Functional Safety of Software-Based Critical Systems

Peter Bernard Ladkin
University of Bielefeld CITEC and Causalis Limited

09 March 2011

Introduction

The international standard for functional safety of systems involving programmable-electronic components, IEC 61508, has been valid since the late 1990's, and Version 2 has just become valid [IEC10]. The scope of the standard is, generally, everything except avionics and medical equipment. It is based on the approach, novel at that time, of quantifying and reducing risk until it is acceptable, rather than the then-prevailing paradigm of finding out everything that could go dangerously wrong with a system or subsystems and fixing it so that it doesn't, an approach deemed Sisyphean.

One could argue that the 61508 standard represents a triumph of consensus-building in a notoriously fractious engineering discipline. Equally, one could argue that it exhibits the disadvantages of standardising through consensus when there isn't so much!

The main area of concern to many is software. Reasons are that (1): the current state of software engineering does not enable a statistically-based approach to assessing software risk at anything like the level we need to meet our hopes for acceptably safe operation [BF93,LS93]; and (2): partly in consequence, the requirements on software concern mostly the software-development process rather than properties of the software itself: process quality rather than product quality. The catch here is that we ultimately care about the software product, and there is no reliably-demonstrated causal connection between process methods and product quality, which depends intuitively not just on what has been done, but on how well it has been done.

I shall assume that independent assessment of the fitness-for-purpose of safety-critical systems is enshrined in local law: that not only is there a developer who wishes to know how appropriate hisher system is, but there is an assessor who has to be convinced. Documentation of fitness-for-purpose in some form is required by the standard, in what many call a «safety case».

Broadly, then, improvement to the standard will occur through two pursuits: (1) improving methods for statistical assessment of software risk; and (2) requiring that objective properties of the software product be assessed which directly establish its fitness for purpose.

The Nitty Gritty

Besides these two general ways of improving the standard, there are some more pragmatic considerations for improvement, some lower-hanging fruit. Here is an example.

Methods are recommended for the development of subsystems of various levels of criticality. One of these methods is «Reliability Block Diagrams» (RBD). RBD is a specific technique for assigning and calculating overall reliabilities of composite system parts given reliabilities of components, and assuming independent failures of those components. Another such method is «Formal Methods», apparently a highly-recommended «technique» for the development of the most critical software.

Reliability Block Diagrams, like Fault Trees, is a very specific technique with a very specific purpose whose use and misuse is more or less well-understood, indeed standardised. «Formal Methods» is a term emcompassing the general use of mathematical and logical methods for assessing and designing software products. There are hundreds of formal methods, from methods of specification to methods of assessment of designs, of source code, and of machine code, code development methods, and derivation of test suites. Indeed, to me, the use of Reliability Block Diagrams is a formal method, but it may fall in the range of «semi-formal», also a category.

Surely the standard can be more specific about Formal Methods. Let's see how.

Suppose you specify the requirements for your system in higher-order formal logic, as well as the design. Suppose you then use the theorem prover PVS to prove that each requirement is fulfilled by the design. Have you applied Formal Methods?

Most people would say: yes. Suppose now you had given the task to a summer intern from the local university, who wanted to learn about specification and verification in PVS, and was very pleased with her results and used them as credit for one of her classes. Would that suffice to demonstrate that you had used «Formal Methods» as recommended by the standard? I hope we agree that this would be more problematic.

Suppose you write your software design in UML. Have you used «Formal Methods»? Most people would say: no. Suppose you write your software design in UML and use a commercial C-code generator from a trusted supplier to derive C-code directly from the UML, untouched by human hand. Have you used «Formal Methods» as recommended by the standard? Maybe: yes.

Suppose you had done any, or all, of these things for highly-critical software you were delivering. Does this suffice to assure the quality of your software? Most people would (I hope!) say: no.

There are a number of reasons why one would say no. The intern may not have captured the requirements correctly, or the design. She may have made mistakes in using PVS, and thought something had checked out which didn't. Or maybe the requirements you gave her weren't adequate (weren't fit for purpose). Or maybe the design you used in the product wasn't identical with the design that she checked. Or maybe different people in the team understood the UML differently (it is an ambiguous medium, after all). Or maybe the C-code generator was faulty. Or maybe the C compiler you used wasn't very good. Or maybe someone had mucked with the microcode on the processor, so that certain object-code instructions didn't work the way the compiler was assuming. There are an awful lot of things which can have gone wrong, despite the best of intentions.

So, just using «Formal Methods» does not suffice to obtain the benefits of these methods. One must use them in a targeted manner, to achieve certain goals, and be clear about what one has done, what goal one has achieved. One can be more precise about exactly what kinds of formal methods were used, given that many are out there and are mature enough for routine industrial use.

Use of Formal Methods

With a little help from our friends, we came up with the following list of tasks, and products of those tasks, for which formal methods can be routinely used in the current state of the practice.

(Some terminology: ESCL is the «Executable Source Code Level»: somewhere between design specification and object code, there is at least one place at which sequences of actions of the processor(s) is written down in something which approximates what used to be called a «higher-level language», and there may be many such places. The ESCL is one of these. For example, if

one generates C-code automatically from finite-state-machine descriptions, then the C code might be the ESCL. If one is programming in Java, then both Java source code and Bytecode could be ESCL. The tasks and objects below are predicated on their being one ESCL. If there are more, then there are extra tasks to check the higher-level ESCL against the lower-level ESCL which are omitted here for simplicity. More about the ESCL later. Suffice it to say that there is one, and how you choose it is up to you.)

1. Formal functional requirements specification (FRS)
2. Formal FRS analysis
3. Formal safety requirements specification (FSRS)
4. Formal FSRS analysis
5. Automated proving/proof checking of properties (consistency, completeness of certain types) of FRS and FSRS
6. Formal modelling, model checking, and model exploration of FRS, FSRS

7. Formal design specification (FDS)
8. Formal analysis of FDS
9. Automated proving/proof checking of fulfilment of the FRS/ FSRS by FDS
10. Formal modelling, model checking, and model exploration of FDS
11. Formal determininistic static analysis of FDS (information flow, data flow, possibilities of run-time error)

12. Codevelopment of FDS with ESCL
13. Automated source-code generation from FDS or intermediate specification (IS)
14. Automated proving/proof checking of fulfilment of FDS by IS

15. Automated verification-condition generation from/with ESCL
16. Rigorous semantics of ESCL
17. Automated ESCL-level proving /proof checking of properties (such as freedom from susceptibility to certain kinds of run-time error)
18. Automated proving/proof checking of fulfilment of FDS by ESCL

19. Formal test generation from FRS
20. Formal test generation from FSRS
21. Formal test generation from FDS
22. Formal test generation from IS
23. Formal test generation from ESCL

24. Formal coding-standards analysis (SPARK, MISRA C, etc)
25. Worst-Case Execution Time (WCET) analysis

26. Monitor synthesis/runtime verification

Please suppose for a moment that we got this right and this is a more or less comprehensive categorisation of state-of-the-practice use of formal methods. The question to be asked, with each of these tasks or objects is: did you do it / have you got it? The answers will give a taste to the software assessor of what might be expected from the software. The answers will equally give a sense to the developer of whether the assessor is likely to start out a more-happy or a less-happy camper!

Software Assurance

Being precise about what formal methods are out there is fairly low-hanging. But merely checking a list of things you've done does not of course suffice to assure quality of the software product. To assure the product, you will have needed to have done those things adequately and well, and you will have needed to have done some specific things; not everything in the list, but some of them. And to satisfy an assessor – indeed to satisfy yourself - you will need to have some proof that these things have been accomplished.

A simplified view of assessing a software product for its fitness for purpose is as follows.

1. You want to know what the software is supposed to do and not to do.
2. You want to know that the object code sitting on the processor does this.
3. You want to know that the operation of the software is free from dangerous quirks.

I take it we know well that none of these steps is routine (the standard, of course, has some teens of steps where here there are just three. For discussion, see [L08]).

Software is usually written in a number of steps, with each step having its own language. The first step, usually called requirements, says what the software is to achieve. The last step, usually object code, specifies precisely what actions the processor(s) is(are) to take. We are concerned with safety, that is, absence of dangerous behavior, so the requirements with which we are concerned are the *safety requirements*, which specify the absence of dangerous behavior. That is step 1. And this absence is what a safety engineer wishes to assure throughout the software development, to object code (and the electrical engineer beyond that). That is step 2. Step 3 is a general catch-all for those phenomena which might not have been covered in Steps 1 and 2: a general «did we do everything right?» step.

There is not usually any direct way of comparing object code with requirements. Software production is broken down into a number of smaller steps, and then one can compare the results of each step with those of its predecessor and successor steps. Most products are in language of some sort, usually formal: design specification, source code, object code. And comparisons often take the form of translation: a compiler, for example, translates source code into object code, and a code-generator translates finite-state-machines or other descriptions into, say, C source code. One wants to know that these translations are «faithful», do the job the engineers are expecting of them. However, some steps are not translations. Obtaining the safety requirements in the first place is not usually an act of translation. And neither is checking executable specifications or code against requirements, although translation will likely be involved.

So, how might step 1 look when formulated in a standard?

Proposal A. An unambiguous, rigorous Functional Requirements Specification (FRS) be required. The FRS needs to be checkable for (i) consistency, and (ii) relative completeness. It be required that it is so checked and the methods and results to appear in the safety case.

Unambiguity and rigor are very important, as follows. Suppose your requirement is *the door shall not be shut after the horse has bolted*. What if there are two doors? Do you mean neither door, or just this one, or just that one? That is the condition of unambiguity. Where there are two or more readings, you have to say which is meant. But what is supposed to happen if the horse is *in the course of bolting*? Is the predicate *has bolted* true as soon as he has the idea in his head and has started to act on it, or is it true only when he is at full tilt disappearing over the horizon? These boundary issues are seen only through consideration of what the requirement means when it is put

up against the world, as it were. Things need to be decided more precisely than they were originally formulated. That is the condition of rigor. One can see that these conditions apply to systems in general, not just to software.

What about consistency and relative completeness? Consistency first. Requirements often derive from two or more differently interested parties, and you need to know that these parties are not formulating requirements that preclude each other, for, if they do, any system whatever which you build is going to fall short of one set or another, and if they are to do with safety then falling short is a Bad Thing. This is not necessarily a rare occurrence, as those experienced with requirements specification for even moderately complex systems will confirm. It's necessary and wise to check, hence the requirement to do so.

Relative completeness is more controversial. Here, people can – and do - argue until the cows come home as to what «completeness» is, and some people will even say there cannot be any [Safecrit10]! In fact, certain notions of completeness can be usefully applied as a quality check on the requirements specification [L10.1,L10.2]. For example, that you have written down requirements which preclude all dangerous behavior *which can be expressed in the language in which you have written the requirements*. So, do you shut the door before the horse has bolted? Maybe you do, so the horse won't be tempted; or maybe you don't, so he won't charge through the door and hurt himself. Relative completeness says you must consider the question and decide. What you cannot express at this point is that the lubrication on the hinges releases poisonous elements into the atmosphere which will make you ill if you open and shut the door continuously. Such a phenomenon *isn't yet in the language* and is therefore best left to discovery later.

It is both useful and wise to check whether you have said everything safety-wise that can be said at this point and that is why the relative completeness condition is there.

Finally, this should all be written down and presented to the assessor, indeed some would argue as I would that it be as far as possible a public document, because users or neighbors of the system might well like to know what they are getting themselves into, and may wish to check for themselves.

So much for general safety requirements. Now to Step 2.

Proposal B. (i) The SW Architecture/Design Spec be rigorous. (ii) There be a formal, rigorous, correct demonstration that the SWA/DS fulfils the FRS.

Most software has a specification of some sort as to how it will fulfil the requirements (not just the safety requirements, but all requirements). This proposal entails that if the software is safety-critical, it must have one. If it doesn't have one in the usual sense, then one would be right to be suspicious. However, if this software exists then it can be deemed to be its own SWA/DS, at the source-code level (ESCL) or the object code. Good luck, in that case, with the rigor and the demonstration! But in any case an ESCL may be defined. So the implicit condition that there be an SWA/DS is no extra burden.

The SWA/DS is the first place at which you say how your software is going to look and what it is going to be doing. Again, rigor is important. Here is the first place at which it can sensibly be asked whether the safety requirements are fulfilled. So it is asked, and B(ii) provides the answer.

Proposal C. (i) If the SW is written using a higher-level programming language than machine code, there be defined a language to be called the Executable Source Code Level. (ii) The SW at the ESCL be rigorously, correctly demonstrated to fulfil the SWA/DS.

Most critical software is written using a programming language at somewhat higher level than machine code. That software in that higher-level language forms an intermediate translation step between SWA/DS and object code. You have to say what your source-code level (ESCL) is. There might be many stages at which you could define an ESCL. If you program in Java, do you take your ECSL to be the Java source, or the Bytecode? Your choice, but you have to say. If you write finite-state-machine descriptions, and then press a button to get C-code implementing your state machines, then is the state-machine-description your ESCL, or is it the resulting C-code? Again, your choice but you have to say. Say both, if you like! Then you have ESCL1 and ESCL2 and a formal translation between them. So you can check ESCL1 against SWA/DS and ESCL2 against object code, and then argue that the tool triggered by your button-press produces a «correct» translation from ESCL1 to ESCL2, whatever «correct» means. The obligation proposed is to show your system at ESCL does what you said it should do at SWA/DS.

An example of an industrially-proven approach to Proposals B and C is [B06].

Proposal D. Compilation is defined to be an operation that translates ESCL into object code (OC) , where OC means the executable bytes that sit on the hardware. There be a rigorous, correct demonstration that the OC fulfils the ESCL.

So, the final step from ESCL to object code.

Note that there are here just two steps from SWA/DS to object code, going through ESCL. There might be many. As I just indicated, maybe one has de facto many ESCLs. Say one has three, ESCL1, ESCL2 and ESCL3. Then just one of these will be chosen as **the** ESCL, say, ESCL2. The demonstration that ESCL2 satisfies ESCL1 will then be part of the demonstration in C(ii) that ESCL2 satisfies SWA/DS, and the demonstration that ESCL3 satisfies ESCL2 will become part of the demonstration in D that the object code has the right relationship to ESCL2. So the condition in C and D is fulfilled also when there are many steps between SWA/DS and object code.

Object code isn't the end of the story. There are things which go wrong when running programs on processors that are not necessarily embodied in the meanings of the programming languages, including object code. Such as attempting a divide operation when the denominator is zero, or trying to put a number in a register that is bigger than will fit. These are collectively known as run-time errors. Run-time errors are generally bad; the processor stops, or continues with bad data. If your software is running a safety monitoring program, then stopping that program is a Bad Thing. If your software is calculating and monitoring safe levels of some possibly dangerous phenomenon, then injecting bad data is a Bad Thing. Generally, run-time errors are Bad Things in critical systems. The state of the practice is that there are mature techniques which will preclude run-time errors. They should be used, and this should be documented.

Proposal E. There be a rigorous, correct demonstration that run-time errors do not cause or contribute to causing dangerous failures.

One may do so by eliminating whole classes of run-time errors, as with SPARK, or by trapping and handling the raised exceptions. And so on. Avoiding run-time errors is state-of-the-practice, so it should be standard.

Proposal Cluster F. Testing. Here, many remain somewhat befuddled. Well-known mathematical results say that appropriate software assurance is just not to be had through practical statistical testing [BF93,LS93]. But one does get some kind of assurance of SW fitness for purpose through even routine testing. The hard part – the problem – here is to make precise what kind of assurance

this is, and specify how it is achieved. To cut to the chase, I have here only partial improvements to offer, and not yet a comprehensive proposal.

Given partial improvements and lack of an encompassing approach, can we best proceed by ignoring this issue for the standard? No, there has to be something here to reflect the necessity of good testing, in whatever that necessity consists.

I have a personal story and a moral, like most people who have written software. When I was writing mostly declarative code in the language REFINE, I wrote a time-interval calculation system over a few months. I did unit testing of the functions as I was writing them, performing (a) sanity checks, and, more thoroughly, (b) boundary-case calculations. Integration of the entire system took a programmer, who had no idea what I had written, two hours, mostly performing (a) and (b) at the integration level. She found one boundary case I had missed, I fixed it, and the code was on demo at a major conference the next day, in 1986. It has been used, I don't know how much, by the client in a system which for many years had an annual conference devoted to it, and I have not heard of any further errors. My part was not big, but it would have been far, far more consumptive of time and effort to develop it in C. My unit testing was goal-directed, semantics-directed, and effective. Semantics-directed testing for (a) and (b) seem to me intuitively to be needed for any system. One could argue that they would be supplanted by effective SPARK-like formal methods; maybe so. But they or an equivalent are somehow needed; routine testing is essential for ultra-highly dependable systems. That is my view. But where are these methods? How can we recommend them to all in a standard if they don't exist, or only appear through happenstance, or with the use of particular toolsets, and so on? We can't.

There needs to be some kind of story on what unit testing and integration testing achieve and what is to be shown concerning that achievement. There are some ways, see below, in which certain kinds of testing produce proof of concrete properties of the system. Such methods need to be in the standard.

Evaluating a System Through Running It

Testing involves giving a system some judiciously chosen input values and seeing what behavior is produced. But testing is not the only way a system is evaluated by running it. Systems are also used, and if a system has been used for long enough, maybe one can say enough about the real behavior of a component when running to use that component somewhere else. A technical term for this in the IEC 61508 standard is «*proven in use*». An assessor will very likely be presented with data about a system or system component as it has been run, and must somehow judge the system or component as fit for purpose. This is a problematic area, in which scientific results are often at variance with the reality of attempted evaluation.

Some guidance for «proven in use» is given in the standard, in Part 7 Annex D, and of course it is logically the same guidance which would be given for drawing conclusions from tests. But it is not clear how useful the guidance really is. (Actually, some of us believe it is clear: it's not very useful!)

The guidance is based on classical frequentist reasoning. For example, one may be able to conclude by performing nearly 500,000 operational hours without seeing a failure that one has attained a 99% confidence level that the software will run for 100,000 hours without a failure. But 100,000 hours without failure is the absolutely lowest level at which the standard recognises a reliability condition for critical hardware (SIL 1 HW is to run between 100,000 hours and 1 million hours without failure). And 500,000 operational hours are hard to come by, because one must be sure that the software is exactly the same throughout (no updated versions!). Furthermore, the operational conditions (environment and distribution of inputs) must be **exactly** the same as for the intended

future use, and the oracle for detecting failure must have been perfect! Subtle differences in operational use can invalidate any hoped-for conclusion. And one failure in this time invalidates the positive conclusion one might hope to draw, so one needs to be sure any failure has been detected: a perfect oracle. These conditions on versions and operational profile are strenuous enough to make the guidance all but inapplicable in most cases for software [LL11].

Since that is so, can we relax the conditions somewhat, and relax our confidence comparatively? The answer is simple but firm: no! The general reason, as we all should know, is that discrete systems controlled by software have sharp choice points, at which behavior is highly discontinuous – and lots of them. You cannot just alter an ops profile «a little bit» and hope the resulting behavior will only alter «a little bit»: the statistical criterion is «no failures», not «one or two».

So what can we do that is more practical?

There are considerable sources of uncertainty in evaluating software. There are versions: does a new, modified «update» share relevant properties of the original version, such as freedom from behavior which would or could cause a dangerous failure? Say, we have seen software from such-and-such a developer before, and they have a detailed track record for quality. How confident may we be that their new software under evaluation attains the level of quality we have come to expect from them?

I think that we might well have to approach this issue step by careful step. Not in general, as attempted by Part 7 Annex D, but through many particular instances. Some recent work of Littlewood and Rushby have produced some guidance on how a statistical evaluation explicitly incorporating uncertainty might work, for a very specific architecture [LR11]. I include a brief sketch here with permission.

Developing Confidence in Certain Architectures Through Practical Statistical Reasoning: Littlewood and Rushby [LR11]

Littlewood and Rushby consider certain types of so-called 1oo2 divergent architectures. 1oo2 divergent architectures are common, although not ubiquitous, in critical control systems and protection systems in diverse industries such as nuclear power, aviation, rail and medical devices.

The calculations concern two-channel architectures in which (i) the system as a whole has at least one guaranteed “safe state”, (ii) one channel, Channel A, provides the required functionality in normal operations, and is as complex as necessary, and (iii) the second channel, Channel B, is very simple, provides limited functionality, but in particular the function of bringing the system into a safe state if Channel A fails dangerously (here the system needs to include a perfect oracle for Channel A's “health”). Both Channels are “hot”. They are assumed to be SW-based, so SW reliability models are appropriate. I consider here demand-based functionality only.

Assumption: the system is such that it does not fail dangerously if either Channel A does not fail or Channel B does not fail.

A key property of the architecture is that Channel B is arguably perfect. The functionality implemented in Channel A is assumed to be complex enough that one can construct an argument for its reliability (in terms of probability of dangerous failure on demand, *pdfd*), but its perfection is implausible.

There are two kinds of probability calculation involved, technically called *aleatory* and *epistemic*, the terms used by Littlewood and Rushby.

Aleatory probabilities are those taken to be inherent in the situation itself, given a sample space. For example, in a sequence of rolls of a fair die, it is taken that each face appears according to a uniform distribution: each of the six faces has an aleatory probability of 1/6 of appearing on any given roll. The aleatory probability of, say, a dangerous failure of the system, will in general not be known.

The epistemic probability reflects what an assessor can know about the system behavior, given the evidence. The assessment of the system is based on the epistemic probability as calculated by the assessor.

The crucial practical point is as follows. Although the aleatory probability can likely not be known, its *form* is determined by the architecture, and this form leads to a particularly simple formula for the epistemic probability, which is easily calculated by any engineer (it is pure arithmetic). Furthermore, the level of evidence required to attain high confidence in appropriately high levels of freedom from dangerous behavior appears to be practical.

Let the aleatory probability of dangerous failure on demand of Channel A on a randomly-selected input be denoted by $pdfd_A$, and the aleatory probability that Channel B is not perfect by pnp_B. The probability $pdfd_A$ is the usual kind of well-understood probability random variable. The probability of non-perfection of B, pnp_B , can also be given a frequentist interpretation, as the probability, say, that a SW-subsystem of this size with this kind of functionality, developed using such-and-such methods, is not perfect. Let the (unknown) values of these probabilities be $\boldsymbol{p_A}$, respectively $\boldsymbol{p_B}$.

Littlewood and Rushby show that the probability that the system fails, that is, that Channel A fails dangerously, and that Channel B also fails at this point, is bounded above by $\boldsymbol{(p_A \times p_B)}$.

This means that the probability of dangerous failure of Channel A is conditionally independent of the probability of imperfection of Channel B in the architecture presented. (This calculation is performed on the conservative assumption not only that Channel B is imperfect, given by p_B , but that it actually does fail, given a dangerous failure on demand of Channel A.)

The question is, now, what epistemic probability an assessor should assign to a system failure on demand. Most general is that an assessor has a joint probability distribution $F(p_A, p_B)$. The probability that the system fails on a random demand is then $\int (p_A \times p_B)\, dF(p_A, p_B)$. If the assessors beliefs are independent, so that $F(p_A, p_B) = F(p_A) \times F(p_B)$, then

$$\int (p_A \times p_B)\, dF(p_A, p_B) = \int p_A\, dF(p_A) \times \int p_B\, dF(p_B)$$

$$= P_A \times P_B$$

The question is whether it is reasonable for an assessor to believe that flaws in Channel A and imperfections in Channel B are independent, and the answer is: likely no. For example, misinterpretations of the engineering requirements may be a common cause of Channel A failing and Channel B not being perfect. Similarly, shared mechanisms and higher-level mechanisms could fail, such as the communications of the coordination between Channels A and B to the encompassing system. Such a failure would also constitute a common-cause failure.

However, Littlewood and Rushby point out that common-cause failures are the only plausible mechanism by which the joint distribution $F(p_A, p_B)$ may not be factored as above. They proceed as follows.

Let C be the assessor's a priori estimate of the probability of common-cause failure of both

channels. Then *C* is placed at point (1,1) in the probability space (meaning: both channels fail with certainty during a common-cause failure). It is further assumed that the probability of common-cause failure is independent of the probability of individual happenstance failure of each channel simultaneously.

The probability that the system fails on a random demand is then given by

$$C + (1 - C) \times (P_A)^* \times (P_B)^*$$

where $(.)^*$ denotes the mean value of the posterior distribution.

(Strictly speaking, $(P_A)^*$ is the conditional distribution given that A is not certain to fail, and $(P_B)^*$ the conditional distribution given that B is not certain to be imperfect. This technical subtlety does not affect the use to which we put this estimate below.)

This is a particularly simple formula, and it is applicable, as follows.

An Illustration

There are *claim limits* established in specific safety-relevant or -critical engineering domains; for example, in the atomic power industry in Great Britain there is a claim limit of 10^{-5} for the probability of common-cause faults per operating hour (ophour). This means that one may not claim a lower probability than this for common-cause faults in a fit-for-purpose certification exercise. This gives us a lowest value for *C* of 10^{-5} in the above formula.

To estimate $(P_A)^*$, observe that it is possible to estimate it to $O(10^{-4})$ per op-hour through using statistically-valid random testing ("statistically-valid" means here that test case selection probabilities are *exactly* those that are (to be) encountered during operations, and that the oracle, the device which tells whether a test case has passed or failed, is perfect). [Littlewood-Strigini, 1993]. However, it may be easier to obtain an estimate more like $O(10^{-3})$ in many cases, so let us suppose that we have obtained an estimate for $(P_A)^*$ of 10^{-3}.

To estimate $(P_B)^*$ is a little more unusual. The posterior probability that Channel B is imperfect is affected by the assessment criteria used for Channel B. For example,

(I) The theorem prover, model checker, or whatever other tools used to perform rigorous checks in II may be unsound.
(II) The requirements may be misunderstood. However, this event is included in the estimate of C and thus plays no further role here.
(III) The requirements, use assumptions, and design may be formulated incompletely or incorrectly. This is further divided into three cases:
  (a) the specification is inconsistent (then no system can be built to this spec!);
  (b) elements of the specification may be wrong (although the spec is consistent);
  (c) the formal specification and verification has discontinuities in it, or is otherwise incomplete.

One can argue that an assessor can form *a posteriori* judgements about these probabilities which would be adequate to estimate $(P_B)^*$ to values in the region of $10^{(-3)}$. (To reach this figure from an estimate of the likelihood of II, III, recall we are assuming conservatively that, if Channel B is imperfect, it *will* fail when Channel A fails.)

Suppose, then, that we have achieved a posteriori estimates such as these for $(P_A)^*$ and $(P_B)^*$. It

follows that the probability of dangerous failure of the system is given by

$$C + (1 - C) \times (P_A)^* \times (P_B)^*$$

that is

$$10^{-5} + (1 - 10^{-5}) \times 10^{-3} \times 10^{-3} \cong 1.1 \times 10^{-5}$$

We have thereby constructed a high degree of confidence in the dangerous-failure-freeness of a 1oo2 system by combining two lower degrees of confidence that attach to each channel separately.

Software SILs

The final question, which vexes many in the safety community, concerns the definition and use of the Safety Integrity Level (SIL) for software. SILs classify electronic hardware into 5 levels (an undesignated level, and then SILs 1-4), assigning acceptable dangerous-failure rates to the hardware for so-called «random» failures. The SIL sets the acceptable rate. Use of the SILs as defined in IEC 61508 is controversial, but there is, equally, general agreement that an assignment of acceptable dangerous-failure rate is a necessary step in a risk-based approach to critical-system design and assessment.

The main problem is that no one knows very well how to evaluate dangerous-failure rates for so-called «systematic» failures. Systematic failures are those which result from design. All failures of software are taken to be systematic in this sense. Of course, the design of complex electronics is also a source of systematic failures.

So the situation with software SILs is this. We are well aware that the execution of software causes unwanted behavior sometimes. In critical systems some of this unwanted behavior might be dangerous. An assignment of an acceptable dangerous-failure rate is a necessary step (see above), but cannot generally be performed in the current state of the practice and thus is omitted in the current version of the standard.

This is not a very satisfactory situation. Indeed, some would say it is incoherent. And yet both Versions 1 and 2 of the standard embrace it. That will amount to nearly twenty years of standardisation at least until Version 3, assuming we can make some progress for Version 3. I hope we will be able to make some progress in the next twenty years on practical quantification of dangerous-failure propensity using specific architectures, as pioneered by the work of Littlewood and Rushby described above.

Summary

I have proposed how we may begin to assess critical software based on documented properties of the software, rather than through the process by which it was developed. I have also proposed that a requirement to do so be in the 61508 standard and have indicated some approaches.

I have addressed the vexing question of what one can know about software through running it, either in test situations or as conclusions from previous use. I have suggested that we are not very far along the road to practical methods of assessment, but that there are pioneering techniques which hold promise. Progress in this area might allow us to set and assess acceptable rates of dangerous failure for software and other designs, which is currently a major unanswered question in the risk-based assessment of critical systems.

Acknowledgements

I owe many thanks to Daniel Jackson, John Knight, Bev Littlewood, John Rushby, Martyn Thomas, and the members of the German standards commission DKE GK914 for functional safety of E/E/PE systems, secretary Ingo Rolle and Chairperson Rainer Faller, as well as the associate commission AK914.0.3 Safe Software, Chairpersons Hanns-Joachim Reder and Andreas Armbrecht, for constructive discussion and considerable help!

Literature

[B06] J. Barnes with Praxis, High Integrity Software, Addison-Wesley 2006.

[BF93] R. Butler and G. Finelli, The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software, IEEE Trans. Soft. Eng., vol. 19(1), January 1993.

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

[L08] 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.

[L10.1] P. B. Ladkin, The Parable of the Exploding Apples, post in AbnormalDistribution blog, http://www.abnormaldistribution.org/2010/11/09/the-parable-of-the-exploding-apples/ , November 8, 2010.

[L10.2] P. B. Ladkin, Formal Definition of the Notion of Safety Requirement, post in AbnormalDistribution blog, http://www.abnormaldistribution.org/2010/11/09/formal-definition-of-the-notion-of-safety-requirement/ , November 9, 2010.

[LL11] 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.

[LR11] B. Littlewood and J. Rushby, Reasoning about the Reliability of Diverse Two-Channel Systems in which One Channel is «Possibly Perfect», to appear, 2011.

[LS93] B. Littlewood and L. Strigini, Validation of Ultra-High Dependability for Software-based Systems, Communications of the ACM, vol. 36(11), pp. 69-80, November 1993.

[Safecrit10] Safety-Critical Systems Mailing List, Archive 2010, http://www.cs.york.ac.uk/hise/safety-critical-archive/2010 , University of York, 2010. See especially the threads entitled «software hazard analysis not useful?» and «On the place of formal analysis».