Software as strong as a dyke

C. Verhoef
Free University of Amsterdam, Amsterdam, The Netherlands,
x@cs.vu.nl

Abstract

The Dutch storm surge barrier's failure rates are designed and certified to fail maximally once every ten thousand years. Recent audits question this failure rate. Software is mentioned as a major cause and quantifying it and its reliability turned out to be a problem. We show how to quantify and assess the software and its failure rates using minimal data. The results are applicable to other dependable systems.

Keywords and Phrases: safety-critical systems, failure rate, residual errors, dependable systems

Introduction

We all remember the catastrophic flooding of New Orleans in 2005. In the Netherlands, in 1953 there was a large flood as well, and to prevent this from happening ever again, a long-term program called the Delta Project was embarked upon which was completed in 1998 with the Rotterdam storm surge barrier. This so-called Maeslantkering intends to protect 1.3 million people in the hinterland and one of the largest harbors in the world by closing the river temporarily. The Netherlands could suffer hundreds of billions of Euros in damage if Rotterdam's protection works were to fail. The overall reliability demand for the Delta Project is that flooding should not exceed a chance of once in ten thousand years.

The American Society of Civil engineers chose the Netherlands North Sea Protection Works as one of the seven wonders of the modern world to pay tribute to the greatest civil engineering achievements of the 20th century. So it's not a surprise that a delegation of fifty Louisiana officials from national, state and local levels visited the Dutch water protection works, including the Maeslantkering. This movable storm surge barrier is closed during extreme storm conditions. The closure operation is initialized and carried out by a software system. The system is triggered when both the predicted storm surge level and the river discharge are expected to exceed a certain level. After this stage, the barrier gates are submerged to a level where forces of river discharge and pressure from the sea are in balance. The "residual" net forces acting on the gates are diverted via two steel constructions larger and stronger than Eiffel towers to ball-and-socket-joints. Each ball-diameter is ten meter and cast in 52000 tons of concrete. These gigantic hip-constructions reside on the riverbanks. See Figure 1 for an aerial view of the Maeslantkering.

Figure 1: The Maeslantkering during a test closure. The twin-rotating gates are closed and ready to submerge. To give an idea of dimensions, each steel door is 22 meters high and 210 meters long. The tiny spots on the roads are automobiles.

Reliability

The software running the Maeslantkering is essential, and is supposed to be as strong as the Dutch dykes: it may fail only once every ten thousand years. It turned out that human control of the storm surge barrier displayed a failure rate in the order of one in thousand for the complex task of decision-, closure- and opening-management. This violated the reliability requirements of the Dutch government. Therefore, computers plus software were chosen to completely autonomously control the barrier. Apparently, important decision makers were convinced that this was feasible. The failure rate demands for this system were required to be $1 : 10000$ for not closing when this was actually necessary, and $1 : 100000$ for not opening the barrier when requested. This asymmetry is caused by the fact that if the sea surge barrier does not reopen, the river discharge can lead to flooding from the inside.

This safety-critical software system is delivered and is running since 1998. Its design, development and construction is in accordance with the IEC 16508 standard describing functional safety of electrical, electronic, or programmable electronic safety-related systems. This standard prescribes for software development a set of best practices. The IEC warns that many factors affect software safety integrity, so it is not possible to combine the best practices to guarantee success in any given application. The recommended software techniques must be chosen with care. For instance, personal competences, experience with certain techniques, familiarity with the domain, size and complexity, industry sector recommendations and recognized best practices plus other standards all play a role. Still a certified safety integrity level for IEC 61508 compliant code is sometimes interpreted as the failure rate of the software. Of course, when you adhere to a set of best practices, the intention is to minimize error, and therefore, failure. But as the IEC warns: compliance with standards does not imply a guarantee on quantitatively defined failure rates.

The software of the Maeslantkering is certified at the highest safety integrity level (SIL 4) by the International Atomic Energy Agency. SIL 4 implies a failure rate of one in ten thousand per demand or once per 100 million hours (in fact SIL levels provide a bandwidth of one order of magnitude around certain failure rates). Indeed, certification confirms that the software is delivered in accordance with the guidelines of IEC 61508, but not that the accompanying failure rates are achieved. We are not aware of scientific evidence that ties the SIL 4 failure rate to these best practices. This is also recognized by the people involved in building the Maeslantkering software given their comment [12]:

> Some people, though not many and sometimes only for publicity reasons, claim that formal methods can guarantee correct software and that no other method can. It will hardly need argumentation to refute this claim: there is not a single method which can achieve perfection. Apart from simply being not true, it is a dangerous claim, because it sets high expectations on formal methods and it presupposes an all-or-nothing attitude towards formal methods.

We entirely agree with this statement, and in fact, there are two major problems with the current practice to allocate SIL levels to software as noted by Bishop [3].

- A safety integrity level is actually associated with a safety function, in our case the safety function to open or close the barrier. The mapping from the SIL label for the function to requirements for the subsystems and associated software does not explicitly recognize the contribution of the system architecture. For example, there is no credit for implementing diverse means to activate the barrier—the software in each subsystem still has to meet the requirements associated with the top-level SIL label. Similarly, no account is taken of the fact that some software components are less critical to safety than others. This has been partly resolved in the draft revision of the IEC61508 standard, where rules are defined for relaxing the SIL requirements for software.

- There is no technical basis for the linkage between recommended software techniques and a target software failure rate implied by a SIL label. Any linkage can at best be shown on average, and it is not certain that a specific development process will achieve a given dangerous failure rate. This leads to an abuse of the standard where compliance with SIL-mandated techniques is deemed sufficient to claim that the dangerous failure rate is in the stated SIL band.

The increasing dependability of our society on safety-critical software-intensive systems justifies that we rethink the SIL-idea for software so that measurable software development techniques can be correlated with failure rates. Furthermore, given the criticality of the Maeslantkering, we will use all the data that is available to us to assess whether its failure rate is within the SIL 4 band. We hope to provide useful input to the maelstrom of failure estimates that politicians, journalists, and others poured out in the Dutch media. These range from alarming failure rates in the order of one in ten to reassuring rates of one in many thousands. The software seems one of the causes for this volatility, as an external assessment put it: software has a major contribution to the entire barrier-process. However, the numerical contributions are based on engineering judgement to which no absolute value judgement can be given.

The available data

The Dutch government is very reluctant with disclosure of data concerning potential problems with the Maeslantkering, moreover, the software organizations involved are not allowed to communicate with others. Therefore, it is difficult to obtain data, and we need creativity to "read the IT-leaves". Fortunately, we recovered from various sources the following data points for the Maeslantkering software. The system contains 450000 lines of C++ code of which 200KLOC is operational code, and 250KLOC for simulating, testing and supporting the operational code [12]. It took about 25 person year, and the duration of the project was three year [6]. It was a fixed-price project and completed in October 1998. During development 1655 problems were reported, and 119 were found during customer acceptance testing. Of those 119 about 27% of the problems were found in critical modules, and 31% in core modules. So during acceptance testing the customer detected 32 problems in critical modules and 37 in core modules. While in operation three residual faults were found until October 2000. During development 85% of the problems were found, during reliability testing another 8%, and customer acceptance another 7% and in operation 0.18% [11, 12].

A millennium of experience

The Dutch have a millennium of experience in building dykes, yet thousand years of experience is no guarantee for absence of design flaws. The Dutch pittoresque village of Wilnis was surprised by a flood on August 26th 2003, when an already weakened dyke of a ring-canal failed due to lack of water instead of too much water. The warmest and driest summer in fifty years lowered the density of the upper part of the dyke culminating in horizontal shear failure [2]: 50 meters of dyke shifted into the village, creating two breaches through which canal-water entered (see Figure 2). The dyke strength assumed a surplus of water, giving the dyke its structural integrity. Centuries of experience could not prevent this flood, since floods are only rarely caused by a shortage of water [2].

The history of information technology spans only 50-odd years. We all know that there are many and major problems with software, and this seems also true for the barrier-software. Namely, a Dutch newspaper reported that since 2001 at least 11 million Euro was spent on improving the barrier-software and its decision process. It was not stated which percentage to which activity, but it is clear that the decision process is expressed in software. So despite the low number of reported faults since Oktober 2000, something must have been wrong after all.

Figure 2: An aerial view of the horizontal shear failure in a small village near Amsterdam, due to lack of water causing the dyke to shift by the pressure of the canal water. Notice the small blue spot in the canal which is a grounded yacht.

Potential failures and problems

Input validation of the Maeslantkering software seems not watertight, given the following testimonial from an insider. A maintenance engineer connected the software to a water sensor upstream where the Rhine enters the Netherlands. This level is uniformly too high, so the barrier started the closing procedure. If this is true, no intelligent input validation is being performed, since the combination of calm maintenance weather plus a virtual thunderous 4 meters above maximum discharge was taken to be valid input and acted upon. Another insider could not confirm this testimonial.

Another near-failure seems to have occurred at the Eastern Scheldt storm surge barrier. This barrier consists of a four-kilometer bridge that turns into a dyke by closing 62 steel doors each 42 meters wide, cast in 65 concrete pillars each the size of a 10-store building (see Figure 3 for an aerial view). This barrier has a maintenance mode, and after maintenance it must be reset to operational mode again. When one maintenance engineer forgot about that, the steel doors went further up when weather conditions made closure necessary. This software failure was solved by humans on-site. If this is really true, mode-monitoring was not implemented properly: the system should not tacitly assume a maintenance mode for more than a predefined time-frame.

external assessment Our inside information is not inconsistent with the findings of a confidential external assessment that is in our posession, where dominant contributions to failure probabilities are formulated for the entire Maeslantkering. The decision part of the software can fail due to acceptance of erroneous external data. The risks are aggravated by the fact that problem solution takes too much time. Moreover, a configuration management system is recommended that should monitor, control, and capture the status of the Maeslantkering components such as valves and switches. For instance the position of so-called needle-valves in the hydraulic system of the ball-joints, and the switches of the 10kV automatons should be saved and restored after maintenance.

The point we like to make here is that the approach taken towards operator-error should not be naive. It is known for a long time that 60–80% of major accidents with

Figure 3: An aerial view of the Eastern Scheldt storm surge barrier. The steel doors are open showing a stratified pattern in the water. Notice the ship in the sea, which is small compared to this barrier.

complex systems such as nuclear power plants, dams, tankers, and airplanes triggered by operators were not solely attributable to carelessness. Other important failure factors include flawed system design, poor training and poor quality control [10]. The rumors about the near-accidents with the Dutch storm surge barriers and the external audit rather indicate design flaws than carelessness by operators. So we agree with the external audit's recommendation to reduce operator-error by adapting the software in such a manner that carelessness cannot cause catastrophic failure.

In Figure 4 we provide an overview of the various failure probabilities for non-closure mentioned in the external assessment. It is interesting to notice that failure probabilities are estimated for the software parts. These probabilities were established via fault tree analysis. The estimated failure probability of all software-related issues totals to a failure rate of 0.02037, the relative contribution of software to the failure rate is 22.2%. This is the second largest contribution, the largest being problems with the ball-joint that takes 5 weeks of repair after each closure operation.

Seiches

In accordance with IEC 61508 some parts of the software were specified with formal methods. However neither formal specifications nor correctness proofs exclude requirements errors. Also in the case of the Maeslantkering, requirements errors cannot be excluded. We provide an example of a potential requirements problem.

This concerns the presence of co-called *seiches*: a standing wave in a body of water, due to wind, weather, or seismic activity. In the period 1995–2001 the harbor of Rotterdam encountered 51 seiche events with an amplitude between 0.25 and 1.69 meter. After the barrier became operational an audit revealed that the effect of seiches on the water level was not accounted for in the software.

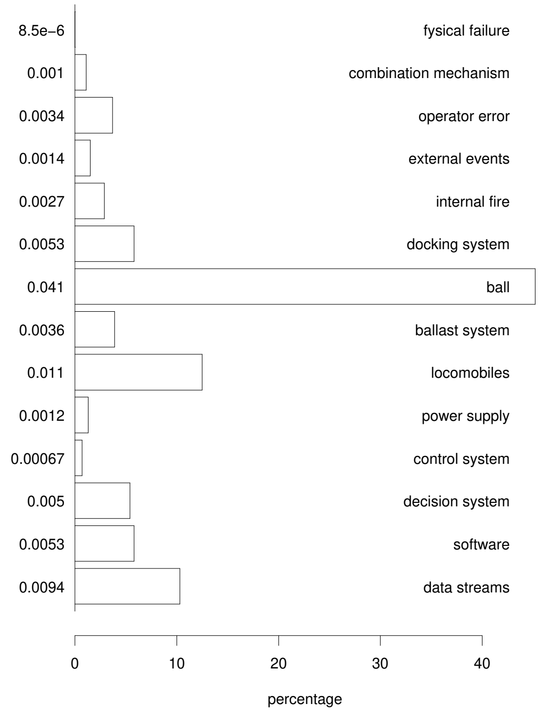

Figure 4: A bar plot with absolute failure estimates for nonclosure, plus their relative contribution as a percentage of the whole. The data is taken from a summary of an assessment report on the Maeslantkering.

In [9] it is shown that all 51 seiches coincided with the passage of a low-pressure weather system, and that when there was also a sharp cold-front, numerical simulations could reproduce the seiche events. In a 2004 PhD Thesis of one of the just cited authors we can read: ”Because of specific circumstances that can occur during the deployment of the barrier, the trough of a seiche in the Waterway Basin can cause a critical situation when the water level on the sea side of the barrier drops below the level on the river side. In extreme situations, this could cause the failure of the storm surge barrier since it is primarily designed for protection against high water levels on the sea side. If the net force directed towards the sea side of the barrier becomes too large, this could cause the ball-joints to be pushed out of their sockets, similar to the dislocation of a shoulder.” As with the dyke that failed due to lack of water, the barrier might fail due to low water levels while designed to protect against high levels.

Our conjecture is corroborated by Vrancken [13] who reported that: ”[t]he problem was detected already in the development phase in 1997, but its solution caused one year of delay in the delivery of the barrier.” So, seiches were taken into account but later and apparently ad hoc: ”the water level is monitored on both sides of the barrier and a system of pumps and valves ensures that the barrier floats to the surface in case the water level on the sea side drops below the level on the river side. This approach is expected to avoid damage to the barrier. However, an actual seiche-prediction system is not available for the closure-management of the Rotterdam storm surge barrier" writes de Jong in his PhD Thesis [8]—he developed an award-winning method for the prediction of the occurrence of seiche episodes.

Best-in-class comparisons

Using benchmark information we can create a more quantitative view on the software and its potential faults. Particularly insightful in this realm are the benchmarks by Capers Jones [7]. Namely, Jones not only provides industry averages but also extreme values. So, best-in-class results can be compared to the data of the Maeslantkering software, which is considered best-in-class, too. The category of software that is most close to Jones' industry partition is either systems software or military software. We will provide relevant extreme values of his benchmarks for both types of software and compare them to data for the Maeslantkering software.

We convert the 450000 Lines of C++ into function points via backfiring: on average it takes 53 lines of C++ to create one function point of software. This yields 8490 function points. We assume that real object-orientation is used, since the C++ was converted by hand from formal Z++ specifications (otherwise a factor of 128 for plain C would have been more appropriate). This tells us that the software is in the 10K function point range. The industry benchmarks stem from 1995–1999, the same period as the Maeslantkering software. The systems software benchmark contains 345 new and 575 enhancement projects; for the military benchmark there are 130 and 135 respectively.

assignment scope The highest assignment scope for systems software development measured by Jones is 315. Note that 25 person year in three years, is on average 8+ persons per year. We positively estimate that the system was made with 12 staff at most, so their assignment scope is at most $8490/12 = 707$. In other words, the amount of function points being dealt with per person is more than twice as high as the maximal value in a benchmark of 345 systems software projects. For military software the highest assignment scope measured is 290, which leads to an even larger deviation.

defect removal efficiency The best-in-class defect removal efficiency for systems software in the 10K function point range is 98% (average: 92%). The best-in-class number of delivered defects is 800 of which 96 high-severity defects (averages: 4400 and 660 respectively). Since this is 2% of the defects, there must have been 40K defects in the software before delivery (average 55K). Public records testified that 1655 problems were reported in the Maeslantkering software [11]. So they reported a factor 24 less defects than best-in-class systems software. Likewise for military software the highest measured defect removal efficiency is 99%, delivering 400 defects, and 48 high-severity ones. Also in this case a factor 24 less errors were reported for the Maeslantkering software.

staff size Best-in-class staff size of systems software in the 10K function point range is 45 (average staff size is 67). The Maeslantkering software used a team of at most 12, which is a factor 3.8 less than the maximal value (and a factor 5.6 less than the average).

Likewise for military software the best-in-class size amounts to 50, a factor 4.2 less (average is 77, or a factor of 6.4 less).

duration Best-in-class systems software takes a minimal value of 25 months in the 10K function point range (average is 36 months). The duration of the Maeslantkering software took three years, which does not deviate from best-in-class data. Best-in-class military software takes minimally 40 months (average is 52 months). So this is in line with best-in-class durations for software under military standards.

cost Costs of the project are not disclosed, although we heard that they do not exceed 1% of the total cost of the Maeslantkering (660 million Euro). For an effort of 25 person year we take the highest burdened compensation rates in Jones' benchmark: his maximum is 252K per year. Since the exact price is not disclosed, we will use 252K as a censored data point. This is consistent with the 1% we heard of, since our total is 6.3 million Euro. Best-in-class cost per function point in the 10K range is $1598.47 (average is $3388.73). The Maeslantkering software is built for at most 742 Euro per function point despite the exceptionally high compensation that we used. So, the barrier-software defeats the best-in-class benchmarks significantly. Likewise, for military software the best-in-class cost per function point is $2985.80 (avg $6055.47), so a factor 4+ more than the Maeslantkering software.

productivity Jones also provides productivity per line of code. Best-in-class systems software development in the 10K function point range amounts to 1095 LOC per staff month. The barrier-software took 300 staff months, so a productivity of 1500 LOC per staff month. This is a factor 1.4 higher than best-in-class. For military software the best-in-class productivity measured is 628 LOC per staff month, which is a factor 2.4 lower than the barrier-productivity.

year	project	integrity	size	defects/KLOC	LOC/wday
1992	ATC display	SIL 2	197000	0.75	13
1997	Helicopter landing system	SIL 4	27000	0.22	7
1999	Smart card security	ITSEC E6	100000	0.04	29
2002	Aircraft test set	SIL 0	35000	< 0.1	28
2003	Secure biometrics	CC EAL 5+	10000	0.00	38

Table 1: Sample rates for deployed certified code taken from [1].

other certified software Peter Amey [1] is one of the few who published productivity rates for deployed certified code including all lifecycle phases and management overhead for five systems. Table 1 displays a small SIL 4 system with a productivity of 7 LOC per workday, or 140 LOC per staff month. This is a factor 10 lower than the barrier-software. The other tabulated systems with various certifications display a productivity of 13–38 LOC per workday. These rates are in the same range as Jones' best-in-class benchmarks: 36 for system software and 21 LOC per workday for military software. Furthermore, the number of defects per 1000 lines of code for the SIL 4 helicopter landing system is 0.22, extrapolated to 450000 LOC this amounts to 99 defects—a small fraction of which are severe. This could mean that there are only a few severe errors in the barrier-software, but comparison should also take velocity of defect removal into account.

defect removal velocity Table 1 also shows a 100K smart card system with 4 delivered defects. The ITSEC E6 is certainly equivalent to SIL 4 in requirements process rigor. The barrier-software delivered 3 faults in the first 2 years of operation. However, during customer testing 119 faults were found. In the smart card system 421 defects were found, of which 10 during a customer test, and 4 in operation (3 code errors, 1 specification error) [1]. Multiplying 421 by 4.5 provides 1895 defects, which is the same order of magnitude as the barrier-software. But here 119 errors were detected during customer testing. So the defect removal velocity is much lower compared to the smart card system. Extrapolating this too, it is not unlikely that there are more than the three reported defects in the barrier-software. This is consistent with the reported significant spending on the software while in operation.

appropriateness of the benchmarks Since formal methods, extreme rigor, and many obligatory software development methods were used to construct the Maeslantkering software, one could object against using industry benchmarks. That is why we used the extreme values, and not industry averages. These extremes can serve as bounds. The hypothesis is that the Maeslantkering software should be better than the extremes in a consistent manner. But comparison indicates inconsistencies: you would expect a lower assignment scope, given the many extra activities, but we found a higher scope than ever measured. You would expect a larger staff size due to the extra activities, but we found a much lower staff size than ever measured. Development schedules are comparable but you would expect longer durations, since more activities took place. You would expect a higher price, but we found a much cheaper price than ever measured. Since the system is certified at SIL 4, fault detection and diagnosis techniques must have been used. Still, an extremely low amount of errors is reported during development: only 1655. This is a factor 24 less than the lowest values in Jones' benchmarks.

one size fits all Another objection could be that the size of 8490 function points is erroneous, since backfiring is not an exact science. In fact an insider told us that our function point estimate is plain wrong, but could not provide us with other figures. Another objection could be that the real size to look at is not 450K but 200K since that is in operation, or that the 200K only contains a small amount of critical code, so that we only need to look at subsets of the system. However, IEC 61508 guidelines do not make such distinctions. Indeed, lower function point sizes lead to much lower high-severity errors. If we reduce the size to an unrealistically low minimum in the range of about a thousand function points the minimal number of high-severity defects delivered is two (military software).

Linking fault to failure

Depending on the argument used you'll find more or less residual defects. However, even the most optimistic arguments show that a few high-severity defects are likely to reside in the barrier-software. Is a few faults low, high, or good enough? Let's approach this issue from the other side: how many residual defects are acceptable for safety integrity level 4? Then we have to link fault to failure. For, one could argue that if these defects never surface there is no problem after all. Or put more formally: the failure rate distribution could be such that no failure materializes, and since we have no knowledge about this distribution, we cannot conclude anything. Is there a justifiable way of linking dangerous software faults to dangerous equipment failures without knowing the failure rate distributions? To some extent there is. Bishop and Bloomfield were instrumental in developing a distribution-free result [4, 5, 3] that shows that, under the following conditions:

- a test interval T using an unchanging operational profile
- there are N residual faults in the software
- a fault is fixed once d failures have occurred

the worst-case failure rate λ after time T is bounded by:

$$\lambda \leq Nd/eT \leq 10\lambda$$

where e is the basis of the natural logarithm ($e \approx 2.718$). The theory gives a worst case bound for a given operational profile, and there are empirical arguments that the best case failure rate should be no more than an order of magnitude better than the bound, hence the 10 in the upper bound.

The distribution-free fault-to-failure bounds show that the reliability of software increases when the operating interval T increases. So the longer the system is in operation without failing the more reliable we expect it to become. Furthermore, the model is robust over the long term with respect to a number of assumption violations: non-stationary input distributions, faulty corrections and imperfect diagnosis [4]. In practice this implies that over long periods of time volatile input distributions are averaged out and faulty corrections approximate the expected failure rate again. For imperfect diagnosis we should use a $d > 1$ since poor diagnosis has the effect of scaling up the failure rate contribution of each fault. However for a high SIL system, we would expect all faults to be fixed so we assume that $d = 1$.

We can use the worst case bound formula *in reverse* to compute the number of residual faults needed to achieve a given failure target (given some level of operational testing). For example, if we assume:

- pre-release testing is ten years, so $T = 10$
- a failure target of 10^{-4} per year after testing T years, so $\lambda = 10^{-4}$

then from the above formula we require that $e \cdot 10^{-3} < N < e \cdot 10^{-2}$. This is a so-called fractional error which should be interpreted as follows. In the worst-case if we implemented the software 368 times we would only expect 1 dangerous fault to be found after delivery to the customer, and in the best-case only one dangerous fault in 37 implementations of the software.

In Table 1, the helicopter landing system SIL 4 software contains $2.2 \cdot 10^{-4}$ faults per line of code, which is in the order of 10^{-4} faults per line of code. Best-in-class benchmarks show delivery of 38 defects for systems software in the 1000 function point range [7]. For C++ code this amounts to $7.2 \cdot 10^{-4}$ faults per line of code. So one could say that best practice can achieve in the order of 10^{-4} faults per LOC. With the required fractional error band of $e \cdot 10^{-3} < N < e \cdot 10^{-2}$, we need the software to be at most 27 lines of code. Even if we assume that only 10% of faults are dangerous and the actual failure rate is only 10% of the worst case bound, the maximum program size would be 2700 lines.

So even with generous assumptions about test time and dangerous fault percentages, we cannot be certain we can reach this goal for realistic software. If we use more realistic figures, e.g., a 200K line program and a fault density of $5 \cdot 10^{-4}$/LOC at the start of customer acceptance we would expect 100 residual faults at the start of customer acceptance (indeed 119 were found for the Maeslantkering). If we further assume that customer acceptance testing is equivalent to 10 operational years, the worst case bound theory predicts a worst case failure rate at the start of operation of 3.7 failures/year and the best case failure rate is 0.37 failures/year. The observed rate of 3 faults in the first 2 years of operation of the Maeslantkering software is within the band predicted by the theory.

Although we know there were at least 3 observed faults, we will use the inequalities to calculate the acceptable bandwidth of dangerous residual defects, for the SIL 4 level of the barrier-software. We need to solve N from the inequalities below:

$$\lambda_i \leq Nd/eT \leq 10\lambda_i$$

where λ_i is the maximum dangerous failure rate permitted in the SIL i band ($i = 1, 2, 3, 4$). We assume that faults are always fixed, so $d = 1$. We take $i = 4$, since the barrier-software was certified at the SIL 4 level. From public records, we can estimate the upper-bound of the test interval T. A publication of October 2000 reports three faults in the Maeslantkering software since the system became operational, which was October 1998 [11]. This means that in the best case the test interval is two years divided by three faults, which amounts maximally to 5844 hours of uninterrupted testing the system while in operation. The maximal SIL 4 dangerous failure rate is $10^{-8}/h$. So solving N amounts to the following dangerous residual failure band $1/6294 \leq N \leq 1/629$. Recall that N is a fractional error: at best that if we implement over six hundred software systems for the surge barrier only one dangerous fault may be present in one of those 600 systems. Fact is that we know that there were 119 errors during customer-acceptance testing, and 3 after, so N is at least 3. We also know that no catastrophes occurred, so let us optimistically assume that our testing interval T is the total operating time of the barrier, which is at the time of writing eight years. This yields a band of dangerous residual fractional errors of: $1/524 \leq N \leq 1/52$. So even with these optimistic assumptions we are way off the required failure band.

What about per demand?

In the above calculations we used a time-frame of continuous operation, and some may argue that a per demand calculation is more appropriate, given the low usage-frequency. On a per demand basis, the same model can be used, only the time interval T becomes the number of test demands D in a realistic operational profile, and the SIL bands take other values. A presentation by people involved in the construction of the barrier-software given at February 4, 2000 stated: "Since then it has rightly been in alert twice", which refers to the barrier-software that was alert and took the right decision. It was never necessary to close it due to bad weather conditions. Although test closures are done by perfect weather conditions (see Figure 1) we will count all seven test-closures. There is a simulator supporting development, and potentially also the operational software. The difficulty with such testing could be that the simulation omits key elements of the true operational profile, like specific weather conditions, changes in operating mode, input/output failures, configuration errors, computing system failures, restarts (think of a power outage), extended intervals between demands that could lead to internal state corruption (like memory leaks), and more. So we are hesitant to count such virtual closures. Therefore, the number of realistic test demands is set to 9. Our analogous formula uses the same notation as before with two exceptions:

$$\Lambda_i \leq Nd/eD \leq 10\Lambda_i$$

here Λ_i is the maximum dangerous failure rate per demand permitted in the SIL i band ($i = 1, 2, 3, 4$) and D is the number of test demands during an unchanging operational profile. For the given SIL level 4, the per-demand failure-rate is 10^{-4}. Using the above formula we find that $1/408 \leq N \leq 1/41$. So, despite the per demand variant, the conclusion remains the same: such fractional residual dangerous fault rates are in our opinion unrealistic.

simulations There may be reasons for believing that even the best-case dangerous failure rate the theory predicts is still lower than in reality. For instance, only a small percentage of failures are actually dangerous, because only a small subset of the faults are safety-critical. Or the worst-case-bound prediction is far too pessimistic, e.g., because all the faults are likely to be found in customer acceptance or early operation. Or there has been a lot of realistic demand-based testing to accelerate test time using the simulator to accelerate testing time (simulation of formal models was used). Using the demand-based variant and the 119 found errors during customer acceptance testing we need up to 437776 test demands to reach 10^{-4} per demand. If we generously assume the 8 years of operation to be test-time, this implies up to 150 simulations per day during its 8 years of operation. We do not have any information to determine whether such a mitigating factor applies in the case of the barrier-software.

Conclusions

Affordable, high-speed production of super-reliable software is the holy grail of software engineering, and we do not exclude the possibility that this is achieved in case of the barrier-software. However, the published data in combination with our analysis suggests that it is highly unlikely that the failure rate of the software controlling the Maeslantkering falls within the SIL 4 band: once every ten thousand years.

On a reassuring note, even if the software failure rate is higher than 10^{-4} per demand, it does not necessarily imply that the overall barrier behavior is unsafe. Most safety-related industries employ diversity to achieve top-level safety goals. For example, in the nuclear industry, diverse reactor shutdown systems are used. In the case of the storm surge barrier, the relatively slow response times make it feasible for the diverse actuation to be implemented by manual override. Typical human error rates are below 10^{-4} per operation, but can be increased with suitable procedures and independent cross-checking. So a combination of computers and operators could achieve high safety targets despite relatively low software reliability. Therefore, it is good news that the Dutch government decided to put a 16-person team in place when closure is apparent.

In nuclear plants, airplanes, dams, tankers, automobiles, more and more dependable software is present, and similar software reliability questions will need an answer. In our opinion, both Jones' benchmarks, and the empirically validated distribution-free fault-to-failure bounds by Bishop and Bloomfield are useful for long-term predictions. We can measure defect removal efficiency much easier than potential failure in the future. There are techniques available to increase the defect removal efficiency, and with historical data we can estimate the number of residual (high-severity) defects. We believe that the IEC 61508 approach recommending best practices is a good start, but this should be augmented with software-specific integrity levels expressed in bandwidths of residual high-severity defects. With the approach illustrated with the barrier-software we can then provide defect removal efficiencies and bounds for failure rates, so that realistic quantification of failure rates for safety-critical software-intensive systems becomes a reality.

References

[1] P. Amey. IEC 61508-conformant software development with SPARK, 2005. `www.rvs.uni-bielefeld.de/Bieleschweig/fifth/download/B5-Amey.pdf`.

[2] S. van Baars. The horizontal failure mechanism of the Wilnis peat dyke. *Géotechnique*, 55(4):319–323, 2005.

[3] P.G. Bishop. SILs and Software. *UK Safety Critical Systems Club Newsletter*, 2005. Available via `www.adelard.com`.

[4] P.G. Bishop and R.E. Bloomfield. A Conservative Theory for Long-Term Reliability Growth. *IEEE Trans. Reliability*, 45(4):550–560, 1996.

[5] P.G. Bishop and R.E. Bloomfield. Worst Case Reliability Prediction Based on a Prior Estimate of Residual Defects. In *Proceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE '02)*, pages 295–303, 2002.

[6] M. Chaudron, J. Tretmans, and K. Wijbrans. Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – World Congress on Formal Methods in the Development of Computing Systems II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1511–1526. Springer-Verlag, 1999.

[7] C. Jones. *Software Assessments, Benchmarks, and Best Practices*. Information Technology Series. Addison-Wesley, 2000.

[8] M.P.C. de Jong. *Origin and prediction of seiches in Rotterdam harbour basins*. PhD thesis, Delft University of Technology, 2004.

[9] M.P.C. de Jong, L.H. Holthuijsen, and J.A. Battjes. Generation of seiches by cold fronts over the southern North Sea. *Journal of Geophysical Research*, 108(C4):14.1–14.10, 2003.

[10] C. Perrow. *Normal Accidents – Living with High Risk Technologies*. Princeton University Press, 1984.

[11] J. Tretmans, K. Wijbrans, and M. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System. Technical Report SVC Report II-06-a-1.1, System Validation Centre, Telematics Institute, 2000. `https://doc.telin.nl/dscgi/ds.py/Get/File-11356/II-06-a.pdf`.

[12] J. Tretmans, K. Wijbrans, and M. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System – Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design*, 19:195–215, 2001.

[13] J.L.M. Vrancken. The human factor in system reliability – The case of the Maeslant movable storm surge barrier in the Netherlands, 2006. In submission.