Perspectives on Probabilistic Assessment of Systems and Software

Emmanuel Ledinot[(1)], Jean-Paul Blanquart[(2)], Jean Gassino[(3)], Bertrand Ricque[(4)], Philippe Baufreton[(4)], Jean-Louis Boulanger[(5)], Jean Louis Camus[(6)], Cyrille Comar[(7)], Hervé Delseny[(8)], Philippe Quéré[(9)]

(1): Contact author, Dassault Aviation, emmanuel.ledinot@dassault-aviation.com; (2): Airbus Defence and Space; (3): Institut de Radioprotection et de Sûreté Nucléaire; (4): Safran; (5): CERTIFER; (6): ANSYS-Esterel Technologies; (7): AdaCore; (8): Airbus; (9): Renault

Abstract:

Safety standards in most domains (aeronautics, automotive, industry, nuclear, railway, space) consider software (and more generally, design) as a deterministic artefact. They propose a global rationale combining probabilistic evidence on hardware random failures and deterministic evidence on systematic causes of failures including software. In a context where software is more and more pervasive in all systems, and where it is sometimes advocated that software complexity and size seem to provide some relevance to a probabilistic view of software behaviour, several initiatives suggest to change the way to address software in the global system safety assessment. This is a complex question with many facets. Among them the authors propose to discuss in the paper:

- foundations, relevance and limits of probabilistic assessment for software,
- relationship between software criticality category, (or class, DAL/SIL/ASIL/SSIL etc.) and probabilistic safety objectives,
- the rationale for software diversification and to what extent probabilistic assessment is part of it.

Keywords: software statistical testing, probabilistic system safety assessment, rationale of safety standards, DAL, SIL, ASIL, SSIL, cross-domain comparison.

1. Introduction – Position of the paper

Over the years, it has been recognized on the one hand that failures of safety systems may be due not only to random failures but also to "systematic causes" (be they called design faults, development errors etc.), and on the other hand that this kind of causes are less amenable to probabilistic assessment than e.g., random hardware failures. This is acknowledged and addressed by most existing safety standards under the form of the combination of quantitative (probabilistic) evidence and qualitative (deterministic) evidence, as appropriate regarding the various kinds of causes (see e.g., [Baufreton et al. 2010]).

The increasing role and complexity of software in systems, including safety critical ones, seem to make probabilistic evaluation of software reliability and statistical testing more and more attractive for some industry actors. They propose to at least modify the equilibrium between the various kinds of evidence in the global assessment framework (quantitative vs. qualitative, probabilistic vs. deterministic, etc.).

Originally motivated by technical discussions held within standardization committees, the work presented in this paper was undertaken by a French cross-domain group working on safety standards([1]). Software probabilistic assessment and statistical testing have been investigated for decades and a comprehensive bibliography would encompass hundreds of references. Starting from [Strigini et al. 1997], [Rushby et al. 2014] and [Ladkin et al. 2015], we address among the many facets of this complex matter the following aspects:

- whether and to which extent probabilistic assessment could be valid for software, particularly for safety critical systems,
- which relationship could exist between software criticality category (or class, DAL/SIL/ASIL/SSIL etc.) and probabilistic safety objectives,
- whether the benefits of solutions such as software diversification could be amenable to some quantification,

[1] Originally created in 2009 as part of the "Club des Grandes Enterprises de l'Embarqué" this working group on safety standards is now attached to Embedded France. It regularly publishes the results of exchange and common work between its members, experts in safety and related standards covering as many domains as aeronautics, automotive, industry, nuclear, railway, space. See e.g., [Baufreton et al. 2010], [Blanquart et al. 2012].

2. Technical Background

2.1. What is software statistical testing?

The characteristic feature of software statistical testing is probabilistic generation of test data. There are various purposes for doing so, and various ways of doing so. Following [Thevenod 91], [Thevenod 95], randomness on inputs may be used to *find faults* or to *assess dependability* at the end of the verification stage. In the former case, *coverage criteria* are the outcome of the statistical testing activity, as opposed to values of *probabilities* in the latter case.

The coverage criteria are based on activation counters logging how the items of the software are exercised by the generated input data. Depending on the nature of these items, statistical testing is qualified as *structural* or *functional*:

- structural if the items are implementation-oriented (e.g. control flow-graphs),
- functional if they are more specification-oriented like state-transition graphs or data-flow graphs.

When randomness is "blind", i.e. when the random input profiles are generated by means of uniform probability laws carrying no frequency information related to actual usage or operation, statistical testing is named *random testing*.

Random testing applies exclusively to statistical testing devoted to finding faults and fault removal. Random testing cannot be used for software dependability assessment. To support software dependability assessment, the randomly generated input data must be statistically consistent with the operational profiles.

It is sometimes claimed that statistical testing may support fault forecasting based on software reliability growth models, in addition to fault finding and removal.

Finding Faults		**Assessing Dependability**
Structural Coverage		Event Probabilities
Specification-oriented	Implementation-oriented	
Functional Statistical Testing	Structural Statistical Testing	Statistical Testing

Table 1: Types of statistical testing

In this paper we exclusively focus on statistical testing for software dependability assessment.

2.2. What is "software failure"?

A given piece of software, intended to perform a specified function, may be affected by faults in its functional requirements or by errors in the development process. When the inputs activate a fault, the computed outputs differ (deterministically) from the intended values and a system failure may occur.

While at random dates hardware components may *lose* some functional capability, software faults, when present, are present from the very beginning. The wording "software failure" inherited from hardware, electrical and mechanical engineering is convenient but misleading. When some definite inputs activate a fault into malfunctioning it may be *perceived* at system or user level as a random failure event. But "randomness" is present only in input and execution context variability, in other words in the operational profile (OP).

Whatever "software failure" event is quantified, a trigger of system safety, system reliability or system availability events, the software[2] to be assessed is a deterministic transformer of the inputs, which vary with some randomness according to the actual OP, hence the critical importance of OP modelling accuracy to ensure validity of the probabilistic assessment. This is the reason why we illustrate sensitivity to OP modelling by an example.

[2] Since our primary concern is safety critical software, software is assumed deterministic

2.3. Probabilistic modelling of software dependability assessment

A probabilistic model consists of a random experiment repeated with independence and observed by means of events that must meet Boolean algebraic properties. Then a probability measure is defined on these events [Rényi 2007].

In case of software statistical testing the random experiment is made of five steps, and is repeated N times where N is the sample size:

- *Set-up* of the program in a state common to the N experiments and ensuring *independence* between them. The more complex the software and its execution environment, the harder to meet this compelling requirement,
- *Random generation* of an input sequence (IS), consistent with OP probability law. OP denotes the set of all possible input sequences submitted to the program at operation-time, plus *frequency information*: a probability density function (pdf) defined over the set of all possible input sequences. In practice OP, as a probability law, is a very complex object: a high-dimension support set, plus a multivariate function defined on this set. The duration of one input sequence is that which is relevant for the quantified event: constant or variable, cycle, minutes, missions; hours, years etc. As previously mentioned, random generation does not mean random testing since the probability laws used on the program inputs are not all uniform. Some may be uniform, but this should not be a "by-default choice" because of lack of information but by explicit choice to ensure adequacy[3] with operational conditions.
- *Run* of the program with the generated IS,
- *Event evaluation* of the run. An event is a predicate defined on some observation variables common to all runs (program I/Os, internal variables, environment variables etc.). Depending on how far the specification is formalized and amenable to execution, the information gathering process to evaluate the event-predicate is automated or not. The event-predicate is the test oracle of deterministic testing,
- *Decision.* In the end, the random experiment is summed-up into a yes/no decision status on the event-predicate. This status is the run-specific realization of the Bernoulli random variable D associated to the event. It is a binary random variable. Its probability law is defined by:
    - Pr(D=0)=p, where p is the parameter of the Bernoulli law of D. We assume that the event associated to D is stated so that "false" means "problem occurrence". Probability p is the risk of "software failure", which has to be estimated by means of the N-sample of independent runs.
    - Pr(D=1)=q=(1-p) since the two events are exclusive and complementary. The realization of D at run n in N is like tossing a coin. Probability p is not necessarily very small, i.e. that of a rare event. It is the limit of C_0/N where C_0 is the number of (D=0) events in the N runs.

Once C_0 is known, it is not possible to compute p from N and C_0 only. One can choose $p= C_0/N$, which is the only sensible choice available. But another N-sample of runs would have given a different count C_0'. Hence the computed p would fluctuate if we repeated the building of samples made of N runs.

This problem is overcome by standard interval estimation of the parameter p for the binomial law of parameters N and $k=C_0$ associated to D. The binomial law is the probability law of the number of heads (resp. tails) observed after tossing N times the same coin of probability p. Handling the random fluctuations of C_0/N over repeated N-run-samples is analytically tractable.

Given an accepted risk of error on the computed p because of performing a *unique* N-run-experiment (this is a *design-office* risk, usually noted α, whose value is commonly *chosen* between 10% and 1%), one can compute an interval in which the true value of p is likely to lie. The probability of the design office event "the true value is *not* in the computed interval" is α. So the confidence on the interval is (1 – α). In the example of section 2.5 that illustrates the sensitivity to OP and the robustness issue, we took α = 1%.

[3] Validity or accuracy might be preferred as synonyms

2.4. Probability of fault freeness

Up to now, we did not address the quantity of faults in the software. We did not consider if there are any, nor how many they are and where they are. We restricted ourselves to an external view, just counting the discrepancies of the program's behaviour when observed through events and N runs.

Statistical testing is sometimes related to this second question of the quantity of residual faults in a piece of software. [Rushby et al. 2014] proposed a conceptual framework to encompass both issues:

- development assurance efficacy, that is estimating the likelihood of existence of residual faults in spite of a rigorous development, and possibly evolution of this likelihood over time (software reliability growth models),
- probabilistic software assessment, a snapshot view of software reliability, without the explicit estimation of the quantity of residual faults.

[Rushby et al. 2014] aims at bridging the gap between *deterministic software* correctness and *probabilistic system* safety. The authors attempt to define the probability of software perfection (fault freeness) and the probability of software failure under "randomly selected demand". They state the formula:

Pr(Sw Fails")=

Pr("Sw Fails" | "Sw is fault-free").Pr("Sw is fault-free") +

Pr("Sw Fails" | "Sw is not fault-free").Pr("Sw is not fault-free").

which we abbreviate in:

$$Pr(SF)=Pr(Sc).Pr(SF|Sc) + Pr(Snc).Pr(SF|Snc)$$

Consistently with the event evaluation stage in the random experiment definition (cf. .section 2.3), we interpret the notion of "Software Failure" in the following way: it assumes the existence of an oracle (computerized or human-based) over the observables of software Sw, and the existence of a set of selected runs that may violate this oracle. 'Sc' means 'S is correct (fault free)', and 'Snc', **n**on **c**orrect, is its negation. The formula is based on the total probability theorem for the "fault-free" vs "not fault-free" alternative.

We question the relevance of the concept underlying Pr(Sc) and Pr(Snc), in practice at the very least. In any case it is true that a fault-free software cannot activate a failure, so the conditional probability equation Pr(SF|Sc)=0 holds, and the formula simplifies to:

$$Pr(SF)=Pr(Snc).Pr(SF|Snc)$$

In [Rushby et al. 2014] different ways to estimate or upper-approximate Pr(Snc) are discussed, attempting to take into account the influence of development assurance levels (DAL, SIL, ASIL, SSIL). In 2.2 and 2.3 we tried to define Pr(SF|Snc) precisely and we sketched out how to compute its estimate using binomial parameter estimation.

We would like to underline that "randomly selected demand" has *no intrinsic meaning*: does it mean e.g., conformant to uniform laws on the inputs, to normal laws, to any arbitrary law on some physically significant combination of some inputs to be estimated in operation?

As already mentioned, Pr(SF) is critically dependent on OP, the probability law that drives the inputs to software [Strigini et al. 1997]. The Ariane 501 accident provided a spectacular example of the critical dependency of Pr(SF) on OP: a range change on a very single parameter (the horizontal velocity) and Pr(SF) jumped from ~0 to 1.

A more precise simplified formula making *explicit* the dependencies with respect to OP would be:

$$P_{OP}(SF)=P(Snc).P_{OP}(SF|\ Snc)$$

2.5. Robustness of software probabilistic assessment

As reviewed in [Ladkin 2015], there are many difficulties to overcome for functional statistical testing to be performed in a valid manner. We would like to underline an additional one: the possible instability, and hence absence of meaning, of the estimated probabilities with respect to great, or even tiny, variations on OP probability density function (pdf).

The example program is derived from the famous Bertrand's paradox [Rényi 2007]. It takes as inputs the coordinates of two points in the plane. It checks whether these points lie on the unit circle centered at the frame's origin and whether the length of the chord defined by the two points is greater than the side length of the encircled equilateral triangle. This geometric property is named (P). The chords meeting (P) are colored green, and the other ones are colored red.

```
function [status]=program(x1,y1,x2,y2)
epsilon=0.01;
R=1;
side=sqrt(3);

if abs(x1^2 + y1^2 - R^2) < epsilon &&
   abs(x2^2 + y2^2 - R^2) < epsilon,
   if sqrt((x2-x1)^2 + (y2-y1)^2) > side,
        status = 1;
   else status = 0; end;
else
    status = -1;
end
```

Figure 1: The source code of the program (distance computation and thresholding) derived from Bertrand's paradox

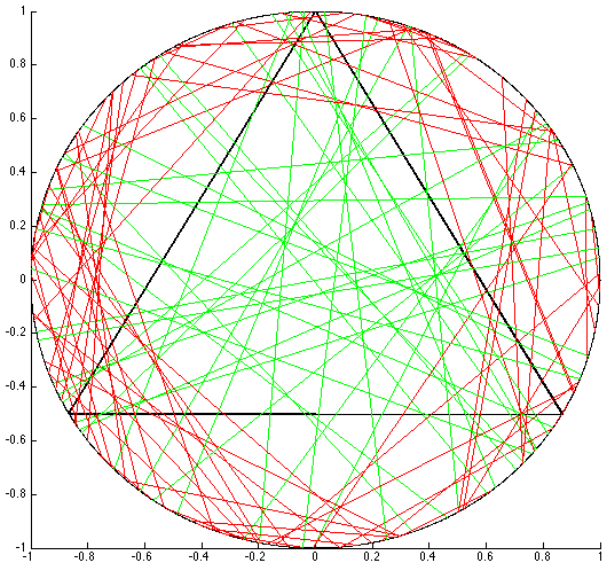

Figure 2: Functional statistical testing of (P) using uniformly generated random chords on the unit circle

We then use six different methods (D1 to D6) to generate the *uniformly* spread points over the circle (the ends of the chords). They only differ in the geometric construction of the points (Cartesian or polar coordinates etc.), all the random variables are uniformly distributed over their range ([-1,+1], [-π, + π], etc.). Two sample sizes are used (1 000 and 100 000). The results of the interval estimation of the binomial parameter at 99% confidence level for the statistical validity of (P) are the following:

	N = 100			N= 100000		
Sampling	**Pmin**	**Pe**	**Pmax**	**Pmin**	**Pe**	**Pmax**
D1	0,26	0,38	0,51	0,36	0,36	0,37
D2	0,22	0,34	0,47	0,33	0,33	0,34
D3	0,23	0,35	0,48	0,33	0,33	0,34
D4	0,27	0,39	0,52	0,33	0,33	0,34
D5	0,38	0,51	0,64	0,49	0,50	0,50
D6	0,38	0,51	0,64	0,50	0,50	0,50

*Table 2: the estimated probabilities of (P) for six different interpretations of "uniform" in the definition of the operational profile and two sample sizes. Pe is the **e**stimated probability p.*

The probability **e**stimate P_e ranges from 0.33 to 0.50, from 1/3 to 1/2, which is a surprising large variation when the six OPs that drive the sampling processes are expected to be similar enough to be considered *equivalent*. They are all eligible interpretations of "uniformly spread on the circle". Then, what is the meaning of these probabilities if they fluctuate so much for nearly undetectable reasons, uncontrollable in software verification practice?

2.6. Estimation of OP distribution laws

Example in section 2.5 illustrates the extreme sensitivity of software dependability assessment to the OP-pdf. Changes of OP-pdf have first order influence on $P_{OP}(SF| Snc)$ even for a program as trivial and regular as that of 2.5.

But even worse, when no change is intended on the law (let's say "uniform" as in 2.5), the way of building the test data generator conformant to this law leaves room to tiny degrees of implementation freedom that may have *also first order* influence on the estimation.

Unfortunately it is so whatever confidence level α is chosen. It is not a matter of estimator convergence and precision depending on N and α. Even with very small values of α (let's take 10^{-5}) and very large samples, the middle of the intervals can have chaotic jumps with respect to seemingly non significant changes in the implementation of the random generator.

This has consequences on qualification of COTS by proof in-use and service history. Extreme care must be taken as to the sensitivity of the computed probabilistic dependability indicators with respect to OP variability between the first and second usage context.

2.7. References to probabilistic software assessment in safety standards

Some standards such as EN 50128 refer to statistical testing as a possible means of software dependability assessment. Statistical evaluation is also referenced in an informative annex of part 7 of [IEC 61508] which states:

"*A probabilistic approach to determining software safety integrity for pre-developed software*". The text of this annex, now 17 years old, is very misleading. There are indications that very complex software such as operating system could be evaluated. This is not possible due to the very strict requirements applicable to operational history and data collection. These requirements are far beyond any practical application for such software. These critical aspects can remain unnoticed by a reader not deeply acquainted with the required mathematical statistics.

Furthermore, the state-of-the-art has significantly evolved since the references quoted in the standard. There is thus a clear need to reshape this text, clarify its mathematical foundations and define its possible scope of application.

Statistical testing is also mentioned in informative annex E of [IEC 60880]; anyway, this standard states that "*The validity of the calculated pfd depends upon the similarity of the profile of the test inputs to the profile of the actual inputs experienced by the system in operation. If (...) used on an unrealistic operational profile (...) a pfd will be estimated that may be very different to the actual system availability that would be obtained in active use. This is a fundamental weakness of the statistical testing approach as it is generally very difficult to accurately determine the operational profile that a system will experience in use, and this is particularly true for systems with large numbers of inputs.*"

The other standards considered in this paper (cf. References) do not resort to statistical assessment of software quality.

3. Software reliability and DAL/SIL/ASIL/SSIL

Given this setting, can we argue some link between development assurance levels and software reliability?

In system safety engineering, the probabilistic safety objective assigned to an entity determines its DAL by means of domain specific regulatory tables [Blanquart et al., 2012], [ED79A/ARP4754A], [EN 50129], [IEC 61508], [ISO 26262]. The converse is irrelevant, probabilities cannot be derived from DALs.

Since probabilities drive DALs at system level (resp. SIL, ASIL or SSIL in automation, automotive and railway), software items included, the question is "why not assigning DALs to legacy software or COTS by means of reliability measurement*?"* This would be some sort of reverse-engineered DAL, substantiated by product assessment instead of process assessment.

In process automation, some equipment and software vendors tend to lobby this way. In aeronautic, space, railway, and nuclear, rigour of component development is explicitly stated in standards as a *contextual* notion. It is system dependent, not specific to the component.

Component reuse or COTS use from system to system without dedicated component contextual evaluation, has to be performed with care. Masking system dependency, a reverse engineered DAL/SIL/ASIL/SSIL would be dangerously misleading.

4. N-version programming and system safety

4.1. Behavioral view .vs. probabilistic view

In all industrial domains it happens that hardware components reliability is too low to meet the catastrophic event probability objectives with single channel architectures. Duplex or triplex architectures introduce redundancies, possibly with hardware dissimilarity, to favour failure independence and thereby meet the quantified rareness objectives as low as e.g., 10^{-9} per hour or 10^{-5} failure on demand.

Because of the influence in the software arena of these compelling architectural patterns, or may be for the sake of uniformity in probabilistic safety and reliability assessment, there is some inclination towards quantifying "software failures", and managing their statistical independence.

In addition, since DAL/SIL/ASIL etc. may be seen as process-based means to ensure quantified safety objectives over all items, including software, there are candidate probabilities for software failures at hand: that of the regulatory tables such as 10^{-9}/h for ASIL D in automotive, or 10^{-5}/h for DAL C in aeronautics etc. As mentioned in §3, the DAL/SIL/ASILs ensue from or may be linked to probability objectives.

Assuming that system and software development assurance meet their objectives, the probability of all "failures" of software S could be uniformly upper-bounded by the regulatory value associated to its DAL/SIL/ASIL.

But even then, even with these sensible but disputable probabilities for 'software failures', software dissimilarity would not be motivated by search for partial or complete *statistical independence*, at least for safety critical software[4].

For safety critical software, dissimilarity, also named N-versions programming, aims at *architectural* and *behavioural* objectives. The aim is to avoid single-cause catastrophic failures initiated by software (or double-cause in space domain). Dissimilarity copes with *possibility* of software-initiated catastrophic behaviours, not *quantity* thereof. It resorts to common cause analysis, for catastrophic effects caused by single (or double) residual faults in specification or implementation.

A 1-version piece of software mapped on k hardware redundancies may generate common cause failures in two ways:

1. On its own, when its k-replicated behaviour turns out to be catastrophic at system level,

2. As a *coupling influence* over the k hardware replicates, which may no longer be independent initiators (both *causally* and *statistically*), in spite of their possibly independent constituencies.

The first one is addressed by software development assurance, which encompasses formal methods to tend to elimination of correctness faults (conformance defects in the wording of standards).

The second kind is addressed by a set of best engineering practices to *isolate* the software behaviour from the execution platform, i.e. from any other influence than its specified inputs, initial state, and configuration parameters. Isolation best practices are detailed in the next section.

So in the setting of safety critical systems, N-version programming is advocated, possibly imposed, either because of confusion with hardware and plant architecting, or because of silent doubt on process assurance's efficacy for the highest criticality levels.

4.2. Conditions for effective 1-SW k-HW redundancies

We consider a unique piece of software replicated on k hardware redundancies, which may be dissimilar or not. Are there conditions to ensure independence of hardware failures in spite of the potential common cause failure created by software uniqueness?

[4] Dealing with lower system/software criticalities (e.g. maintenance functions and system reliability).is another issue outside the scope of this paper.

Two-way isolation of the application software from its environment meets these conditions:

- No influence of the environment (operating system, hardware/software integration, etc.) on the functional behavior of the application layer. It must depend exclusively on its specified inputs and initial state.
- Conversely, no influence of the application software on the operating system, the execution platform, and more generally any external item other than the specified outputs.

4.3. Conditions for effective k-SW k-HW redundancies

In case of k-version programming over the k hardware redundancies, one may distinguish two situations:

- k-version at implementation level only,
- k-version at specification level and implementation level.

The intended meaning of specification here includes system requirements allocated to software, software high level requirements, functional requirements, design and low level requirements.

Considering k-version implementation to enforce software failure independence, [Knight et al. 1986] provided experimental evidence of non-effectiveness. Moreover, as software development methods significantly improved since the late 80s (model-based design, automatic code generation, model-checking), correctness of implementation is not the major concern.

In spite of software engineering progress, validity and completeness of system and software specifications remain a major issue. When possible, diversification at specification level would be beneficial. But it is most often than not very difficult to state the same algorithmic problem in two actually different manners, and then prove that the two formulations define the same set of expected behaviors…

Functional diversity (provision of different functions, e.g. based on different physical phenomena, to achieve the same safety objective) is even stronger than specification diversity, and is used in nuclear, space and other domains.

5. Conclusion

We tried to delineate some border lines in system and software safety assessment, mainly deterministic vs. random, behavioural vs. statistical.

These border lines are here and there left implicit in the standards, possibly because of some subtleties in their rationale, possibly also because of the limits of current engineering methods and tools.

We mainly focused on the validity conditions of probabilistic “software failure” estimation and on two system level aspects of probabilistic software assessment: design assurance levels and n-version programming. The validity of the operational profile distribution law and the sensitivity, possibly chaotic to this law, seemed to us the main impediments to probabilistic assessment of software dependability (not even mentioning the well-known difficulties related to the needed computation effort and time for ultrahigh reliability software, definitely an important issue as well though not addressed here).

Unfortunately, these impediments are even more hindering as software complexity increases, whereas statistical testing is sometimes advocated as an opportunity for greater cost effectiveness on very large software.

Driven by the increasing complexity of software and the trend toward ubiquitous systems of systems, there is some incentive to grant credit to statistical software assessment in safety standards under revision.

We explained why we remain extremely cautious about the validity of computed probabilities related to “software failures” and why we feel some danger in such a trend.

Basically, we reject, for ultrahigh reliability software, a move towards more statistical assessment against less development assurance. However, such a move may be debatable on low reliability software.

6. References

[Baufreton et al., 2010] P. Baufreton, JP. Blanquart, JL. Boulanger, H. Delseny, JC. Derrien, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, J. Machrouh, P. Quéré, B. Ricque, “Multi-domain comparison of safety standards”, ERTS-2010, 19-21 May 2010, Toulouse, France.

[Blanquart et al., 2012] JP. Blanquart, JM. Astruc, P. Baufreton, JL. Boulanger, H. Delseny, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, J. Machrouh, P. Quéré, B. Ricque, “Criticality categories across safety standards in different domains”, ERTS-2012, 1-3 February 2012, Toulouse, France.

[Butler et al. 1993] R. W. Butler, G. B. Finelli “The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software” IEEE Transactions on Software Engineering Vol 19 n°1 January 1993.

[ECSS-Q40] “Space product assurance – Safety”, European Cooperation for Space Standardisation, ECSS-Q-ST-40C, 6/3/2009.

[ED79A/ARP4754A] “Guidelines for Development of Civil Aircraft and Systems”, EUROCAE ED-79A and SAE ARP 4754A, 21/12/2010.

[EN 50129] “Railway applications – Communications, signalling and processing systems – Safety related electronic systems for signalling”, CENELEC, EN 50129:2003, 7/5/2003.

[IEC 60880] “Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions”, IEC 60880, edition 2.0, 2006-05.

[IEC 61508] “Functional safety of electrical/electronic/ programmable electronic safety-related systems IEC 61508 Parts 1-7, Edition 2.0, 4/2010.

[ISO 26262] “Road vehicles – Functional safety” ISO 26262 Parts 1-9, first edition, 2011-11-15, ISO 26262 Part 10, 2012-08-01

[Knight et al. 1986], J. Knight, N. Leveson, “An experimental evaluation of the assumptions of independence in multiversion programming,” IEEE Transactions on Software Engineering, vol. SE-12, pp. 96–109, Jan. 1986.

[Ladkin et al. 2015] Peter Ladkin, Bev Littlewood “Practical Statistical Evaluation of Software” Version 4 of 01/03/2015 – IEC 61508 – 7 Annex D Working Group.

[Rényi A. 2007] Probability Theory, Dover 2007.

[Rushby et al. 2014] John Rushby, Bev Littlewood, Lorenzo Strigini “Evaluating the Assessment of Software Fault-Freeness” AESSCS Workshop 13 May 2014 Newcastle upon Tyne UK.

[Strigini et al. 1997] Strigini L., Littlewood B. “Guidelines for Statistical Testing” - ESA/ESTEC Study Contract PASCON/WO6-CCN2/TN12 n°10662/93/NL/NB WO6-CCN.

[Thévenod 91] Pascale Thévenod-Fosse, Hélène Waeselynck “An investigation of statistical software testing”, Journal of Software Testing, Verification and Reliability, vol. 1, (2): 5--25, 1991.

[Thévenod 95] Pascale Thévenod-Fosse, Hélène Waeselynck, Yves Crouzet “Software statistical testing”. In Brian Randell, Jean-Claude Laprie, Hermann Kopetz, Bev Littlewood (eds.): Predictably Dependable Computing Systems, Springer Verlag, ESPRIT Basic research Series, pp. 253-272, 1995.

7. Glossary

ASIL Automotive Safety Integrity Level

COTS Commercial Off-The-Shelf (component)

DAL Development Assurance Level

OP Operational Profile

PDF Probability Density Function

PFD Probability of Failure on Demand

SIL Safety Integrity Level

SSIL Software Safety Integrity Level