CHAPTER 4

Qualitative Considerations on PFDavg and Bernoulli-pfd

I consider here in more detail than in Chapter 2 the probabilistic modelling of a software component of an on-demand safety function. Since the function is on-demand, it is appropriate to consider modelling as a Bernoulli Process.

A Bernoulli Process has a parameter, probability of failure on demand (pfd), which is constant over the service life of the product, no matter how many demands there are. The Process is said in statistical nomenclature to be "memoryless".

IEC 61508 defines a notion of average probability of failure on demand, PFD_{avg}. This notion relates to the unavailability of an item to perform its service, and is primarily oriented towards hardware, with fault detection, proof tests, repair times and so on all taken into account.

I consider here whether these notions are related in the case of software, and, if so, how.

4.1 Standard Definition of PFDavg

As noted in Chapter 2, Rausand [9] considers probability of failure on demand in Section 7.5.1 and average frequency of a dangerous failure per hour in Section 7.5.2. The book specifically "*focuses on IEC 61508 and ..... IEC 61511*." His concepts are meant to be the same as those defined in IEC 61508-4:2010 Clauses 3.6.17-19. Rausand defines:

> The average probability of (dangerous) failure on demand, PFD_{avg}, is

> the average probability that the item (SIS, subsystem, voted group, or channel) is not able to perform its specified safety function if a demand should occur. [9, Section 7.5.1, p182]

("SIS" is shorthand for "safety instrumented system", a concept from IEC 61511 [2]). He says further it is "*the same as the average unavailability of the item (see Appendix A) and is equal to the long-term average proportion of time where the item is not able to perform its safety function*." [9, Section 7.5.1, p182] He defines "average unavailability" in Appendix A.5.1. Where $X(t)$ is a Boolean variable indicating whether the item is available at t ($X(t) = 1$) or not ($X(t) = 0$), he defines the availability of the item $A(t)$ in Equation (A.44) as

$$A(t) = Prob(X(t) = 1) = Pr(\text{The item is able to function at time } t)$$

and its unavailability in Equation (A.45) as ...

$$1 - A(t) = Pr(\text{The item is not able to function at time } t)$$

(The term "Pr" here means "probability"). For comparison, consider the definition in IEC 60050 Part 192 [9, 192-08-04]:

> 192-08-04
> Symbol $U(t)$
>
> instantaneous unavailability
>
> probability that an item is not in a state to perform as required at a given instant

Thus Rausand's "unavailability" is the same as IEC 60050's "instantaneous unavailability".

The "*average unavailability*" is defined by Rausand in Equation (A.46) as the integral of the unavailability over a time interval, divided by the length of the time interval. Let the interval be $(T, T + \tau)$. Then the average unavailability is

$$\frac{1}{\tau}.\int_{T}^{T+\tau}(1 - A(t))dt$$

In IEC 61703:2016 [IEC16.1] and ISO 12489:2013 [ISO13], the definition of mean availability is given as an integral:

$$\overline{A}(t_1, t_2) = \frac{1}{(t_2 - t_1)}\int_{t_1}^{t_2} A(t)dt$$

respectively,

$$\overline{A}(t_1, t_2) = \frac{1}{(t_2 - t_1)} \int_{t_1}^{t_2} A(\tau) d\tau$$

which is the same availability formula with T replaced by t_1 and $T + \tau$ by t_2. Unavailability is the complement function as above, and so mean unavailability is

$$\overline{U}(t_1, t_2) = \frac{1}{(t2 - t1)} \int_{t_1}^{t_2} U(t) dt = \frac{1}{(t_2 - t_1)} \int_{t_1}^{t_2} (1 - A(t)) dt$$

as above, again with the change in notation for the time boundary points. This notion is called "mean unavailability" , alternatively "average unavailability", in IEC 60050 [9, Definition 192-08-06], with the same typological conventions as in IEC 61703.

Notice that the meaning of this expression is quite complicated. The average availability of an item between t_1 and t_2 is not the proportion of the actual time it was actually available over the interval (t_1, t_2), which is

$$\frac{1}{(t_2 - t_1)} \int_{t_1}^{t_2} X(t) dt$$

but the integral of the *probability* that it was available at a given time. So how do you determine that probability? You could say: we have a Weibull distribution for hardware failures of this particular item, say $WeibullItem(t)$. That's what gives us the initial probability of the availability of the item over the interval (t_1, t_2). And then, after a failure, we have to factor in

- detection time,
- repair time,

and, if the hardwareof the item is replaced, start the Weibull distribution anew from the end of the repair time. If this sounds complicated, that is because it is. And that is just the pure hardware failures. What about the software? When a demand is not being handled, the computation defined by the software is not being executed. Is the software available? Not necessarily; for example,

- it may be in the middle of a computation and the item design is such that cannot spawn a new computation to handle a new demand if one arrives; or
- it may be hung and the hang has not yet been detected - that is, it is not in a start state ready to perform a computation.

So the software may have availability/unavailability properties also.

4.2 What Are E/E/PE Safety Functions Composed Of?

Rausand considers safety-instrumented functions, SIFs, which are implemented by safety-instrumented systems, SISs. These are concepts which appear in IEC 61511:2016 [2]. SIFs are safety functions, as they are called in IEC 61508-2010 (which I shall also call SFs). Rausand's book contains a whole chapter, Chapter 8, 82pp long, on PFD_{avg}. He speaks in Chapter 8 primarily of "low-demand mode". This is a concept from IEC 61508:2010 [1] and is a demand-driven mode which may appropriately be modelled by Bernoulli Processes.

These functions are normally thought of as being performed; performance requires HW of some sort. When software executes, it executes on hardware. Software can be considered for various purposes largely independent of the hardware on which it runs, but the notions of availability and unavailability of safety functions require us to consider the software which implements those safety functions along with the hardware on which it runs. So what is software exactly? How do we characterise it as part of the item performing a safety function (or part of a safety function)? I suggest that we can consider software in this case as a design for a computation. This notion applies to source code in a high-level language, as well as object code loaded into and ready to execute on a processor. The software is available if the item is ready to start a computation according to that design, and it is unavailable if the item is not ready to start such a computation.

Such a design might have errors in it, indeed it often does. An error in the design means that when the computation is performed according to the design (when the software is run), the result is not what was expected or required. We can say the computation has failed (failure is an event), and the fault, the cause of this failure, is an error in the design (a software error). When I speak here of a software failure, I mean a computation failure caused by a software error. We can equivalently speak of a software error manifesting in the computation. Note there are also failures of the computation on non-faulty hardware which are not due to errors in the design, that is, not due to software error. For example, a computation begins in a start state, and it may be that the hardware can only conduct the computation beginning from some other state than a start state (e.g., on a serial processor, the program counter may not be pointing at a starting command of the computation, or the computational variables may not be initialised to acceptable starting values, when

the computation is invoked). If a computation starts from an acceptable start state, I call it an initialised computation, and if a computation starts from a non-start state, I call it an uninitialised computation.

When we are speaking of executing software, we are speaking of performing a computation according to the design given explicitly in the software. This is so whether we are speaking of software programs written in high-level language (so-called source code), or object code (compiled and linked code loaded onto a processor, ready for execution). Such an execution may fail, or it may succeed. Success means that the execution and its result fulfil the requirements on that computation, and failure correspondingly means that the execution and its result do not fulfil those requirements. (The requirements might be - usually are - written in a requirements specification, but it may also be that the specification of requirements does not include the environment under which the current execution is performing, in which case "fulfill[ing] the requirements" means rather fulfilling the expectations of the function in that case.) Modelling the software using a Bernoulli Process means that one considers a computation as independently as possible of the hardware which is performing the computation. A Bernoulli trial consists of running the software (from an acceptable start state) on its inputs, until the computation terminates. The result of the computation may be what is required/expected, in which case we designate the trial a "success". Or things may not work out to success:

- the computation may terminate without success because the hardware is faulty;
- the computation may terminate without success because of a software error;
- the computation may terminate without success because it was uninitialised;
- the computation may terminate without success because of an unanticipated interaction between the hardware and the software;
- the computation may not terminate, because of
  - faulty hardware, or
  - faulty software design, or
  - an uninitialised start, or
  - an unanticipated hardware-software interaction
  - and so on....
- and so on....

In order to model the computation as a Bernoulli process, in order to detect software error, it is necessary to select from all demand computations those which represent Bernoulli trials and leave out those which do not represent Bernoulli trials. Some failed demand computations will be such that there is some failure of a hardware function, with or without the involvement of a software error. These are candidates for not being Bernoulli trials. However, a software error could manifest itself in such a trial, but not be noticed because of the hardware failure. In this case we say that the software error is *masked*. There was indeed a Bernoulli trial of the software, even though the hardware failed, and it was a failed trial because a software error manifested. We might well have missed that failed trial because we were concentrating on the hardware. It is a condition of successful evaluation using Bernoulli-Process modelling that computational errors due to software errors are not masked and overlooked in this way.

Bernoulli trials must also be statistically independent of one another (this is a prerequisite for the application of the mathematics of Bernoulli Processes). It is generally understood that uninitialised demand computations do not constitute Bernoulli trials; that only initialised computations are appropriately considered as Bernoulli trials [6].

We need to consider the characteristics of failed computations taken as software, running on hardware, in order to determine any relation between the software execution considered as a Bernoulli Process and the meaning of the parameters of this Process such as pfd (which I shall also call $Bernoulli - pfd$), and the computations taken as involving hardware running according to the computational design given by the software, which has parameters such as average probability of failure on demand, PFD_{avg}.

4.3 Characterisation of Situations of Unavailability

Let us first consider components or items implementing part or all of a safety function: hardware which may or may not perform according to computational designs given by software. The considerations on availability or unavailability of SFs are as follows. Say there is an SF which is available, but not in use, at time t. We can say its status is $AVAIL$ at t, $AVAIL(t)$. We recall that Rausand uses the formulation $X(t) = 1$. Then the SF might become unavailable at time t_1; $UNAVAIL$ at t_1, $UNAVAIL(t_1)$,

$\neg AVAIL(t_1)$, $X(t_1) = 0$.

Has the SF dangerously failed if it is unavailable? No, not necessarily. The SF may be in a necessary recovery period after an invocation, and the system environment and overall design may be such that there can be no demands on it during this recovery period. But suppose that the unavailability indeed constitutes a dangerous failure. That dangerous failure DF might be detected, DDF, or it might be undetected, DUF.

Critical items which react on demand are usually subject to a proof test. A proof test is a test performed at regular intervals to ensure that the item is in working order: in this case, that would mean

- non-faulty hardware
- in a start state
- waiting to execute its critical function on demand.

Proof tests take place at regular intervals of time specified in the overall system design. There are three situations which can occur in a specified interval Int between proof tests. Proof tests take place after a specified regular time period, whose length I denote by τ. For the purpose of clarifying the three situations, I ignore here the possibility of multiple dangerous failures of the same item in the interval Int. The clarification follows that of Rausand [9, Section 8.1.1], with some additions. For simplicity, let Int begin at $t = 0$. The three situations are:

- No DF occurs in Int. Then the proof test occurs at time τ. The proof test takes place in interval PT, taking time PTI, which may be "rather short": in process-industry terms that might mean 1-2 hours, but can in some cases be significantly longer. The SF is $AVAIL$ throughout Int. We may consider whether it is $AVAIL$ or $UNAVAIL$ during the interval PT. Thus the entire interval cycle consists of Int followed by the proof test, $Int \frown PT$, and is of length $(\tau + PTI)$. The function is $AVAIL$ for time τ at least, and may be $AVAIL$ for $(\tau + PTI)$, depending on the design. This situation is depicted in Figure 4.1.

- A single DDF and no DUF occurs in Int. When the DF occurs, there is an interval until it is detected. The detection interval DI of length D is a fraction of the time τ_D between two diagnostic tests (diagnostic tests are the execution of the detection mechanism, not to be confused with the proof test of the item). Then follows a repair interval RI, of length R. Over the course of Int, the SF is

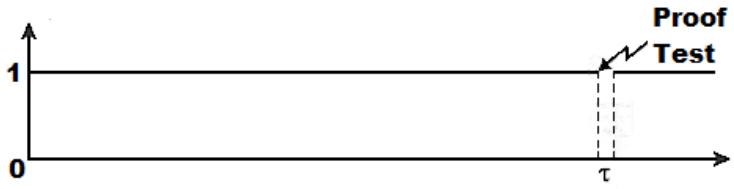

Figure 4.1: Proof Test (after Rausand)

$AVAIL$ for time $(\tau - D - R)$ and is $UNAVAIL$ for time $(D + R)$. The maximal size of D is τ_D and the mean length of RI is known as the *mean time to repair*, $MTTR$, a parameter generally expected to be known for the item. On the basis of the known quantities τ_D and $MTTR$, we could simplify the $AVAIL$ time calculation to $(\tau - \tau_D - MTTR)$. This situation is depicted in Figure 4.2.

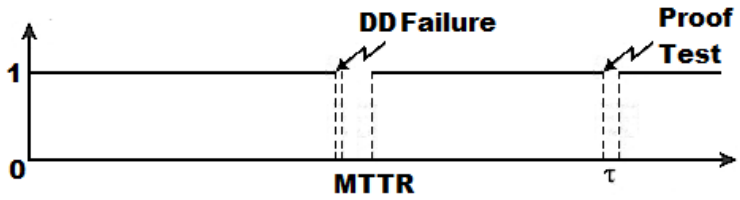

Figure 4.2: Proof Test (after Rausand)

- A DUF occurs in Int, say at time T_0. This remains undetected until the proof test starts, so the SF is $UNAVAIL$ during Int for time $(D1 = \tau - T_0)$. Then the proof test occurs in a time interval PT of length PTI and the fault is detected. The fault is then repaired. SF is $AVAIL$ for T_0 and $UNAVAIL$ for time $(\tau - T_0)$ over Int. The combination of proof test with detection, followed by the time for repair, expressed in known quantities, can be taken to be $(PTI + MTTR)$. This situation is depicted in Figure 4.3.

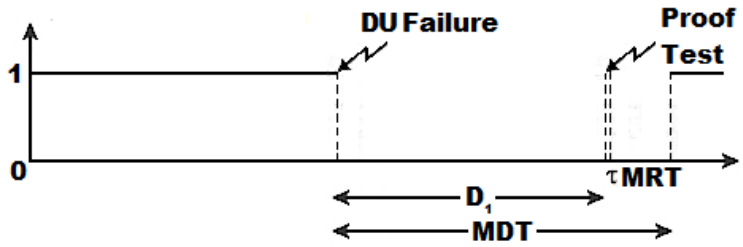

Figure 4.3: Proof Test (after Rausand)

4.4 AVAILability and Failure of Software

Let us call the condition of an error manifesting on execution starting from an acceptable start state a *live-fail*. Live-fails can occur because a software error manifests, or because a hardware failure occurs during execution of the SF, or both. Let us denote by *live-fail(SW)* a live-fail which is due to manifestation of a software error. (Note that live-fail(SW) includes possibly-masked manifestations!) The mathematics of Bernoulli Processes applied to software error manifestations requires that $Pr(live - fail(SW))$, the probability of a live-fail(SW) on demand, is constant, that is, that the process is memoryless, and some other minor conditions. Since, during a live-fail, the execution started from an acceptable start state, the SF must have been $AVAIL$ at the start of the execution. Whether the SF remains $AVAIL$ during its execution is largely a matter of requirement and system design. If there is a need for the SF to remain $AVAIL$ in case another demand occurs during the response to the first demand, then multiple instances of the SF may be required. If there can be shown to be no further demand on the SF that can occur during execution, for example, for a software-based SCRAM in an nuclear power plant, then there is no need for multiple instances.

A computation running on a processor exhibits a mixture of

- $UNAVAIL$,
- correct execution, and
- live-fail.

Correct execution corresponds to a Bernoulli trial for the software which is successful. A live-fail(SW) corresponds to a Bernoulli trial for the software which is unsuccessful.

What about $UNAVAIL$ conditions for software? Can or could they be involved in Bernoulli trials in a Bernoulli Process?

Let us suppose that there is one instance of a computation $AVAIL$able at any one time. When a demand occurs and the computation starts, then we may consider that it is not in a position to accept a further demand until it has terminated and reset to a start state. This is one way in which the software can become $UNAVAIL$, namely that a computation is in course.

Classic run-time-errors in software are

- overflow,
- underflow,
- divide-by-zero,
- call-into-void.

They are certainly dangerous failures in that they inhibit the computation. However, they are detected because each run-time error raises an exception condition, an interrupt flag, in the processor which is read before the next processor cycle. They fit the definition of DDF, and the progression of a DDF as above, but there is a question whether one would want to consider checking interrupt flags a "proof test" (usually not) or trapping to exception handling a "time to repair" (also usually not) when these occur at a rate of GHz. Furthermore,

- handling such exceptions is part of software design. When they are appropriately handled, the computation may still succeed. When handled inappropriately, it may be considered a live-fail(SW);
- commercial software development tools are available which demonstrably eliminate the possibility of run-time errors occurring. Some of them work very well indeed.

Hence it is likely inappropriate to consider run-time error occurrence in software to render the SW $UNAVAIL$; more appropriate to consider a run-time error a failed Bernoulli trial.

Some other error behaviour might appropriately be considered DDF if the software and processor are instrumented: deadlock, for example. Again, such a detected failure could be considered a live-fail(SW) and thus a Bernoulli trial. An undetected deadlock would lead to $UNAVAIL(SW)$. Should this happen, it could rightly be considered poor software design and verification. Tasks meant to progress can be

assured progression by various means, including internal tests for deadlock followed by resolution mechanisms, and these can be incorporated during development.

Some error behaviour might be DUF unless there is some fairly subtle instrumentation: for example, various cases of livelock, or operating-system priority-inversion phenomena (as happened to the Mars Pathfinder probe [8]). These are characterised by states of the software (the states may cycle, as in livelock examples, but once you are in any of the states in the cycle you have entered the cycle and remain in it). If there is livelock, then a computation will not terminate unless and until the livelock is detected and resolved. The software is $UNAVAIL$ between the commencement of the livelock through its detection and resolution. Whether this is an $UNAVAIL$ condition or an example of part of the computation during a Bernoulli trial depends on the software architecture and the software requirements.

If the SF software enters any of these error conditions, and the condition is not resolved, it is not in a position to complete execution of the safety function successfully, nor is it in a position to respond to further demands (to start a computation corresponding to a new demand). Best said, it is $UNAVAIL$. Just as with HW, you need to reset it (to some start state) in order to make it $AVAIL$ again. That is what constitutes the repair, in this case. RI is typically very, very short (not longer than the "short" 1-2 hours mentioned earlier). Similar to HW, when a software-implemented SF is $UNAVAIL$, it stays $UNAVAI$L until the condition is detected, which means either by diagnostics or by proof test, and then it is repaired (again, typically meaning reset/reboot). However, as I have noted above, much of the diagnostics and repair of such conditions could be rendered internal to the software, as part of fault-tolerant design and programming.

Software becoming and remaining $UNAVAIL$ during an execution which does not successfully terminate can reasonably be regarded as a fail-live(SW) condition and therefore as an unsuccessful Bernoulli trial. If software becomes $UNAVAIL$ due to exhibiting one serial logical computation instance and being in process of computing, then either an analysis of demand should have shown that no demands will occur then, or multiple instances of the computation must be made available by design. It is hard to think of instances of software becoming $UNAVAIL$ which are not situations which should be rectified during system development

4.5 Software FoD Contributes Nothing to PFDavg

Whether or not software may become UNAVAIL, software failure on demand occurs when $((HW \text{ is } AVAIL \wedge SF \text{ is } UNAVAIL) \vee \text{live}-\text{fail(SW)})$. The condition $(HW \text{ is } AVAIL \wedge SF \text{ is } UNAVAIL)$ is questionable, as just explained. Live-fails are the failures primarily addressed in Bernoulli-Process modelling.

A PFD_{avg} can be calculated in the ways specified by IEC 61508 also for SW, and it will generally not be the same as $Bernoulli - pfd = Pr(\text{live}-\text{fail(SW)})$, which is constant. Consider the following (where $AVAIL/UNAVAIL$ here concern the software alone).

$$FD(t) = DEMAND(t) \wedge (UNAVAIL(t) \vee (AVAIL(t) \wedge \text{live}-\text{fail(SW)}(t)))$$

This predicate expresses all ways in which a computation fails to fulfil its requirement when a demand is placed at time t. When there is a demand at t, we can identify the probability of live$-$fail(SW)(t) with the Bernoulli-pfd, which is by hypothesis constant.

The average of this will be the integral over the appropriate interval:

$$FDavg = \frac{1}{\tau}\int_{T}^{T+\tau} (UNAVAIL(t) \vee (AVAIL(t) \wedge DEMAND(t) \wedge \text{live}-\text{fail(SW)}(t)))dt$$

$=$

$$\frac{1}{\tau}.\int_{T}^{T+\tau} UNAVAIL(t)dt$$

$$+\frac{1}{\tau}.\int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t) \wedge \text{live}-\text{fail(SW)}(t))dt$$

since $UNAVAIL(t)$ and $AVAIL(t)$ are mutually exclusive. Consider now the second term of the sum

$$\frac{1}{\tau}.\int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t) \wedge \text{live}-\text{fail(SW)}(t))dt$$

Since live$-$fail(SW)(t) is a Boolean over this interval, it follows that

$$\int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t) \wedge \text{live}-\text{fail(SW)}(t))dt$$

$$\leq \int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t))dt$$

Also,

$$\int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t))dt \leq \int_{T}^{T+\tau} AVAIL(t)dt \times \int_{T}^{T+\tau} DEMAND(t)dt$$

But $\int_{T}^{T+\tau} DEMAND(x)dx = 0$, for a set of demands plausibly occurs in $[T, T+\tau)$ only on a set of measure zero (actually, plausibly only a finite set of points t; see below. This is obviously true of any time interval, not just $[T, T+\tau)$). It follows that

$$\int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t))dt = 0$$

and thus that

$$\int_{T}^{T+\tau} (AVAIL(t) \wedge DEMAND(t) \wedge \text{live}-\text{fail(SW)}(t))dt = 0$$

and thus that

$$FDavg$$
$$= \frac{1}{\tau}. \int_{T}^{T+\tau} (UNAVAIL(t) \vee (AVAIL(t) \wedge DEMAND(t) \wedge \text{live}-\text{fail(SW)}(t)))dt$$
$$= \frac{1}{\tau}. \int_{T}^{T+\tau} UNAVAIL(t)dt$$

So the average failure on demand of on-demand software over an internal is just the period of time for which it is $UNAVAIL$. I have argued that most prima facie instances of $UNAVAIL$ can and/or should be accommodated within software design itself, so that

- persistent instances of unavailability in a computation are not an actual instance of $UNAVAIL$ and can be considered rather a live-fail(SW) as part of a Bernoulli trial; and
- when software is or remains $UNAVAIL$, it is not participating or able to participate in a Bernoulli trial.

This second point, along with the observation that the contribution of software to unavailability is the length of time for which it is $UNAVAIL$, shows that the Bernoulli-Process-type failures of the software, live-fail(SW), contribute nothing, zero, to FD_{avg}. Since live-fail(SW) events contribute zero to FD_{avg}, they can equally contribute nothing to the mean of this quantity, PFD_{avg}. I have thereby rigorously shown what was already largely observed in Chapter 2, namely that the parameters of the Bernoulli Process are unrelated to PFD_{avg}.

4.6 Formulas for UNAVAIL of Software

We considered above that the software can be unavailable, just as the hardware can be unavailable, and this software-unavailability takes up time; it is not a point process such as the Bernoulli-trial failures. Software unavailability must be detected and rectified. We have noted that certain SW development methods eliminate the possibility of run-time-error; that certain static-analysis techniques can often eliminate the possibility of deadlock or livelock; run-time verification can help eliminate the remaining possibilities of livelock, priority-inversion and so on. It is often possible to (correctly) verify your SW free of violations of safety and liveness properties. If all of these measures are successfully undertaken, the software is never $UNAVAIL$.

Nevertheless, the intellectual effort required to eliminate all possibility of $UNAVAIL$ might be too high. It might be reasonable to accept a small possibility of software $UNAVAI$L, in which case means of calculation are needed to demonstrate that this possibility is acceptable (IEC 61508 is predicated on the basis of acceptable risk). This can generally only be done thoroughly with time as a parameter when the stochastic process of the occurrence of demands in time is known and considered [7]. I consider here purely combinatorial formulas.

When there is software which is part of a safety function and this functions on demand, it is normally the case that, between demands, the software is in some kind of a “start” state, a state from which it can execute its function according to its specification. After a demand has completed, the SW may well be in a “finish” state which is not a “start” state. Such software will have to “reboot” or be “reset”, that is, transition to a “start” state, whence it is ready to service a new demand. The “reset” will take a non-zero time interval of length, say, T_{reset} (and let us suppose here that T_{reset} is constant). If any further demands on it occur in this time interval when the

software is resetting, the software instance may be considered to be $UNAVAIL$ and we may presume the demand on it will fail.

Observe that in this case the memoryless-ness of the Bernoulli Process is violated. Let the Bernoulli-pfd be $p \neq 1$. Say a demand at t completes at t_1. Then in the time interval (t, t_1) we may presume the software is unavailable because it is executing, and in the time $(t_1, t_1 + T_{reset})$ it is unavailable because it is resetting. Hence in the interval $(t, t_1 + T_{reset})$ the probability of failure on demand is 1, which by hypothesis is not equal to p.

This observation is also valid for the case in which there is a run-time failure which is not recovered, or a deadlock or livelock which is not recovered. In these situations, the defined safety-function functionality of the software remains unavailable until the software is successfully recovered to a start state. In the unrecovered condition, the probability of failure on demand is 1, which is not equal to p.

It follows that, under the conditions of unavailability we have considered, the Bernoulli-process assumptions are violated for the period of time during which the software remains unrecovered. This justifies the consideration of Bernoulli trials only during the period in which the software is $AVAIL$. (I have also noted that the boundary between Bernoulli trials and $UNAVAIL$ of software is flexible, and that design plays a big part in this.)

It follows from this that the software works as two disjoint Bernoulli processes; one which fails with Bernoulli pfd $p \neq 1$ (outside any interval $(t, t_1 + T_{reset})$ where a demand at t completes at t_1), and one which fails with pfd 1 (inside any interval $(t, t_1 + T_{reset})$ where a demand at t completes at t_1). I shall let the reader choose the convention as to what happens with a demand exactly at $t_1 + T_{reset}$.

There are two formulas for the unavailability of the SW in the interval (T_1, T_2) and three cases:

If no demand happens in $[T_2 - T_{reset}, T_2]$,

$$UNAVAIL = \sum\nolimits_{AllDemands}(t, t_1 + T_{reset}))$$

If a demand happens at t_1 where $T_2 - T_{reset} < t_1 \leq T_2$ and completes at $t_2 \leq T_2$,

$$UNAVAIL = \sum\nolimits_{(AllDemandsExceptTheLast)}(t, t_1 + T_{reset})) + (T_2 - t_1)$$

If a demand happens at t_1 where $T_2 - T_{reset} < t_1 \leq T_2$ and does not complete by

T_2,

$$UNAVAIL = \sum_{(AllDemandsExceptTheLast)}(t, t_1 + T_{reset})) + (T_2 - t_1)$$

If we consider execution time to be constant T_{exec}, these simplify to
If no demand happens in $[T_2 - T_{reset}, T_2]$,

$$UNAVAIL = (\text{no.}-\text{of}-\text{demands}).(T_{exec} + T_{reset})$$

If a demand happens at t_1 where $T_2 - T_{reset} < t_1 \leq T_2$ and completes at $t_2 \leq T_2$,

$$UNAVAIL = (\text{no.}-\text{of}-\text{demands } -1).(T_{exec} + T_{reset}) + (T_2 - t_1)$$

If a demand happens at t_1 where $T_2 \breve{} T_{reset} < t_1 \leq T_2$ and does not complete by T_2,

$$UNAVAIL = (\text{no.}-\text{of}-\text{demands } -1).(T_{exec} + T_{reset}) + (T_2 - t_1)$$

The Bernoulli-pfd, which has solely to do with whether the SW executes according to specification and thus whether or not there are systematic failures, is thus decoupled from the $UNAVAIL$ability of the SW, which has solely to do with the time-in-execution and the time-to-reset, given by these formulas.

4.7 Summary

I have considered the notions of PFD_{avg} and of Bernoulli-pfd. I have argued that failures of Bernoulli trials of software executing on hardware do not contribute to calculations of FD_{avg} and therefore not to its mean, PFD_{avg}. However, just like hardware, software may be unavailable when a demand occurs, for a variety of reasons, some good, some not so good. During such times, the probability of failure on a demand is 1. It thus makes sense to consider software residing on and occasionally executing an on-demand safety function (or part) on a processor to be logically a disjoint sum of two Bernoulli processes:

- one (in which the software is ready to execute its safety function) with a pfd of p $\neq$ 1, and
- one (in which the software is UNAVAIL) with a pfd of 1.

Bibliography

[1] International Electrotechnical Commission, IEC 60050, *International Electrotechnical Vocabulary*, on-line at www.electropedia.org . Part 192 is at http://www.electropedia.org/iev/iev.nsf/index?openform&part=192

[2] International Electrotechnical Commission, IEC 61508:2010, Functional safety of electric/electronic/programmable electronic safety-related systems. 7 parts, IEC, 2010.

[3] International Electotechnical Commission, IEC 61703, Mathematical expressions for reliability, availability, maintainability and maintenance support terms. IEC, 2016.

[4] International Electrotechnical Commission, IEC 61511:2016, Functional safety - safety instrumented systems for the process industry sector. 3 parts, IEC, 2016.

[5] International Standards Organisation, ISO 12489:2013, Petroleum, petrochemical and natural gas industries – Reliability modelling and calculation of safety systems, ISO, 2013.

[6] Bev Littlewood, personal communication (many times, 2010-2017).

[7] Bev Littlewood, Note on a Bernoulli process embedded in a Poisson process, Document DKE 914.0.9_2016-0018, DKE Working Group on Statistical Evaluation of Critical Software, August 2016.

[8] Rapita Systems Ltd, *What really happened to the software on the Mars Pathfinder Spacecraft?* blog post, no date. Available at https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

[9] Marvin Rausand, Reliability of Safety-Critical Systems, John Wiley and Sons, 2014.