far reaching. (Mandelbrot and Hudson 2004) claimed that stock market prices follow a power law rather than the normal distribution. Power laws have fatter tails than the normal distribution. A statistical model based on the normal distribution would have underestimated the probability of unlikely events such as stock market crashes.

There are many examples of incorrect application of modelling to problems. Indeed, a quote generally attributed to George Box is, "All models are wrong, but some are useful", which tries to explain that models are not the real artefact but may nevertheless be useful. The scientist, engineer, forecaster must therefore ensure that the assumptions behind the use of the model are the right assumptions to be making and which must be understood before valid conclusions can be made. Without this understanding, a false premise can lead to almost any conclusion one would like, or indeed, not like.

3.3 Assumptions

3.3.1 Preamble

Many software reliability models assume that software execution is a Bernoulli process. It is therefore assumed that:

1. Executing a software program results in one of two outcomes, which we term Success or Failure; and
2. The probability of Success is the same every time the software is executed.

The first of these assumptions seems a reasonable assumption. The software either produces the correct output or it does not. We accept it is not always easy to determine whether an output is correct or not, but we will assume for now that we can do this. The second of these assumptions is an unwarranted assumption in most cases, as we explore in the following examples.

3.3.2 Example 1: State

Most software programs contain state. This could be in the form of a variable stored in Random Access Memory (RAM), a record stored on a Solid-State Drive (SSD), a Hard Disk Drive (HDD), in the Cloud, or even as the contents of a memory cache. State essentially introduces bias into the system. A fair coin or die does not have "memory". Software state is worse than a biased coin because the bias could change from one trial to the next. Suppose a software program increments an unsigned 16-bit counter. When the counter reaches 65,535, it wraps around to zero, causing the program to produce the wrong output. The probability of success is not the same every time the software is executed. Assuming there are no other defects, the probability of success the first 65,535 times the software program is executed is one; the probability of success the $65{,}536^{th}$ time the software program is executed is zero. If we had executed the software program 65,535 times and observed it produce the correct output each time, we would have a high degree of confidence that the software program is correct, yet it will fail the very next time it is invoked.

The presence of state means that whether invoking a software program results in success or failure depends not just on the inputs presented at that time, but on previous invocations

of the software. If you are worrying about cache contents, then it also depends on other software that is running on the same hardware (even if we're using time slicing!). If the software halts the processor or enters an infinite loop, then all subsequent invocations will fail until the software is reset. More subtly, a previous invocation of the software could have resulted in the state changing in such a way that a future invocation of the software will fail. This means that the probability of success is not the same every time the software is executed.

3.3.3 Example 2: Environment

The mathematical models assume that the probability of success is the same every time the software is executed. How can the probability of success be the same when the software is dependent on its environment, and that environment is continually changing, often in subtle ways? Proponents of software reliability modelling may argue that it is simply a matter of ensuring that the software is exercised in its operational environment. However, it is very difficult to identify all the dependencies of the software on its environment. Failure to do so can lead to misplaced confidence in the software's reliability.

Suppose a software program does not handle 29 February correctly in a leap year. Assuming there were no other defects, had we started running the program on 1 March 2016, then by 28 February 2020 we would have observed 35,040 hours of failure free operation. We would have had a high degree of confidence that the software is correct, yet it would have failed the very next day, 29 February 2020. Software reliability models attempt to predict future software behaviour based on observations of past software behaviour. There is a saying, attributed to Niels Bohr but which is apparently an old Danish proverb, that, "it is difficult to predict, especially the future".

The Ariane 5 accident showed that even a small change in operational profile (from Ariane 4 to Ariane 5) can have an unexpected and catastrophic impact on software behaviour (O'Halloran 2005).

3.3.4 Example 3: Defect Distribution

How are defects distributed by component in software systems? (Hopkins and Hatton 2019) analysed the software defects in a numerical library written in Fortran. They found very strong evidence of defect clustering with typically 80% of all components in the library exhibiting no defect. Furthermore, software errors are more likely to occur at parameter limits and boundaries, which is why software testing techniques such as boundary value analysis are so effective.

Defect clustering is in fact so ubiquitous that it appeared 4th in the "top 10 defect reduction list" compiled by (Boehm and Basili 2001). It follows that if a defect is observed in the implementation of a software feature, it is likely there will be further defects in features implemented by the same software component. Defect clustering turns out to be the result of combining the assumption of uniform distribution of defects across lines of code (or more accurately tokens) with the asymptotic power-law distribution of component lengths which holds for all software systems (Hatton 2014). Suppose then we have a software application that contains ten software components, and that we are told that this software application contains ten defects. The most likely scenario given our prior knowledge is that typically 8 of the 10 components are defect free and the remaining 2 contain all 10 defects. Of course, we don't know which but, as soon as we encounter a defect in a component, we know it's likely to have more defects and we should change our strategy

accordingly. Accommodating this in software temporal failure models is by no means obvious, and has not to our knowledge been accomplished.

3.3.5 Example 4: Non-Operational Modes and Easter Eggs

Some software programs have different modes of operation. As well as the normal operational mode, there may be non-operational modes, such as a bootstrap mode, a software update mode, or a debug mode. Should the software inadvertently enter one of these non-operational modes, then it will stop working as expected.

As described in (Ladkin and Littlewood 2016), another variation is that some software contains what are known as Easter Eggs. These are features that are intended to surprise and delight the user. For example, both Microsoft® Excel 97 and Google Earth contain a flight simulator. The Excel 97 flight simulator is launched by creating a new worksheet, pressing F5, typing "L97:X97", pressing Enter, pressing Tab once, holding down Ctrl + Shift and left clicking the Chart Wizard toolbar icon. The Google Earth flight simulator is launched by typing Ctrl + Alt + A. If an Easter Egg is activated inadvertently, then again, the software will stop working as expected.

Easter Eggs is a term also used in the cyber security context. They are “surprises” that can be triggered by environmental conditions decided upon by a malicious actor. Should those conditions be met, the Easter Egg will execute. These conditions may not have been thought of by developers as unintentional conditions that trigger the surprise and, of course, would have been missed by test. Consequently, the software will work normally, possibly for many years, until the right conditions are met, and this evades statistical modelling.

3.3.6 Example 5: Duration of the Experiment

From a statistical point of view, it does not matter whether we run one copy of the software for a million hours or a million copies of the software for an hour each; both experiments result in one million hours of operation. In the real world, these two experiments are not the same. Running a million copies of the software for an hour each does not give us confidence that the software will run for two hours, let alone for days, months or years. Why do we expect the environment’s behaviour for one hour to be the same as the environment’s continuous behaviour for a million hours?

3.4 Limitations

There are some severe limitations that apply even if we accept that the assumptions are valid for a particular software application:

1. A very large number of hours of operation is required to produce a statistically significant result (Butler and Finelli 1993, Kalra and Paddock 2016).
2. Furthermore, the mathematical models used require that we observe no failures. If we observe a failure, we need to start the experiment again. Running a software program for 10,000 hours and observing no failures is not the same as running a software program for 10,000 hours, observing 10 failures and fixing the defects that caused those 10 failures. In both cases, there are no known defects at the end of the experiment. However, in the first case, we would have a high degree of confidence that if we ran the software for another 10,000 hours, we would still see no defects. In the second case, if we ran the software for

another 10,000 hours, we would expect to see further defects (they would just not be the same defects).

3. No software changes can take place during the experiment. If the software is changed in any way, the experiment must be restarted unless justification can be provided as to why the changes made do not invalidate the data collected up to that point. Indeed, this is the approach used in hardware reliability experiments such as those used for airframe stress tests.
4. The software reliability claim is only valid for the software version that was tested, running on the same hardware that was used in the test. If a new software version is released, the experiment must be repeated.
5. The operational profile must be identical.
6. There needs to be an effective system for observing, recording, and documenting faults.

3.5 Dissimilar Software

There is a belief that dissimilar software development will defend against common mode or common cause development errors in safety critical systems. In an experiment reported in (Knight and Leveson 1986), the programs were individually extremely reliable but that the number of tests in which more than one program failed was substantially more than expected. (Knight and Leveson 1986) showed that there were common mode errors introduced into software regardless of independence of people, hence diversity didn't achieve the desired outcome. These were common mode errors in the software implementation. Dissimilar software development would not have been able to protect against common mode errors in the software requirements, since the same software requirements were given to all the participants in the experiment.

In a later paper, (Knight and Leveson 1990) stated, "*Until N-version programming has been shown to achieve ultra-high reliability and/or has been shown to achieve higher reliability than alternative ways of building software, the claims that it does so should be considered unproven hypotheses. Until these hypotheses are shown to hold for controlled experiments, depending on N-version programming in real systems to achieve ultra-high reliability where people's lives are at risk seems to us to raise important ethical and moral questions. Attacking us or our papers will not change this*".

On the other hand, (Hatton 1997) concluded, "*The balance of data tends to suggest that N version techniques are preferable in software development when the cost of failure is high due to our inability to make one really good version whatever techniques we currently use. Even though the advantage is much less than that for N independent channels, the difference is still substantial and in this case gave a factor of 5–9 improvement on average for a majority voted 3 version system compared with a single version typical of the current state of the art*". However, this claim was based only on the data from the Knight and Leveson experiment and on the state of the art at the time. More work like (Hatton 1997) is needed in order to widen the samples, and to account for modern development and verification practices.

EASA is promoting dissimilar software. There is a difference of opinion between the FAA and EASA, so dissimilar software is currently the topic of a harmonization effort between the FAA and EASA. EASA have been keen to emphasise that they do not mandate dissimilar software development, but that they do require independence. It is worth examining in this paper, the basis for EASA's position.

At a meeting at which one of the authors was present, EASA stated that for flight controls, EASA will not accept full reliance on Development Assurance and Quality Assurance as

sole mitigation of a common mode leading to a total loss of flight controls. When asked, the EASA presenter stated this EASA position is documented in the EASA Generic Certification Review Item (CRI) on "Consideration of Common Mode Failures and Errors in Flight Control Functions" (EASA 2014). This Generic CRI states that the EASA position is that Development Assurance alone is not necessarily sufficient to establish an acceptable level of safety for Flight Control functions, as described specifically in CAST 24 (CAST 2006), and that mitigation means or techniques should be provided to protect against Common Mode Failures/Errors, including software development errors. (EASA 2014) does not specifically address dissimilar software development. (EASA 2014) is a draft CRI that has never been formally issued.

(CAST 2006) does discuss design diversity, dissimilar software, and N-version design. (CAST 2006) cites three papers on design diversity and N-version design, which are (Littlewood 1996), (Littlewood et al. 2000), and (Littlewood et al. 2001). (CAST 2006) has now been withdrawn. It is worth examining here, therefore, what these papers say:

1. (Littlewood 1996) concludes, "*In real applications of this work, it will be necessary to estimate what has actually been achieved, rather than to rely upon the more general results presented here. It is here that hardware engineers have a considerable advantage over their software counterparts. As long as there is sufficient failure data, from previous use of the different component types, that 'covers' the environments within which the new system will operate, the distributions that are needed to compute system reliability will be estimable. This contrasts with the software diversity situation, where estimability is severely constrained by the practical limitations to the number of versions that can be developed*".
2. (Littlewood et al. 2000) claims that, "*The idea that diversity may be a more cost effective way to deliver high diversity is alive and recently it was spelled out by Hatton*" (Hatton 1997). (Littlewood et al. 2000) concludes, "*it appears that Hatton's suggestion that design diversity is always going to be more cost effective than developing a single version software is not trustworthy. In order for us to be certain that diversity will bring more than it takes we need to measure the dependence, which is currently an open question*". We should note that Hatton did not actually suggest that design diversity is "always" going to be more cost effective. Hatton's claim was considerably more circumspect than claimed in (Littlewood et al. 2000) and, as noted earlier, it is recommended that more work like the (Hatton 1997) study is conducted.
3. (Littlewood et al. 2001) presents a theoretical model of hardware redundancy. This model assumes that the presence or absence of each fault (which is claimed to be a random event) is independent of the presence of any other fault, and of the presence in another version. They admit this is the novel assumption that allows them to predict distributions of the probability of failure per demand (pfd) of versions and systems. There is no basis for this assumption, even for hardware, other than that it is a convenience that makes the mathematics tractable. The assumption is certainly not valid for software, as is acknowledged as (Littlewood et al. 2001) concludes, "*However, we do not hide that many of these results are only useful for better understanding these complex, counter-intuitive problems: they do not lead to simple, general recipes for design and assessment. Such understanding is necessary before it is possible to begin engineering diverse fault-tolerant systems with dependability assurance founded on formal models*" and "*Difficulties remain in using some of this theory — most particularly in populating the models with estimates of their key parameters when dealing with real systems*".

In summary, the EASA position is based upon an unissued CRI, which itself was based upon CAST-24 (now withdrawn), which cites the three papers discussed above, which do not appear to support the EASA position.

3.6 Are the Models Correct? Are they Useful?

The proponents of software reliability claim that the advantage of assuming that software execution is a Bernoulli process is that the pertinent mathematics is simple, clear, and well understood. This may be so, but it does not follow that a Bernoulli process is an accurate (or even useful) model of software failures.

The use of a Bernoulli process to model software failures is fundamentally flawed. Treating software execution as a Bernoulli process assumes that the probability of success is the same every time the software is executed, which we have shown to be an unwarranted assumption.

This assumption *may* be warranted in specific circumstances. For example, the assumption might be warranted for a software interlock implemented in predicate logic with no state. Where the assumption is unwarranted, it leads to an exaggerated confidence in probabilistic testing and product service history.

3.7 Can the Models be Fixed?

(Hatton 2012) presented a mathematical model that hypothesises that software defects follow a power law distribution. This hypothesis was validated by an experiment using 55 million lines of source code. Just as improved financial models have been constructed that use power law distributions, it may be possible to create improved models of software defects using power law distributions. However, defect density is a static property of the software, while failure rates are the result of interaction between the software and its environment over time. While it might be possible to predict the probability of defects using improved models based on Hatton's work, that is not the same as predicting the probability of system failure as a result of the use of that software. If the conditions that trigger the defect are not present then the software will work, and the system will not fail; the converse is also true.

4 The Way Forward

4.1 Aerospace

(Littlewood and Strigini 1993) concluded that, "*The main point of our paper has been to try to show that here lies the main difficulty when we want to build a system with an ultra-high dependability requirement. Even if, by the adoption of the best design practices, we were to succeed in achieving such a dependability goal, it is our view that we would not know of our achievement at least until the product proved itself after a very great deal of operational use. Nor can we expect to overcome this problem with improved mathematical and statistical techniques*". (Littlewood and Strigini 1993) went on to state, "*Is there a way out of this impasse? One approach would be to acknowledge that it is not possible to confirm that a sufficiently high, numerically expressed dependability had been achieved, and instead make a decision about whether the delivered system is 'good enough' on different grounds. This is the approach currently adopted for the certification*

of software in safety-critical avionics for civil airliners". (Littlewood and Strigini 1993) claims, "*this falls far short of providing the assurance needed*".

RTCA/DO-178B (RTCA 1992) was published in 1992. It was superseded by RTCA/DO-178C (RTCA 2011), which made minimal changes to the core document, in 2011. There are now 27,012 commercial jet airplanes in service worldwide (Boeing 2021). No hull loss accidents in passenger service have been ascribed to failure of software developed to RTCA/DO-178B to implement its requirements correctly (Daniels 2011). Software has been a contributing factor in a small number of accidents and in-flight upsets. Modern aircraft and their software are extraordinarily safe (Rushby 2012). We now have nearly 30 years of service experience that satisfying the objectives of RTCA/DO-178B/C has been sufficient to address the software considerations in aircraft certification.

4.2 Proof and Economics

We contend that our ability to make one really good version has improved in the last 30 years through the introduction of improved software development and verification techniques. In particular, the processing power of modern CPUs has made automated program proof increasingly tractable. Formal methods allow us to verify that a software artefact is correct, complete, and unambiguous.

There are plenty of anecdotes that claim that formal methods will save developers time and money; some may even say that it is 'better' than any other approach. Indeed, (Rushby 2009) and (Littlewood and Rushby 2012) wrote that formal methods may have a part to play in providing higher assurance based upon the idea introduced by (Littlewood 1998) that software may be "possibly perfect". The general proposition is that, "possible perfection provides a bridge between the verification activities used to ensure correctness of software and the probabilistic estimates required for failure at the system level". Given that use of formal methods can qualitatively achieve 'at least as good software' as any other approach, the question is what evidence is there that it does save time and money?

It has been the case for a number of years that large developers, such as Google, Facebook, Amazon, and Microsoft have been recruiting formal methods expertise with the aim of being more robust (Garavel et al 2020). For example, (Ball et al. 2004) describes how Microsoft Research developed a Static Driver Verifier using formal methods. This tool is now widely used to verify third party Windows device drivers and has greatly reduced the number of Blue Screens of Death (BSOD). Again, this is a qualitative argument, but the market appears to have its view. While there exists the possibility that better software reliability models could be developed, the only techniques available at the present time that can predict accurately the behaviour of software before it is executed are formal methods. An approach therefore might be to use formal methods wherever practicable and to back this up with testing and/or simulation where needed in order to increase confidence.

The use of formal techniques, like any software development technique, has to be based upon assumptions about the environment and there are limits to what these techniques can, or indeed should be, used for (Murray and van Oorschot 2018).

4.3 An Industrial Experiment

An experiment, run by Warwick University Manufacturing Group in 2018, on automotive software showed the benefits of the use of formal techniques. The blind trial involved benchmarks of real automotive requirements and design for 6 functions that had been seeded with 48 errors. The trial measurement process was developed by York Metrics and

time taken for each stage of the verification process was recorded. There were 3 processes used independently by 3 different people: baseline human review, tool assisted using Simulink Design Verifier (SLDV) and a formal method-based tool developed by D-RisQ Ltd called Modelworks®, and in all 3 cases the 48 errors were detected. In the case of Modelworks®, a 49th error was discovered that had not been seeded. However, the time for Modelworks® to discover these errors was between 60–80% faster across all 6 cases measured, in one case reducing from 52 hours down to 10 hours; see Figure 1. It will be noted that there are 7 sets of measurements. Experiment PP failed to complete analysis as time for the project ran out, though Modelworks® found the relevant errors. The 7th case is a repeat of TA (i.e., TA2) by a different set of 3 people in a second company applying their own processes for the benchmark. It can be seen that the results are very close (but not the same) thus demonstrating consistency in the experimental results.

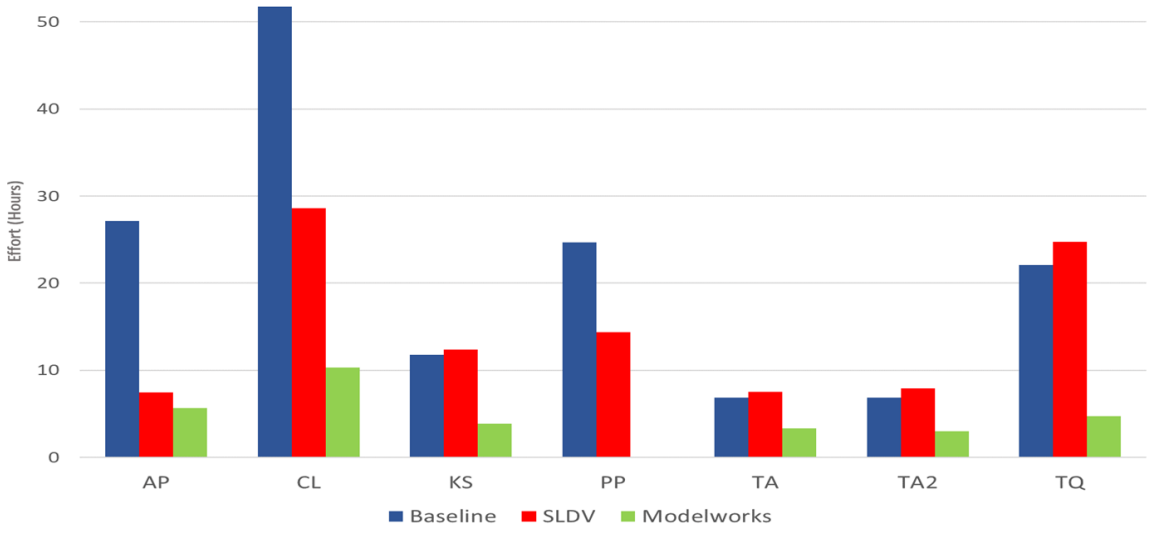

Figure 1 ~ Industrial Scale Benchmarking Study

4.4 Requirements Engineering

The weak link, for avionics software at least, remains requirements engineering. There has not been a single hull loss accident in passenger service that has been ascribed to failure of software developed to RTCA/DO-178B/C to implement its requirements correctly. There have been several accidents where software implemented the requirements correctly, but those requirements specified unsafe behaviour under some unforeseen circumstance. For example, an Airbus A320 overran the runway at Warsaw on 14 September 1993 (MCAAI 1994). A contributing factor was that deployment of the ground spoilers and engine thrust reversers was delayed because of a requirement to deploy them only when both main landing gear struts indicated Weight on Wheels. In another accident, an Airbus A320 overran the runway at Sao Paulo on 17 July 2007 (CENIPA 2009). A contributing factor was that the pilot only pulled one thrust lever into the reverse thrust position (the other thrust reverser was known to be inoperative), but a requirement stated that both thrust levers must be in the idle or reverse thrust position for either of the thrust reversers to be deployed. Finally, in the two recent Boeing 737 MAX accidents on 29 October 2018 and 10 March 2019, the Manoeuvring Characteristics Augmentation System (MCAS) software implemented its requirements correctly, but the requirements caused full nose down trim to be applied following an Angle of Attack sensor failure (Daniels 2020).

As Nancy Leveson has said, “Software-related accidents are usually caused by flawed requirements”. It therefore follows that our efforts should be focused on writing better requirements. Formal methods can help with writing better requirements by using formal requirements languages with unambiguous semantics and formal methods tools that can ensure the requirements are complete and consistent.

5 Conclusion

We have shown in this paper that software reliability models that assume that software execution is a Bernoulli process are flawed. The assumption of independence may be warranted in specific circumstances, but otherwise these models lead to an exaggerated confidence in probabilistic testing and product service history. These models cannot, in general, be used to demonstrate ultra-high reliability.

There seems to be an opportunity for the research community to investigate further the efficacy of N-version programming in mitigating the impact of software errors.

The situation is not as unsatisfactory as was stated in (Littlewood and Strigini 1993). We have now accumulated nearly 30 years of service experience that satisfying the objectives of RTCA/DO-178B/C has been sufficient to address the software considerations in aircraft certification.

The state of the art has improved considerably since RTCA/DO-178B was published in 1992. Formal methods are now used more widely, and automated program proof is now tractable and cost-effective.

Requirements continue to be the weak link. Analysis of historical aircraft accidents suggests that in those accidents where software was involved, the software implemented its requirements correctly, but the requirements specified behaviour that was unsafe and led to the accident.

We need to focus on getting the requirements right if we are to improve safety.

References

Ball, T., Cook, B., Levin, V. & Rajamani, S.K. (2004). *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft* (Technical Report MSR-TR-2004-08). Microsoft Research. Retrieved 22 September 2021 from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2004-08.pdf

Boehm, B. & Basili, V. R. (2001). Software Defect Reduction Top 10 List. In *IEEE Computer. January 2001,* 135–137. Retrieved 8 November 2021 from https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf

Boeing. (2021). *Statistical Summary of Commercial Jet Airplane Accidents, Worldwide Operations 1959 – 2020*, Boeing. Retrieved 9 November 2021 from https://www.boeing.com/resources/boeingdotcom/company/about_bca/pdf/statsum.pdf

Butler, R.W., & Finelli, G.B. (1993). The infeasibility of quantifying the reliability of life-critical real-time software. In *IEEE Transactions on Software Engineering, 19(1)*, 3–12. Retrieved 28 September 2021 from https://shemesh.larc.nasa.gov/fm/papers/Butler-nonq-paper.pdf

CAST. (2006). *Reliance on Development Assurance Alone When Performing a Complex and Full-Time Critical Function* (Position Paper CAST-24 Rev 2). FAA Certification Authorities Software Team.

CENIPA. (2009). *Final Report A – No 67/CENIPA/2009.* Retrieved 12 November 2021 from http://sistema.cenipa.aer.mil.br/cenipa/paginas/relatorios/rf/en/3054ing_2007.pdf

Daniels, D. (2011). *Thoughts from the DO-178C committee*. Paper presented at the 6th IET International Conference on System Safety, 2011.

Daniels, D. (2020). *The Boeing 737 MAX Accidents*. Paper presented at the 28th Safety-Critical Systems Symposium, York, UK. Retrieved 9 November 2021 from https://scsc.uk/rp154.1:1

EASA. (2014). *Consideration of Common Mode Failures and Errors in Flight Control Functions* (Draft Generic CRI D-05), European Aviation Safety Agency.

Garavel, H., ter Beek. M. & van de Pol, J. (2020). *The 2020 Expert Survey on Formal Methods.* Paper presented at the 25th International Conference on Formal Methods for Industrial Critical Systems, Vienna, Austria. Retrieved 9 November 2021 from https://hal.inria.fr/hal-03082818/document

Hatton, L. (1997). *Are N average software versions better than 1 good version?* IEEE Software, SE-14(6), 71–76. Retrieved 22 September 2021 from https://www.leshatton.org/Documents/Nver_1297.pdf

Hatton, L. (2012). *Power-laws and the Conservation of Information in discrete token systems: Part 2 The role of defect.* Retrieved 17 September 2021, from https://arxiv.org/abs/1209.1652

Hatton, L. (2014). *Conservation of Information: Software's Hidden Clockwork?* IEEE Transactions on Software Engineering, 40(5), 450–460. Retrieved 8 November 2021, from https://www.leshatton.org/Documents/TSE-2013-08-0271_V1.0a.pdf

Hopkins, T.R. & Hatton, L. (2019). *Defect patterns and software metric correlations in a mature ubiquitous system.* Retrieved 17 September 2021, from https://arxiv.org/abs/1912.04014

IEC. (2010a). *Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 3: Software requirements.* International Electrotechnical Commission.

IEC. (2010b). *Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 7: Overview of techniques and measures.* International Electrotechnical Commission.

Kalra, N., Paddock, S.M. (2016). *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation. Retrieved 11 November 2021 from https://www.rand.org/pubs/research_reports/RR1478.html

Knight, J.C. & Leveson N.G. (1986). *An experimental evaluation of the assumption of independence in multi-version programming.* IEEE Transactions on Software Engineering, 12(1), 96–109. Retrieved 22 September 2021 from http://sunnyday.mit.edu/papers/nver-tse.pdf

Knight, J.C. & Leveson N.G. (1990). *A reply to the criticisms of the Knight & Leveson experiment.* ACM SIGSOFT Software Engineering Notes, *15*(1), 24–35. Retrieved 8 November 2021 from http://sunnyday.mit.edu/critics.pdf

Ladkin, P.B. & Littlewood, B. (2016, February). *Practical Statistical Evaluation of Critical Software.* Paper presented at the 24th Safety-Critical Systems Symposium, Brighton, UK. Retrieved 27 September 2021 from https://scsc.uk/r131/8:1

Littlewood, B. & Verrall, J.L. (1973, April). *A Bayesian reliability growth model for computer software*. Paper presented at the IEEE Symposium on Computer Software Reliability, New York, USA.

Littlewood, B. & Strigini, L. (1993). *Validation of Ultrahigh Dependability for Software-Based Systems*. Communications of the ACM (CACM), 36(11), pp. 69–80. Retrieved 9 November 2021 from https://openaccess.city.ac.uk/id/eprint/1251/1/CACMnov93.pdf

Littlewood, B. (1996). *The impact of diversity upon common mode failures*. Reliability Engineering & System Safety, 51(1), 101–113. Retrieved 22 September 2021 from https://openaccess.city.ac.uk/id/eprint/1630/

Littlewood, B. (1998) *The use of proof in diversity arguments*. IEEE Transactions on Software Engineering, 26(10), 1022–1023. https://www.researchgate.net/publication/3188113_The_Use_of_Proof_in_Diversity_Arguments_in

Littlewood, B., Popov, P. & Strigini, L. (2000). *N-version design versus one good version.* Paper presented at the International Conference on Dependable Systems & Networks, New York, USA. Retrieved 22 September 2021 from https://www.researchgate.net/publication/2585976_N-version_design_Versus_one_Good_Version

Littlewood, B., Popov, P. & Strigini, L. (2001, February). *Design Diversity: an update from research on reliability modelling.* Presented at the 9th Safety-Critical Systems Symposium, Bristol, UK. Retrieved 22 September 2021 from https://openaccess.city.ac.uk/id/eprint/261/2/SCSS2001_v_12.forDistrib.pdf

Littlewood, B. and Rushby, J. (2012). *Reasoning about the Reliability of Diverse Two-Channel Systems in Which One Channel is "Possibly Perfect".* IEEE Transactions on Software Engineering, 38(5), 1178–1194. Retrieved 9 November 2021 from https://openaccess.city.ac.uk/id/eprint/1069/1/1oo2-revised-13apr11.pdf

Mandelbrot, B.B & Hudson, R.L. (2004). *The (Mis)Behavior of Markets: A Fractal View of Risk, Ruin and Reward*, Basic Books.

MCAAI. (1994). *Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993*, Main Commission Aircraft Accident Investigation Warsaw, March 1994. Retrieved 9 November 2021 from http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html

Murray, T. and van Oorschot, P. C. (2018). *BP: Formal Proofs, the Fine Print and Side Effects.* Paper presented at 2018 IEEE Cybersecurity Development (SecDev). Retrieved 9 November 2021 from https://people.scs.carleton.ca/~paulv/papers/secdev2018.pdf

O'Halloran, C. (2005). *Ariane 5: Learning from Failure.* Proceedings of the 23rd International System Safety Conference, San Diego, 2005.

RTCA. (1992). *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, RTCA, Inc. Also available as EUROCAE Document ED-12B.

RTCA. (2011). *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178C, RTCA, Inc. Also available as EUROCAE Document ED-12C.

RTCA. (2021). *Terms of Reference, Special Committee (SC) 240, Topics on Software Advancement (Revision 1)*, RTCA Paper No. 083-21/PMC-2139, RTCA, Inc. retrieved 9 November 2021 from https://www.rtca.org/wp-content/uploads/2021/05/SC-240-TOR-Rev-1-Approved-2021-03-18.pdf

Rushby, J. (2009). *Software Verification and System Assurance*. Presented at the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Hanoi, Vietnam, November 2009. In SEFM Proceedings pp. 3–10. Retrieved 22 September 2021 from http://www.csl.sri.com/users/rushby/papers/sefm09.pdf

Rushby, J. (2012). *New Challenges in Certification for Aircraft Software.* Presented at the Ninth ACM International Conference on Embedded Software (EMSOFT), Taipei, Taiwan. Retrieved 22 September 2021 from http://www.csl.sri.com/users/rushby/papers/emsoft11.html

This collation page left blank intentionally.