Validation of Ultra-High Dependability for Software-based Systems

Bev Littlewood, CSR, City University, London

Lorenzo Strigini, IEI-CNR, Pisa

Abstract

Modern society depends on computers for a number of critical tasks in which failure can have very high costs. As a consequence, high levels of dependability (reliability, safety, etc.) are required from such computers, including their software. Whenever a quantitative approach to risk is adopted, these requirements must be stated in quantitative terms, and a rigorous demonstration of their being attained is necessary. For software used in the most critical roles, such demonstrations are not usually supplied. The fact is that the dependability requirements often lie near the limit of the current state of the art, or beyond, in terms not only of the ability to satisfy them, but also, and more often, of the ability to demonstrate that they are satisfied in the individual operational products (validation). We discuss reasons why such demonstrations *cannot* usually be provided with the means available: reliability growth models, testing with stable reliability, structural dependability modelling, as well as more informal arguments based on good engineering practice. We state some rigorous arguments about the limits of what can be validated with each of such means. Combining evidence from these different sources would seem to raise the levels that can be validated; yet this improvement is not such as to solve the problem. It appears that engineering practice must take into account the fact that no solution exists, at present, for the validation of ultra-high dependability in systems relying on complex software.

Introduction

Computers are used in an increasing number of applications where their failure may be very costly, in terms of monetary loss and/or human suffering. Examples include control of aircraft, industrial plants, railway and air traffic, weapons and military units, banking and commercial transactions. To reduce the cost to society of operating these computer-controlled systems, dependability requirements for their essential computers are often very high. The most frequently quoted requirement is probably that of less than 10^{-9} "catastrophic failure conditions" per hour of operation (excluding software faults) in civil transport airplanes [9]; in the U.S. Federal Aviation Administration's Advanced Automation System (for air traffic control), one of the key computer systems has a required total unavailability of less than 10^{-7}, or 3 seconds per year; a safety-critical processor for urban trains must fail with a probability of less than 10^{-12} per hour. We shall call these levels of dependability 'ultra-high', for brevity: we are talking about levels that are unusually high, and obviously very difficult to achieve.

The main problem with achieving such levels of dependability is the possibility of subtle design faults. This problem is common to all complex systems, but the prevalence of software technology for the production of very complex**,** digital systems has made it predominantly felt in the software field. Most of the following discussion, therefore, refers directly to 'software', although it is applicable in a more general scope.

Software dependability is well known to be a serious problem. Whereas a mechanical engineer can to some degree decide the reliability of a part whilst designing it (within a certain range of technologically feasible values) by setting design and manufacturing parameters, a software designer (and therefore any designer of computer-based systems) has very little ability to plan the dependability of a product. Among the reasons for this are: i) software failures are due to design faults, which are difficult both to avoid and to tolerate; ii) software is often used to implement radically new systems, which cannot benefit much from knowledge acquired from previous, successful designs; and iii) digital systems in general implement discontinuous input-to-output mappings intractable by simple mathematical modelling. This last point is particularly important: continuity assumptions cannot be used in validating software, and failures are caused by the occurrence of specific, non-obvious *combinations* of events, rather than from excessive levels of some identifiable stress factor.

Given this common unsatisfactory state of the art, adding the requirement of dependability levels that are at the boundary of what is known to be attainable, or beyond, evidently creates a very serious problem.

These ultra-high levels of dependability would also be difficult to attain in a system built purely in hardware. It is true that we now have means of capturing the statistical properties of component life-times, and the effect that component unreliability has on overall system reliability. We know how to employ simple redundancy schemes to obtain high overall system reliability from relatively unreliable components. However, these methods ignore the effects of design faults. Thus the 'solution' we sometimes hear to the software problem - 'build it in hardware instead' - is usually no solution at all. The problem of design dependability arises because the *complexity* of the systems is so great that we cannot simply postulate the absence of design faults, and concentrate on the conventional hardware reliability evaluation. Apart from the increased complexity that may arise from *implementing* functions in software (think of a software-based thermostat switch vs a mechanical one using a bi-metallic strip), if the *functionality* required implies a similar complexity in both cases, a 'purely hardware' system version may be as difficult to validate as a software-based version.

It could be argued here, in favour of the 'hardware solution', that the discipline of building a system purely in hardware might serve as a constraint upon complexity and thus contribute to greater design dependability. The price paid, of course, would be a reduction in functionality. However, the tendency seems to be to move in the opposite direction: to implement as much as possible of the functionality in software, sometimes with an intent to reduce the 'hardware' unreliability (besides producing other benefits, such as ease of modification). Ideally such trade-offs would be made quantitatively, but this is rare. Even when it is reasonable to use software to overcome the reliability limitations of a hardware solution, a decision often follows to take advantage of the use of software to build a *more* complex product, providing more desirable functionality but at the risk of lower design dependability. This tendency to exploit the versatility of software-based systems, at the expense of greater complexity, is understandable in many routine, low-risk applications, but questionable in the safety-critical area.

The problems posed by the procurement of computers with ultra-high dependability requirements are of three kinds:

- *specifying* the dependability requirements: selecting the dependability goals (requirements) that have to be pursued in building the computing system, based on known or assumed goals for the part of the world that is directly affected by the computing system;
- designing and implementing the computing system so as to *achieve* the dependability required. This can be very difficult, as we shall see later;
- *validating*: gaining confidence that a certain dependability goal (requirement) has been attained. This is a problem of *evaluation*.

This paper is mainly about this final problem. More particularly, we want to investigate the limits to the levels of dependability that can currently be validated. Our premise is that it is unreasonable to exempt software from quantitative reliability requirements, whenever dependability and risk are otherwise treated in a quantitative fashion; then, there should be an obligation to convincingly and rigorously demonstrate that such requirements are satisfied. For complex software, there is an often voiced fear that we are now building systems with such high dependability requirements that, even if they were satisfied, they cannot be validated with present methods, or even with all foreseeable methods. Without pretending to give a definitive view, we review the current situation, presenting some rigorous arguments about the limits of current methods.

1. Specification of dependability requirements

The goal of very high dependability in a computer that is a part of a larger system is to help attain some dependability goal for that larger system. Requirements for the computer are then derived from a dependability analysis for the larger system. For instance, to assess the safety of a new industrial plant, we would ideally build a model where the dependability figures for the computers are parameters. Then, to estimate the effect of the dependability of the computer on the safety of the plant, we need an understanding both of the internal operation of the computer (to assess the probabilities of different failure modes) and of the interactions between the computer and the rest of the plant (to know the probabilities that different computer failures will cause failures or accidents). The latter analysis, done before the computer is designed, yields the dependability requirements for the computer system.

Not all critical computing systems have explicit numerical requirements of very high dependability. In some cases, the application of computers started before the attendant risks were clearly perceived, so that no need was felt for special requirements. Military systems often have lower safety requirements than similar, civil applications, since there is a trade-off between safety from enemy action, which is largely based on technological sophistication, and safety against faults of the system itself, which would instead require simple, conservative designs. In certain proposed applications, such as robotic surgery, the human activity that the computer system is replacing is known to be quite failure-prone, so that a moderately dependable robot would often reduce the current level of risk, by among other things allowing intervention in situations where human surgeons could not act.

How exactly it is appropriate to express dependability requirements will vary from one application to another. Some examples are:

- *Rate of occurrence of failures* (ROCOF): appropriate in a system which actively controls some potentially dangerous process;

- *Probability of failure on demand*: suitable for a 'safety' system, which is only called upon to act when *another* system gets into a potentially unsafe condition. An example is an emergency shut-down system for a nuclear reactor;
- *Probability of failure-free survival of mission*. This makes sense in circumstances where there is a 'natural' mission time;
- *Availability*. This could be used in circumstances where the amount of loss incurred as a result of system failure depends on the length of time the system is unavailable. An airline reservation system, or a telephone switch, would clearly have availability requirements.

These examples are by no means exhaustive: for example, the *mean time to failure* (or *between failures*) is still widely used. They are meant to show that selecting an appropriate way of expressing a dependability requirement is a non-trivial task. The measures above are obviously interdependent, and the selection of a particular one, or more, is a matter of judgment. For example, in the case of an air traffic control system, it would be wise to consider the precise way in which unavailability might show itself: infrequent long outages are probably less dangerous than frequent short ones, or simply than certain kinds of incorrect output. It might be better to be also concerned, therefore, with the *rate* of incidents rather than just their contribution to total down time. In the case of flight-control avionics, the probability of safety-related failure is highest at take-off and landing, so that it may be sensible to express the requirement in terms of a probability of a failure-free flight: i.e. as a rate per flight rather than a rate per hour.

In what follows we shall be less concerned with the way in which dependability requirements should be expressed than with the actual numerical levels, and whether we can reasonably expect to be able to say that a certain level has been achieved in a particular case.

Clearly, validating some of the required reliability levels presents formidable problems: some are several orders of magnitude beyond the state of the art. However, that should not be taken to mean that such numbers for required dependability are *meaningless*. The aforementioned requirement for critical avionic systems of 10^{-9} failures per hour represents a probability of about 0.1 that at least one such failure will occur in a fleet of 1000 aircraft over a 30 year lifetime [16]; since this safety-critical system is likely to be one of many on the aircraft, such a probability does not seem an unreasonable requirement.

2. Reliability growth modelling

The most mature techniques for software dependability assessment are those for modelling the growth in software reliability that takes place as a result of fault removal. There is now an extensive body of literature together with some considerable experience of these techniques in industrial practice, and it is now often possible to obtain measures of the operational reliability of a program.

We shall consider here the direct evaluation of the reliability of a software product from observation of its actual failure process during operation. This is the problem of *reliability growth*: in its simplest form it is assumed that when a failure occurs there is an attempt to remove the design fault which caused the failure, whereupon the software is set running again, eventually to fail once again. The successive times of failure-free working are the input to probabilistic reliability growth models, which use these data to estimate the current reliability of the program under study, and to predict how the reliability will change in the future.

Many detailed models exist, purporting to represent the stochastic failure process. Unfortunately, no single model can be trusted to give accurate results in all circumstances, nor is there any way for choosing *a priori* the most suitable model for a particular situation. Recent work, however, has largely resolved this difficulty [1]. We can now apply many of the available models to the failure data coming from a particular product, analyse the predictive accuracy of different models, and gradually learn which (if any) of the different predictions can be trusted.

Some warnings apply here:

- successful predictions depend upon the observed failure process being similar to that which it is desired to predict: the techniques are essentially sophisticated forms of extrapolation. In particular, if we wish to predict the operational reliability of a program from failure data obtained during testing, the selected test cases must be representative (statistically) of the inputs during operational use. This is not always easy, but there is some experience of it being carried out in realistic industrial conditions, with the test-based predictions validated in later operational use [7].

- although we often speak loosely of the reliability of a software product, we really mean the reliability of the product working in a particular environment: the perceived reliability will vary considerably from one user to another. It is a truism, for example, that operating system reliability differs greatly from one site to another. It is not currently possible to test a program in one environment (i.e., with a given selection of test cases) and use the reliability growth modelling techniques to predict how reliable it will be in another. This may be possible in the future but we currently do not even know how to characterise the 'stress' of a software system's environment so that we could relate one to another.

With these reservations, it is now possible to obtain accurate reliability predictions for software in many cases and, perhaps equally importantly, to know when particular predictions can be trusted. Unfortunately, such methods are really only suitable for the assurance of relatively modest reliability goals. The following example, although it refers to modest reliability levels, will serve to illustrate the difficulties related to ultra-high levels.

sample size, i	elapsed time, t_i	achieved mttf, m_i	ti/mi
40	6380	288.8	22.1
50	10089	375.0	26.9
60	12560	392.5	32.0
70	16186	437.5	37.0
80	20567	490.4	41.9
90	29361	617.3	47.7
100	42015	776.3	54.1
110	49416	841.6	58.7
120	56485	896.4	63.0
130	74364	1054.1	70.1

Table 1 An illustration of the law of diminishing returns in heroic debugging. Here the total execution time required to reach a particular mean time to failure is compared with the mean itself.

Table 1 shows a simple analysis of some failure data from the testing of a command and control system, using a software reliability growth model. The question 'how reliable is the program now?' is answered immediately following the 40th, 50th, . . , 130th failures, in the form, (in this case) of a *mean time to next failure*. Alongside the mttf in the table is the total execution time on test that was needed to achieve that estimated mttf. Clearly, the mttf of this system (and hence its reliability) improves as the testing progresses. However, the final column shows a clear *law of diminishing returns*: later improvements in the mttf require proportionally longer testing.

Of course, this is only a single piece of evidence, involving a particular measure of reliability, the use of a particular model to perform the calculations, and a particular program under study. However, this law of diminishing returns has been observed consistently in our own work, and the reasons are not hard to see.

A plausible conceptual model of the software failure process is as follows. A program starts life with a finite number of faults, and these are encountered in a purely unpredictable fashion. Different faults contribute differently to the overall unreliability of the program: some are 'larger' than others, i.e., they would show themselves (if not removed) at a larger *rate*. Thus different faults have different rates of occurrence.

Suppose we adopt a policy of carrying out fixes at each failure, and assume for simplicity that each fix attempt is successful. As debugging progresses, a fault with a larger rate will tend to show itself before a fault with a smaller rate: large faults get removed earlier than small ones. Hence the law of diminishing returns. As debugging progresses and the program becomes more reliable, it becomes harder to find faults (because the rate at which the *program* is failing is becoming smaller), and the improvements to the reliability resulting from these fault-removals are also becoming smaller and smaller. Some dramatic evidence for this interpretation was found in data about some large systems [2], where about one third of the faults only caused errors at the rate of about once every *5000 years* of execution.

It should be noted that these limits to the confidence we can gain from a reliability growth study are not due to inadequacies of the models. Rather they are a consequence of the relative paucity of the information available. If we want to have an assurance of high dependability, using only information obtained from the failure process, then we need to observe the system for a very long time. No improvement in the predictive accuracy of reliability growth modelling will be able to overcome this inherent difficulty.

Let us now return to the assumption that it is possible to fix a fault when it has been revealed during the test, *and to know that the fix is successful*. In fact, there has been no serious attempt to model the fault-fixing operation and most reliability growth models simply assume that fixes are perfect, or average out any short-term reversals to give the longer term trend. It is fairly easy to incorporate the possibility of a purely ineffectual fix (simply introduce an extra parameter representing the probability that an attempted fix leaves the reliability unchanged), but the more realistic situation in which an attempted fix introduces a *novel* fault seems much harder and has not been studied to any great extent. For a safety-critical application, this source of inaccuracy cannot be ignored, as the potential increase in unreliability due to a bad fix is unbounded. In order to have high confidence that the reliability after a fix was even as high as it was immediately prior to the last failure, it would be necessary to have high confidence that no new fault had been introduced. The only way to obtain this confidence seems to be to exercise the software for a long time and never see a failure arise from the fix. Eventually, of course, as the software runs failure-free after this fix, one would acquire confidence in the efficacy of the fix, and thus (taken with the earlier evidence) of the program as a whole. The question of what can be concluded from a period of failure-free working since the last fix is one we shall now consider in some detail.

3. Inferences to be drawn from perfect working

For safety-critical software, it could be argued that acceptance of a product should only be based on the evidence gathered during testing. For the purposes of this analysis, let us ignore possible reservations about whether this testing is representative of operational use. If the software was tested without failure (the 'best news' it is possible to obtain from operational testing alone) for, let us say, 1000 hours, how confident can one be about its future performance, and therefore its suitability for use in a safety-critical application?

Let the random variable T represent the time to next failure of the program under examination, and let us assume that this program has been on test for a period t_0, during which no failures have occurred. Let us assume that the distribution of T is exponential with rate λ, and so has mean time to failure $\theta = \lambda^{-1}$. These assumptions are plausible, since we are interested only in the time to the first failure, and so do not need to be concerned about problems such as possible dependence between faults, one fault 'hiding' another, etc. In fact it is useful to think of the sequence of failures as a Poisson process, even though we shall only be interested in the first event.

There are classical approaches to this problem giving, for example, confidence bounds on parameters such as the probability of failure on demand [17]. However, we believe that the Bayesian approach is more satisfactory since it provides a single subjective probability figure about the *event* of interest, not merely a parameter. Details are in the **Sidebar** (*end of this document*), where it is shown that the posterior reliability function is

$$R(t \mid \text{no failures in time } t_o) \equiv P(T > t \mid \text{data}) = \left(\frac{b + t_0}{b + t_0 + t}\right)^a ,$$

where a and b are parameters that represent the 'prior belief' of the observer about the parameter λ. Clearly, a and b will depend upon the nature of the knowledge this observer has, so it is not possible to say anything specific here about the values they take. However, in the **Sidebar** we see how it is possible to model an observer who comes to this problem with a special kind of 'complete ignorance'. In this case

$$R(t \mid \text{no failures in time } t_o) = t_o / (t + t_o)$$

When we have seen a period t_o of failure-free working there is a 50:50 chance that we shall wait a further period exceeding t_o until the first failure, since $R(t_o \mid 0, t_o) = 1/2$. It follows that if we want the posterior median time to failure to be, e.g., 10^9 hours, and we do not bring any prior belief to the problem, we would need to see 10^9 hours of failure-free working!

The full Bayesian analysis shows clearly the two sources of final confidence in the system dependability: i) the prior belief of the observer (represented by the prior distribution, Gam(a, b)); and ii) the likelihood function representing what was actually observed of the system failure behaviour. The relative paucity of information we can gain from the latter suggests that, to obtain an assurance of very high dependability, an observer must start with very strong *a priori* beliefs. In the **Sidebar** we give an example where, if we require a median time to failure of 10^6, and are only prepared to test for 10^3, we essentially need to *start* with the belief in the 10^6 median.

We have treated this example in some detail because it shows the great gap that exists between the levels of dependability that are often required for a safety-critical system, and that which can feasibly be learned from observation of system behaviour. Certainly it is possible to quibble about the details of how we represent prior ignorance, but such quibbles cannot remove this gap of *several orders of magnitude*.

It is also worth noting that, although we have preferred a Bayesian approach, the conclusions of this section remain the same if a classical view of probability is adopted. Unless the Bayesian brings *extremely* strong prior beliefs to the problem (and these must, of course, have a scientific foundation), both approaches will generally give results that agree to within an order of magnitude. We shall now discuss how sufficiently strong prior beliefs could be justified.

4. Other sources of evidence for validation

Products with very high design reliability do exist, as demonstrated by the statistics collected during the operational use of very many copies. These include some products containing complex software. For example, the Bell system telephone switches in the U.S. reportedly satisfied (before the large outage on the 15th of January, 1990) the requirement of 3 minutes maximum down-time per year [11].

Is it possible to have justified confidence in such extremely successful operation *before* observing it in real use? More precisely, can we identify factors that increase our confidence in a system beyond that which we can realistically gain just from direct testing?

For non-software engineering, there are such factors that help build confidence in this way. First, the design methods used are well tested *and* based on a well understood theory for predicting system behaviour. Following this accepted practice in the design of a novel system makes it more credible that design errors are avoided. However, additional sources of trust are needed and in fact exist.

For example, systems can be over-engineered to cope with circumstances that are more stressful than will occur in operation. Such systems can then be expected to have high dependability in an environment that is merely normally stressful. Of course, arguments of this kind depend upon a basic continuity of the system model, and identifiable stress factors, which are both normally lacking in software. Another example is the common practice of building prototypes which can be debugged extensively. When the operational testing of the prototype is over, there is increased confidence that all relevant design flaws have been exposed. When very high reliability is required, additional design principles can be applied, to guarantee better understanding of the product's behaviour. Among these are:

- step-wise evolution: new designs contain as little change as possible with respect to previous, trusted designs;
- simple design, to avoid errors born of complexity. Although designs cannot be simplified arbitrarily (some minimum level of complexity is dictated by each application's requirements), a general attitude of privileging simplicity in all design decisions improves confidence in the resulting design.

These aids can also be used to obtain confidence in software designs, though usually to a lower extent, as we shall argue in the rest of this section.

4.1. Past experience with similar products, or products of the same process

A new product is usually considered more trustworthy if based on 'proven' technologies or design method. This can be stated more rigorously as a statistical argument. The product is considered as an element of a larger population, whose statistical properties are reasonably well-known: this population may be that

- of 'similar' products for which sufficient operational experience exists;
- of products developed by the same organization, or using similar methods (process-based arguments).

We may be interested in forecasting the dependability of a new design, or of a new instance of a design (say an individual installation of a computer system). In the former case, we can use experience with samples of products built using other, similar designs; in the latter, we can use experience with other instances of the same design. Two separate sources of error are present in such forecasts: i) inaccuracy caused by inference from the sample rather than the whole population; ii) uncertainty as to whether the population from which the sample has been drawn actually includes the new product that is being evaluated: there may be some important aspect of the new product which has not been seen before. Such statistical arguments can only be applied, therefore, when there is a high degree of similarity between the old products and the new one.

It is thus a commonly accepted belief that the risk that a new product will behave in ways unforeseen by the designer increases with the degree of novelty of the design. Unfortunately, for software we do not have a clear understanding of how perceived differences between two products should affect our expectations.There are obvious *factors* of similarity (application, developing organization, methods and tools used), but for evaluation we would need a convincing *measure* of similarity. No such measure exists. In our case, we would like to be able to measure the degree of similarity of software development processes, of the resulting software products, and even of the underlying problems that the products are being created to solve. Would it be reasonable to assume, for example, that building an auto-land system is similar to building a fly-by-wire system? And if so, what is the degree of similarity?

Even if we can resolve these difficulties so that the statistical model can be used, we still face the problem of the amount of evidence available, and the results are rather disappointing. Consider, for example, the question: how much can we trust a particular product if it is the output of a methodology that has *always* been 'successful' in the past? can we assume that the present application of the methodology has been successful?

The problem is formally stated, with some loss of realism, as follows. The applications of the methodology to each new development are independent trials, with the same probability p of success (i.e., compliance of the product with our ultra-high reliability requirements, whatever they are). Then, if we start with prior ignorance about p, our posterior expectation about p, after observing n successes out of n trials, would be [15]:

$$E(p \mid n \text{ successes in } n \text{ trials}) = (n+1)/(n+2).$$

With the usual small values of n, there is a significant probability that the next application of the methodology will not be 'successful', i.e., it will not achieve ultra-high operational reliability.

If, on the other hand, we could bring to the problem a prior belief that p were very small, the posterior belief would be correspondingly strong. However, in such a case the additional confidence to be gained from an additional, successful trial would not be significant.

The assumptions that go into this result can, of course, be challenged in detail. However, this formal analysis at least captures the magnitude of the difficulty facing us. In fact, the assumptions are probably optimistic. For example the 'similarity' assumption is likely to be violated. Also, it must be admitted that, for most software development methodologies, there is little quantitative evidence of 'success' in the sense indicated: hence the ‘prior ignorance’ analysis seems the most appropriate one. The history of Software Engineering is strewn with very strong claims for the efficacy of particular methodologies, which, in the formalism used here, would be represented by large prior probabilities of success. However, these claims are not usually supported by much evidence, and we do not believe that such strong prior beliefs can usually be justified. In conclusion, claims about the dependability of a product, based solely upon the efficacy of the development methodology used, must be justified by formal analysis of statistical evidence: at this stage, only modest claims can be supported.

4.2. Structural reliability modelling

Structural reliability models allow one to deduce the overall dependability of a product from the dependability figures of its parts. This method has given good results as a design aid to obtain high dependability in mechanical and electronic systems . A model is built to describe the effects of the behaviours of individual parts on the whole product, and reliability data for the individual parts (obtained at lower cost than would be needed for the whole system, possibly from use of the same modules in previous products) can be used to estimate the parameters of the model. Thus it can be argued that the reliability goal has been attained, without a statistical test of the product.

Structural modelling has obvious limitations with respect to design faults, and software in particular. It relies on the correctness of the model used, which is often directly derived from the design itself. Thus a Markov model of software execution will typically assume transitions among executions of different modules to be as planned by the designer, so neglecting certain categories of typical errors, e.g., upsets in the flow of control, or omissions in the specification. In addition, the probabilities to be used as parameters are difficult to estimate.

In the context of ultra-high reliability, one important aspect of structural modelling is its application to modelling fault tolerance obtained through design diversity [4]. Here, two or more versions are developed by different teams in the hope that faults will tend to be different and so failures can be masked by a suitable adjudication mechanism at run time. The concept owes much to the hardware redundancy idea, where several similar components are in parallel. In the hardware analogy, it is sometimes reasonable to assume that the stochastic failure processes of the different components in a parallel configuration are *independent*. It is then easy to show that a system of arbitrarily high reliability can be constructed from unreliable components.

In practice, of course, the independence assumption is often unreasonable: components tend to fail together as a result of common environmental factors, for example. In the case of design diversity, the independence assumption is unwarranted: recent experiments [3, 6, 12] suggest that design diversity does bring an increase in reliability compared with single versions, but this increase is much less than what completely independent failure behaviour would imply. One reason for this is the similarity of errors in the construction of more than one version. Another reason [14] is that, even if the different versions really are independent objects (defined in a plausible, precise fashion), they will still fail dependently as a result of variation of the 'intrinsic hardness' of the problem over the input space. Put simply, the failure of version A on a particular input suggests that this is a 'difficult' input, and thus the probability that version B will also fail on the same input is greater than it otherwise would be. The greater this variation of 'difficulty' over the input space, the greater will be the dependence in the observed failure behaviour of versions.

Since we cannot *assume* independence of failure behaviour for component versions, reliability models for fault-tolerant software must take into account failure dependency between diverse software variants. The degree of dependence must be estimated for each particular case. Trying to measure it directly from the operational behaviour of the system would lead us back to the 'black-box' estimation problem that has already defeated us [16]. If we wish to estimate it indirectly, via evidence of the degrees of version dependence obtained from previous projects, we are once more in the same difficulties we have seen earlier: past experience of successfully achieving low version dependence does not allow us to conclude that we shall be almost certain to succeed in the present instance.

4.3. Proofs and formal methods

There are many roles that formal methods may play in the implementation and validation of highly dependable software. Some of the techniques that have been labelled 'formal methods' are merely particular examples of software development process, such as we have considered earlier: formal specification techniques are an example. Perhaps the most important aspect of formal methods for ultra-high dependability concerns the notion of proof.

There are at least two important applications of proofs. One can prove that a design has certain properties: a protocol is deadlock-free, for example, or a given output will only be produced if there is consensus between two modules. Such properties can be used to simplify the task of dependability evaluation. Secondly, it is possible to prove that there is consistency between different stages in the evolution of the product from (formal) specification to final implementation.

There are two problems with proofs [5]. Firstly, proofs are subject to error: direct human error, in proofs by hand or with the aid of semiautomatic tools, and/or error by the proof-producing software in machine proof. In principle, one could assign a failure probability to proofs, and use this in a probabilistic analysis. In practice, such probabilities would be quite difficult to assign, and to incorporate into a dependability evaluation. Even if we knew that a given proof were erroneous, we would not know anything about the true state of things, which might range from a very high to a very low failure rate.

We believe that this problem of error in proof can be greatly reduced by improvements in techniques. However, a more fundamental second problem would remain. Proofs apply to some formal description of the product (a design at some stage of refinement, typically) and its environment rather than to the product itself in the real world. This is well known in non-software engineering, where design margins can be employed to guard against inadequacies of the formal model: all the uncertainties in the mapping of the design into reality are lumped into an additional stress. The difficulty with trying to do something similar for software-based systems is, of course, that the notion of stress is less well defined, and may sometimes not be definable. Indeed, for those macroscopic phenomena that are known to behave as 'stress' (load in operating systems, for example), there is no clear understanding of the exact nature of their impact on the behaviour of computers. Thus, the so-called 'stress testing' of software is at best seen much more as an aid in debugging than as a means of dependability evaluation. This problem of possible mismatch between the model and reality is not helped by formal methods (although they can help in clarifying the implications of the top-level specifications), and will remain as a source of uncertainty. We believe that proofs may eventually give "practically complete" assurance about software developed for small, well-understood application problems, but the set of these problems is now empty and there is no way of foreseeing whether it will grow to be of some significance.

The issue of when a design is simple enough that it can be trusted to be fault-free based only on logical arguments about the design itself can be treated, again, as a peculiar kind of process-based argument. Convincing statistics about large populations of small, *similar* programs could be used to substantiate such claims, but this approach presents the difficulties discussed in Section 4.1.

4.4. Combination of different kinds of evidence

We have seen how we can draw only quite weak conclusions from even extensive evidence of a product working without failure, or of a development process demonstrating its ability to produce highly dependable products. It is often the case that many such sources of evidence are available: for example, if a software product has been tested for a given time without failures, has been produced by a company with a long, successful experience in the same field, and uses design diversity in some critical parts, what can be concluded about its dependability? A problem with most attempts to argue that a product satisfies very strict dependability requirements is that information gained from different sources is combined in an informal, non rigorous, and often unconvincing manner.

This problem surfaces in many forms. It arises when discussing how to allocate resources between static analysis of software and dynamic testing. It arises when deciding whether to use new or older, more proven technology. For example, in some space applications, relatively old computers with low throughput and small memory are used. The limitation to old technology is not due solely to the small market available, but also to the slow certification process, meant to assure the dependability of these components. Arguably, faster CPUs and larger memories would allow more use of modern programming concepts, which do not emphasize thrifty use of memory and CPU cycles. Another example is the debate about using low-level or high-level languages in small, safety-critical systems. High-level languages offer the potential for fewer coding errors, but more dependence on compiler technology.

In all these cases, a project manager can control many variables known to affect the dependability of the final product, but varying one of them to improve the product's dependability also affects other variables so as to possibly indirectly *decrease* dependability. Without a clear, quantitative model of how all these variables contribute to the final result, it is difficult to make rational decisions to improve dependability (let alone to optimize it).

In principle, once again, a Bayesian approach can be applied to this problem. Assume that we know that a product was developed using a process that has shown itself to be successful on previous similar products, and a long operational test of the product shows no failures. Clearly, we would be more confident here than we would have been if we had only one of the two pieces of evidence. Moreover, we would have a clear, verifiable representation of how we gained this higher confidence. However, we would not be *very* much more confident. We have seen earlier that if we start with a reasonable prior belief and observe evidence for the efficacy of a process, our posterior belief will not be very strong. Now the latter is the belief we shall use as our prior before observing the test results on the particular product we have built. Such a prior, followed by evidence of failure-free working for a long time, will result in a posterior belief about the system dependability that is better than we obtained in an earlier section with the 'ignorance' prior, but it will not be a dramatic improvement.

4.5. Validation by stepwise improvement of a product

In *conventional*, non-software, systems trust can often be built up by considering their history, if they are the product of a series of stepwise improvements. The designers of each new generation of product make only small improvements to the previous design. The small improvement typically implies the redesign of some part, in such a way that the overall system dependability is almost certainly not reduced and is likely to be improved. For example, if the motivation for the new design is to improve on the reliability of the previous one, the candidates for change are those parts that proved to be 'weak links' in the old design. The redesigned parts can be proven to be better than the old ones by normal testing techniques (including stress or accelerated testing, where considered trustworthy). Then a simple structural reliability model of the system allows one to conclude that improving the parts did improve the whole. A series of small improvements, each left in operation long enough to yield a convincing statistical record, *could* lead to trustworthy ultra-high dependability starting from a product with ordinary dependability.

Even for non-software systems, however, this conclusion is not always well-founded. For the above argument to be convincing, we need to be sure that the system behaviour can be described by well-behaved models, and there are reasons why this may not be the case. Reinforcing one part of a mechanical structure can sometimes redirect the strain to some other part that has not been designed to accept it. If the non-software system is sufficiently complex that there is a significant chance of the introduction of design faults during the stepwise 'improvements', we cannot assume increasing dependability. These remarks apply *a fortiori* to software itself, where there is evidence that for some large systems the reliability approaches an asymptote as the history of fixes becomes very long. There is uncertainty whether a particular fix will make things better or worse, and eventually a stage is reached where the expected reliability improvement from a fix is zero: we are as likely to reduce the reliability of the software as to improve it [2].

If the policy of stepwise improvement is to be convincing, at the very least designs must be kept such that the interactions among parts, and the effect of each part on global dependability, are simple and well-understood. Only then can validation by analysis and modelling of individual products be trusted. Two common applications of this principle are avoiding non-obvious error propagation paths (e.g. by physical separation of different software-implemented functions into separated hardware units, and, in multiprogrammed computers, by hardware protection of process memory); and building safety (or security) monitors as independent as possible of the main control systems.

In conclusion, evolutionary design practice is clearly desirable (when feasible), but it does not provide a *general* answer to the problem of assuring ultra-high dependability.

5. Discussion

A common theme underlying much of our discussion is that of the limitations of formal, mathematically tractable models. We describe reality by simplified models, suitable for the application of some rigorous (possibly probabilistic) reasoning technique. The validity of our results then depends on the validity of the modelling assumptions. There is always a chance that reality will violate some of these assumptions: we need to form an idea of the probability of this happening. Examples of such assumptions are:

- in reliability growth modelling: regularity of the growth process, and in particular, realism of the test data with respect to the intended operational environment;
- in process-based statistical arguments: representativeness of the new product with respect to the known population from which statistical extrapolations are made;
- in structural reliability modelling: correctness of the parameters, completeness of the model (e.g., independence assumptions, transitions in a Markov model, branches in a fault tree);
- in proofs of correctness: conformance of the specification to the informal engineering model.

[19] describes this issue, as it arises normally in the design of fault-tolerant systems, in terms of *assumption coverage*. In a typical example, a designer can assume that only certain component failure modes are possible, such that they are provably (or with some known probability) tolerated by the fault-tolerance mechanisms in the system. By thus neglecting the possibility of some failures a simpler design becomes sufficient, requiring less computing time and less hardware, which of course tends to improve reliability. In evaluation, one can separately evaluate the probability of such an assumption actually holding, P(A), and use it as a parameter in the expression of failure probability for the system. At the very least, one can obtain a conservative estimate by assuming that the system will fail whenever the assumption is violated.

In this example, the 'coverage' of the restrictive fault assumption, P(A), can probably be estimated by tests on the individual components. Using the assumption in design is a 'divide and conquer' approach to attaining and proving the required dependability. This may prove more effective than designing to tolerate arbitrary, even very unlikely, faults. The important consideration here is that a rigorous mathematical argument can be used to choose one or the other approach. Of course, other assumptions underlie such an argument, e.g. a correct implementation of the design, so this way of reasoning does not by itself simplify the validation of ultra-high dependability. Its use is, rather, in providing a clear description of some sources of uncertainty in the evaluation. In the probabilistic terminology that we have used widely, we are expressing the high-level probability statements of dependability in terms of lower level probabilities: *conditional* probability statements about the system behaviour *assuming that* certain fault hypotheses are true, together with probability statements about the truth of these hypotheses.

The approach just outlined shows how considerations of validation can be taken into account rationally in the design process. Indeed, there *are* well-known principles of good design that can aid the system validation process. Absence of unnecessary complexity, reuse of tried and trusted designs with known dependability, an evolutionary rather than revolutionary approach to the creation of new products, separation of safety concerns, are all examples of known good practice not only for attaining high dependability levels, but for validating them as well. In particular, the separation of safety concerns has an important role: isolating safety-critical functions in (as near as possible) separate, simple subsystems simplifies safety evaluation; directing the analysis of a design specifically to look for dangerous failure modes (rather than for every deviation from the nominal required behaviour) may make the analysis less arduous and hence more trustworthy [13]. Such general principles are already part of many standards and of normal practice, although there are trends to the contrary, like the plans to integrate all avionic software in a few large computers per aircraft.

However, these methods do not solve the problem of obtaining *quantitative*, probabilistic statements about dangerous failures. 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. Rather, the problem is in the paucity of evidence: informally, in order to believe in these very high dependability levels we need more evidence than could reasonably be expected to be available. Work on the formalization of the combination of evidence obtained from different sources could help to certify slightly higher levels of dependability than currently possible. However, it is fraught with difficulties, and it does not seem likely to move us much closer to the levels of dependability that we call 'ultra-high'.

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 [20,21]:

> ' . . techniques for estimating the post-verification probabilities of software errors were examined. The objective was to develop numerical requirements for such probabilities for digital computer-based equipment and systems certification. The conclusion reached, however, was that currently available methods do not yield results in which confidence can be placed to the level required for this purpose. Accordingly, this document does not state post-verification software error requirements in these terms.'

This seems to mean that a very low failure probability is *required* but that, since its achievement cannot be proven in practice, some other, *insufficient* method of certification will be adopted. Indeed, the remainder of the document only gives guidelines to 'good practice' and documentation, which as we have seen fall far short of providing the assurance needed. In other words, directives are given to support *qualitative* evaluation only. This approach is not new. The *design* reliability of hardware systems has traditionally not been evaluated as part of certification procedures: reliability estimation has referred only to physical failures. Sometimes such a view was reasonable, since the complexity of the design was low and it could be plausibly argued that it was 'almost certainly' correct. In addition, evolutionary design was the rule. Similar considerations apply for the design of *environments* containing computers, like industrial plants or space systems: actually, probabilistic risk assessment techniques are not yet widely applied in some high-risk industries [10]. Rather, qualitative techniques, such as Failure Mode and Effect Analysis, are trusted to find 'weak links' in designs and cause adequate corrective actions. There is probably a need for further research for improving the practice of Probabilistic Risk Assessment: conventional 'hardware' reliability or safety evaluation does not consider design faults, and yet many accidents in complex systems, not involving software, appear to be caused by the design, either of the physical system or of the procedures for its use [18].

In support of the qualitative approach, it might be argued that stating requirements so high that they cannot be used in the acceptance or the certification of a product serves more to pave the way for future litigation than to safeguard current users. However, ultra-high dependability requirements for a computer typically result from a probabilistic risk assessment of the environment where the computer has to be deployed: if the computer does not satisfy its requirements, some requirement for the environment itself cannot be proven to be satisfied. As a very minimum, it would seem necessary to perform sensitivity and worst-case analyses on the models used for risk assessment, with respect to the uncertainty on the computer dependability figures. For instance, if the requirement of a 10^{-9} probability of catastrophic failures for systems in civilian aircraft were derived from such a risk analysis, it would have to apply to all failures, including those caused by design. The problem with placing complex software in a position where this requirement applies would then be, of course, that there would be no way to demonstrate achievement of this requirement.

This leads to another way to avoid the problem, that is, to designing environments in such a way that ultra-high dependability is *not* required of software. For instance, guidelines for the U.K. nuclear power industry are that the assumed probability of failure on demand for safety-critical software can never be better than 10^{-4} [22]. This leads back to the accepted practice of separate and redundant safety mechanisms. Of course, probabilistic risk assessment and requirement specification are complex issues. It is difficult to model the effects of computer failure on complex environments, and there may be no explicit statement of desirable levels of risk. Even the seemingly obvious requirement that the introduction of a new computer must not make the risk level any worse is often ambiguous. Consider for instance an alarm system, intended to signal some dangerous situation to a human operator. Its contribution to the global risk level is a function of the probabilities of its different failure modes (false alarms vs missed alarms), its human interface and the training of operators, etc. If one were to procure a new model of alarm system, risk assessment would be relatively easy only if the new model mimicked the old one almost perfectly. In other words, whenever computers are introduced with new functions, specification of their requirements is subject to difficulties and uncertainties in the analysis of the environment, not unlike those in the dependability analysis of the computer itself. Obviously, a prudent solution is to overstate dependability requirements for the computer, but this leads us back to the very problem from which we started.

Uncertainty is an unavoidable consequence of having computers (or any very complex design) as parts of hazardous but reasonably safe environments. In view of this, one may question the policy of using computers in such environments. A few distinctions may be useful here. In the first place, there are those existing environments that are known to be within the required risk level in their current configuration. The substitution of components of that configuration with new computers may or may not imply ultra-high dependability requirements for the latter. In view of the considerations which we have presented, these kinds of requirements should be seen with scepticism. A different case presents itself when the introduction of a new computer is meant to obtain a substantially new and better environment. As an example, computers are introduced in "fly-by-wire" aircraft not merely to reproduce the somewhat limited functionality of conventional control systems, but also to provide extra functions (such as some degree of protection against pilot error). Then, there is usually no question of avoiding the use of the new computer, but a proper risk analysis should take into account the difficulties discussed in this paper. An attempt should usually be made to design for validation, as outlined above. However, the comparatively low levels of dependability that can be validated should always be taken into account in evaluating the costs and benefits of the new environment proposed.

Acknowledgments

This work was supported in part by the Commission of the European Communities, in the framework of the ESPRIT Basic Research Action 3092 "Predictably Dependable Computing Systems". This paper owes much to discussions with other members of the project, in particular during a project workshop held on the topic in October 1990 at LRI-University of Paris Sud in Orsay. We are grateful to John Rushby for some valuable comments and suggestions on an earlier version of this paper.

References

1. A.A. Abdel-Ghaly, P.Y. Chan and B. Littlewood, "Evaluation of competing software reliability predictions," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp.950-967, 1986.

2. E.N. Adams, "Optimizing preventive service of software products," *IBM Journal Research and Development*, vol. 28, no. 1, pp.2-14, 1984.

3. T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding, "Software fault tolerance: an evaluation," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp.1502-1510, 1985.

4. A. Avizienis and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and experiments", IEEE Computer, 17, pp. 67-80, 1984.

5. J. Barwise, "Mathematical proofs of computer system correctness", *Notices of the AMS*, 36, pp 844-851, Sept 1989.

6. P. G. Bishop and F. D. Pullen. "PODS Revisited - A Study of Software Failure Behaviour," in *Proc. 18th International Symposium on Fault-Tolerant Computing*, pp. 1-8., Tokyo, Japan, 1988.

7. P.A. Curritt, M. Dyer and H.D. Mills, "Certifying the reliability of software," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp.3-11, 1986.

8. M.H. de Groot. *Optimal Statistical Decisions,* New York, McGraw-Hill, 1970.

9. Federal Aviation Administration Advisory Circular , AC 25.1309-1A.

10. B.J. Garrick, "The Approach to Risk Analysis in Three Industries: Nuclear Power, Space Systems, and Chemical Process," *Reliability Engineering and System Safety*, vol. 23, no. 3, pp.195-205, 1988.

11. F.K. Giloth and K.D. Prantzen. "Can the reliability of digital telecommunication switching systems be predicted and measured?," in *Proc. 13th International Symposium on Fault-Tolerant Computing,* pp. 392-397, Milano, Italy, 1983.

12. J.C. Knight and N.G. Leveson. "An empirical study of failure probabilities in multi-version software," in *Proc. 16th International Symposium on Fault-Tolerant Computing,* pp. 165-170, Vienna, Austria, 1986.

13. Nancy Leveson, "Software safety in embedded computer systems", CACM, Vol 34, No 2, pp34-46, 1991

14. B. Littlewood and D.R. Miller, "Conceptual modelling of coincident failures in multiversion software," *IEEE Transactions on Software Engineering*, vol. SE-15, no. 12, pp.1596-1614, 1989.

15. B. Littlewood, "Limits to evaluation of software Dependability', in *Software Reliability and Metrics* (Proceedings of 7th Annual CSR Conference, Garmisch-Partenkirchen), Eds N. Fenton, B. Littlewood, pp81-110, London, Elsevier.

16. D.R. Miller. "The role of statistical modelling and inference in software quality assurance," in *Software Certification,* pp. 135-152, Barking, Elsevier, 1989.

17. D.L. Parnas, A.J.van Schouwen and S.P. Kwan, "Evaluation of Safety-Critical Software," *Communications of the ACM*, vol. 33, no. 6, pp.636-648, 1990.

18. C. Perrow. *Normal Accidents - Living with High Risk Technologies,* New York, Basic Books, 1984.

19. D. Powell. Fault Assumption and Assumption Coverage, ESPRIT PDCS Technical Report, 1991.

20. RTCA/EUROCAE: Radio Technical Commission for Aeronautics and European Organization for Civil Aviation Electronics*Software Considerations in Airborne Systems and Equipment Certification,* , Doc DO178A/EUROCAE ED-12A, 1985.

21. RTCA Committee SC-176, *Software Considerations in Airborne Systems and Equipment Certification*, Draft DO-178-B.7, July 27, 1992, RTCA (Requirements and Technical Concepts for Aviation - 1140 Connecticut Ave, NW, Suite 1020, Washington DC 20036).

22. N. Wainwright, 'Software aspects of digital computer based protection systems', *Assessment Guide AG3*, Nuclear Installations Inspectorate, Issue 1 (draft), 1991.

23. G. Wright and P. Ayton. *Judgemental Forecasting,* Chichester, John Wiley, 1987.

Sidebar

The Bayesian approach to statistics takes the view that probabilities are subjective; they represent the strength of belief of an observer about whether certain events will take place. It is assumed that this observer will have some prior beliefs which will change as a result of seeing the outcome of the 'experiment' (i.e. the collection of data). Bayesian theory provides a formalism for this transformation from prior to posterior belief. For our own purposes, the major advantage of the theory is the idea that prior belief can be quantified and incorporated with experimental evidence into a final probability assessment..

Suppose that we have, in the example in the main text, seen x failures of the program during the period of testing t_0. Bayes theorem states

$$p(\lambda \mid \text{data}) \propto p(\lambda).\ p(\text{data} \mid \lambda)$$

where the distribution $p(\lambda)$ represents the prior belief about the rate of occurrence of failures, λ, and $p(\lambda \mid \text{data})$ represents the posterior belief after seeing the data.

This use of Bayes theorem shows how our *a priori* belief changes into our *a posteriori* belief via use of the *likelihood function*, $p(\text{data} \mid \lambda)$, for the evidence we have obtained from the experiment: here proportional to $\lambda^x \exp(-\lambda t_0)$ because of the assumption of a Poisson process.

Notice the natural way in which we can here express confidence statements about λ in the form of probability statements in which λ itself is a random variable. This contrasts with the rather contrived interpretation of a confidence statement in the classical context, which must invoke a set of experiments which *could hypothetically* have been performed, and which treats the quantity calculated from the data as the sole random variable about which confidence is being expressed. This advantage of the Bayesian approach becomes even more clear as we go on to obtain probability statements about T, the residual time to next failure.

To proceed further, we need to formalise the representation of our prior belief about λ, i.e. we need to decide on a form for $p(\lambda)$ above. There are good arguments [8] that this should be chosen from the *conjugate family* of distributions for our particular likelihood function: this is the (unique) *parametric family* of distributions which has the property that both posterior distribution and prior will be a member of the same family. The informal idea is that there should be a certain homogeneity in the way in which our beliefs change as we acquire more evidence about the unknown rate parameter λ. Under conjugacy, such homogeneous change is represented by changes solely in the values of the hyper-parameters of the conjugate family.

The conjugate family here is the gamma, $\text{Gam}(\alpha, \beta)$, which has probability density function $\beta^\alpha \lambda^{\alpha-1} e^{-\beta\lambda} / \Gamma(\alpha)$, where the hyper-parameters α, β are both positive. If we let $\text{Gam}(a, b)$ represent our prior belief (for some suitable choice of a and b), our posterior belief about λ , $p(\lambda \mid x, t_0)$, is represented by $\text{Gam}(a + x, b + t_0)$. This fits in well with intuition: for example the expected value, $E(\lambda)$, changes from a/b to $(a + x)/(b + t_0)$, so that observing a small number of failures, x, in a long time t_0, will cause the posterior expected value to be smaller than the prior.

We can now make probability statements about λ, and about T itself:

$$\begin{aligned} p(t \mid x, t_0) &= \int p(t \mid \lambda)\ p(\lambda \mid x, t_0)\ d\lambda \\ &= \frac{(a + x)(b + t_0)^{a + x}}{(b + t_0 + t)^{a + x + 1}} \end{aligned}$$

It follows that the reliability function is

$$\begin{aligned} R(t \mid x, t_o) &= P(T > t \mid x, t_o) \\ &= \left(\frac{b + t_0}{b + t_0 + t}\right)^{a + x} \end{aligned}$$

and in our case, when $x = 0$,

$$= \left(\frac{b + t_0}{b + t_0 + t}\right)^{a}$$

The naturalness of the Bayesian method is bought at some price. Its subjective nature forces the users to describe formally their prior belief about the parameter(s): in our case, to choose the hyper-parameters a and b. However, it is rarely the case that we do not have *some* prior belief, and in this example it is possible to think of various sources for such belief: experience of developing similar software systems in the past, previous experience of the efficacy of the software development methodology used, etc. Eliciting such subjective prior belief from people in order to arrive at values for a and b is a non-trivial exercise, although there has been some recent progress in semi-automatic elicitation procedures [23].

It is often tempting to avoid the difficulty of selecting values of hyper-parameters of the prior distribution to represent 'your' beliefs, by devising instead a prior distribution which represents 'total ignorance'. Unfortunately this is difficult. One argument here runs as follows. Since $\mathrm{Gam}(a + x, b + t_o)$ represents our posterior belief about the rate, *large* parameter values represent beliefs based on considerable data (i.e. large x, t_o). To represent initial ignorance, therefore, we should take a and b as small as possible. Now $\mathrm{Gam}(a, b)$ exists for all $a, b > 0$. If we take a and b both very small, the posterior distribution for the rate is approximately $\mathrm{Gam}(x, t_o)$, with the approximation improving as $a, b \to 0$. We could therefore informally think of $\mathrm{Gam}(x, t_o)$ as the posterior in which the data 'speak for themselves'.

This argument is fairly convincing if we are to see a reasonable amount of data when we conduct the data-gathering experiment. Unfortunately it breaks down precisely in the case in which we are interested here: when $x = 0$ the posterior distribution for the rate is proportional to λ^{-1} and is thus improper (i.e., it yields a total probability mass greater than 1). Worse, the predictive distribution for T is also improper, and is thus useless for prediction.

Here is a possible way forward. Choose the improper prior

$$p(\lambda) \equiv 1$$

giving the posterior

$$p(\lambda \mid 0, t_o) = t_0 \exp(-\lambda t_o)$$

which is a proper distribution. More importantly, the predictive distribution for T is also proper:

$$\begin{aligned} p(t \mid 0, t_o) &= \int p(t \mid \lambda)\, p(\lambda \mid 0, t_o)\, d\lambda \\ &= t_o / (t_o + t)^2 \end{aligned}$$

The reliability function is

$$\begin{aligned} R(t \mid 0, t_o) &\equiv P(T > t \mid 0, t_o) \\ &= t_o / (t + t_o) \end{aligned}$$

and in particular, $R(t_o \mid 0, t_o) = {}^1/_2$: i.e. we have a 50:50 chance of seeing a further period of failure-free working as has already been observed.

The conclusion here is that observing a long period of failure-free working does not *in itself* allow us to conclude that a system is ultra-reliable. However, it must be admitted that the prior distribution here is rather unrealistic. Therefore, let us consider the case where the observer has genuine prior beliefs about λ, perhaps arising from his or her faith in the efficacy of the methods used to develop the software. Consider the following example: the reliability requirement is that the median time to failure is 10^6 hours, and the test has shown failure-free working for 10^3 hours, what prior belief would the observer have needed in order to conclude that the requirement had been met?

From above, (a, b) must satisfy

$$^1/_2 = \left(\frac{b + 10^3}{b + 10^3 + 10^6}\right)^a$$

which implies, since $b > 0$, that $a > 0.1003288$. It is instructive to examine what is implied by prior beliefs in this solution set. Consider, for example, $a = 0.11$, $b = 837.2$. Is this a 'reasonable' prior belief? We think not, since the *prior* probability that $T > 10^6$ is 0.458; in other words, the observer must believe *a priori* that there is almost a 50:50 chance of surviving for 10^6 hours. If $a = 0.50$, $b = 332333$, the prior $P(T > 10^6)$ is 0.499. As a increases this problem becomes worse.

In other words, to believe that 'this is a 10^6 system' after seeing only 10^3 hours of failure-free working, we must *initially* believe it was a 10^6 system. To end up with a very high confidence in a system, when we can see only a modest amount of testing, we must *bring to the problem* the relevant degree of belief.