Safety-Critical Systems
Transcript

Date: Tuesday, 10 January 2017 - 6:00PM
Location: Museum of London
Safety-Critical Systems
Professor Martyn Thomas

Introduction

Computer systems are used in many safety applications where a failure may increase the risk that someone will be injured or killed. We may distinguish between safety-related systems where the risk is relatively small (for example the temperature controller in a domestic oven) and safety-critical systems where the risk is much higher (for example the interlocking between the signals and points on a railway).

The use of programmable systems in safety applications is relatively recent. This lecture explores the difficulties of applying established safety principles to software based safety-critical systems.

The Causes of Accidents

Many accidents do not have a single cause. Suppose that a motorist is driving their new sports car to an important meeting, skids on a patch of oil on the road around a corner, and hits a road sign. What caused the accident? Was it the speed, the oil, the position of the road sign or something else?

Was the accident caused by the motorist driving too fast? If so, was it because they were late for their meeting, or because they were trying out the cornering abilities of their new car, or because they were thinking about the important meeting and not paying attention to their speed, or because they had not had appropriate driving tuition? If they were late, was it because their alarm clock had failed, or because one of their children had mislaid their school books, or...

Was the accident caused by the oil on the road? It was there because the gearbox on a preceding lorry had leaked, but was that caused by a design fault, or poor maintenance, or damage from a pothole that hadn’t been repaired by the local council, or ...

Was the accident caused by the position of the road sign? It was placed there because there were road works ahead, because the road edge has subsided, because it had been undermined by water running from a poorly maintained farm stream and an overweight lorry had passed too close to the edge.

What should we say caused this accident? This is an artificial example but many real-life accidents are at least as complex as this. Accidents happen because several things combine together. There may be many contributory factors and no single cause.

But many accidents are attributed to errors by human operators of equipment – pilots and car drivers, for example. As software replaces more and more of these human operators, the scope for software failures to be a leading contributor to accidents will increase.

Safety and Engineering

Many of the principles and techniques that are used to develop and assess safety-critical systems have their origins in the process industries, such as chemical plants and oil refineries, when these were relatively simple and controlled manually through valves, switches and pumps, monitored by thermometers and pressure gauges and protected by pressure release valves and alarms.

In such systems, the common causes of failure were pipes becoming corroded and fracturing, valves sticking, or other physical problems with individual components. The safety of the whole system depended on the reliability of individual components, so systems were (and still are) designed to eliminate (so far as practicable) single points of failure where one component could fail and cause an accident. In those cases where the single point of failure cannot be eliminated (the rotor blades in a helicopter, for example) these critical components have to be designed and built in such a way that failure is extremely improbable. A process of hazard analysis attempts to identify all the dangerous states that the system could get into, and fault trees are drawn up to analyse the circumstances that could lead to each dangerous occurrence.

Engineers distinguish between faults and failures: a fault is defined as an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure, and failure is defined as the lack of ability of a component, equipment, sub system, or system to perform its intended function as designed. A failure may be the result of one or many faults.
Having designed the system, the engineers will often consider each component and subsystem and analyse what would happen if it failed, drawing on knowledge of the different ways that such components can fail (to give a trivial example, a valve may stick open or shut). This process is called Failure Modes and Effects Analysis (FMEA). A related process that also considers criticality is called FMECA. There are rigorous techniques that a safety analyst may use to analyse the behaviour of complex technical and socio-technical systems but most FMEA / FMECA analyses currently rely on intuition and judgement.

After all this analysis, the engineers can calculate how reliable they need each component to be for the overall system to be adequately safe. But this poses the following question:

**Safety Requirements: how safe is safe enough?**

“How safe is safe enough” is a social question rather than a scientific or engineering one. It is a question that can be asked in a variety of ways, such as “what probability of failure should we permit for the protection system of this nuclear reactor?”, “what probability of failure should we permit for safety-critical aircraft components?” or “how much should we spend to avoid fatal accidents on the roads or railways?”. From the answers chosen in each context, you can calculate the value placed on a statistical life or VSL. This is defined as the additional cost that individuals would be willing to bear for improvements in safety (that is, reductions in risks) that, in the aggregate, reduce the expected number of fatalities by one.

The calculated VSL varies quite widely but it can be important to use a consistent figure, so that (for example) the Department of Transport can decide where it would be most beneficial to spend road safety money, and the Department of Health can decide how to allocate NHS budgets between different ways of improving health outcomes. The VSL can be calculated in a variety of ways and the value differs in different countries.

In Great Britain, the health and safety of workers and others who may be affected by the risks from work are regulated by the 1974 Health and Safety at Work Act (HSWA) which is enforced primarily by the Health and Safety Executive, HSE. HSWA states (as general duties) that

> it shall be the duty of every employer to ensure, so far as is reasonably practicable, the health, safety and welfare at work of all his employees and it shall be the duty of every employer to conduct his undertaking in such a way as to ensure, so far as is reasonably practicable, that persons not in his employment who may be affected thereby are not thereby exposed to risks to their health or safety.

The phrase so far as is reasonably practical means that someone who creates a risk to themselves, their employees or the public has a duty under the HSWA to assess the risk and, if it is not so low as to be clearly tolerable, they must take action to reduce the risk so that it is As Low as is Reasonably Practicable. This is known as the ALARP principle. HSE has explained ALARP as follows.

The definition set out by the Court of Appeal (in its judgment in Edwards v. National Coal Board, [1949] 1 All ER 743) is:

> “Reasonably practicable‘ is a narrower term than ‘physically possible‘… a computation must be made by the owner in which the quantum of risk is placed on one scale and the sacrifice involved in the measures necessary for averting the risk (whether in money, time or trouble) is placed in the other, and that, if it be shown that there is a gross disproportion between them – the risk being insignificant in relation to the sacrifice – the defendants discharge the onus on them.”

In essence, making sure a risk has been reduced ALARP is about weighing the risk against the sacrifice needed to further reduce it. The decision is weighted in favour of health and safety because the presumption is that the duty-holder should implement the risk reduction measure. To avoid having to make this sacrifice, the duty-holder must be able to show that it would be grossly disproportionate to the benefits of risk reduction that would be achieved. Thus, the process is not one of balancing the costs and benefits of measures but, rather, of adopting measures except where they are ruled out because they involve grossly disproportionate sacrifices. Extreme examples might be:

> To spend £1m to prevent five staff suffering bruised knees is obviously grossly disproportionate; but

> To spend £1m to prevent a major explosion capable of killing 150 people is obviously proportionate.

Of course, in reality many decisions about risk and the controls that achieve ALARP are not so obvious. Factors come into play such as ongoing costs set against remote chances of one-off events, or daily expense and supervision time required to ensure that, for example, employees wear ear defenders set against a chance of developing hearing loss at some time in the future. It requires judgment. There is no simple formula for computing what is ALARP.
There is much more guidance on the HSE website. I particularly recommend the 88pp publication Reducing risks, protecting people from which the following diagram is taken.

In many industries there are agreed standards or certification requirements that provide guidance on the question how safe is safe enough? One example is the DO-178 guidance for certifying flight-critical aircraft components (including software). The FAA (for the USA) and EASA (for Europe) require that catastrophic failure conditions (those which would prevent continued safe flight and landing) should be extremely improbable (which you may find comforting) though neither regulation gives an equivalent numerical probability for this phrase.

John Rushby’s excellent review of the issues in certifying aircraft software explains

Neither FAR 25.1309 nor CS 25.1309 defines “extremely improbable” and related terms; these are explicated in FAA Advisory Circular (AC) 25.1309 and EASA Acceptable Means of Compliance (AMC) 25.1309. These state, for example, that “extremely improbable” means “so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type”. AC 25.1309 further states that “when using quantitative analyses. . . numerical probabilities. . . on the order of 10-9 per flight-hour may be used. . . as aids to engineering judgment . . . to . . . help determine compliance“ with the requirement for extremely improbable failure conditions. An explanation for this figure can be derived as follows: suppose there are 100 aircraft of the type, each flying 3,000 hours per year over a lifetime of 33 years (thereby accumulating about 107 flight-hours) and that there are 10 systems on board, each with 10 potentially catastrophic failure conditions; then the “budget” for each is about 10-9 per hour if such a condition is not expected to occur in the entire operational life of all airplanes of the type. An alternative explanation is given in Section 6a of AMC 25.1309): the historical record for the previous (pre-software-intensive) generation of aircraft showed a serious accident rate of approximately 1 per million hours of flight, with about 10% due to systems failure; the same assumption as before about the number of potentially catastrophic failure conditions then indicates each should have a failure probability less than 10-9 per hour if the overall level of safety is to be maintained. Even though recent aircraft types have production runs in the thousands, much higher utilization, and longer service lifetimes than assumed in these calculations, and also have a better safety record, AMC 25.1309 states that a probability of 10-9 per hour has “become commonly accepted as an aid to engineering judgement” for the “extremely improbable” requirement for catastrophic failure conditions.

So the target probability of catastrophic failure for safety-critical systems in aircraft is 10-9 per hour, or one failure in a billion hours. It is interesting to contrast this with the COMMON POSITION OF INTERNATIONAL NUCLEAR REGULATORS AND AUTHORISED TECHNICAL SUPPORT ORGANISATIONS on the licensing of safety critical software for nuclear reactors, which states: “Reliability claims for a single software based system important to safety of lower than 10-4 probability of failure (on demand or dangerous failure per year) shall be treated with extreme caution”. We shall shortly consider the problems of establishing low failure probabilities for software-based systems and some of the proposed solutions.

Hazard, Risk, Safety and Reliability

Before we continue, it is important to distinguish what we mean by a hazard and a risk. A hazard is anything that may cause harm. The risk is the chance, high or low, that somebody could be harmed by these and other hazards, together with an indication of how serious the harm could be. A hazard is something that could lead to harm (an accident) and a risk is the combination of the probability that the hazard will lead to an accident and the likely severity of the accident if it occurs. When engineers design safety-critical systems, they try to identify all the potential hazards that their system creates or that it should control: for a chemical factory this might include the release of toxic materials, or an explosion or a fire. For each hazard, the risk is assessed and if the risk is not acceptable but can be made tolerable, measures are introduced to reduce it ALARP. This ALARP approach is used in Britain and in other countries with related legal systems, but not universally.

It is also important to distinguish safety from reliability. A system can be safe (because it cannot cause harm) and yet be unreliable (because it does not perform its required functions when they are needed). If your car often fails to start, it may be perfectly safe though very unreliable. In general, if a system can fail safe then it can be safe whilst being unreliable if the failures occur too often.

Software-based systems

Software-based systems are often very complex. A modern car contains 100 million lines of software, some of it safety-critical. As we have seen above, the standards for assuring the safety of electronic systems originated in the approach adopted by the process industries; this approach was based on analysing how the reliability of system components contributed to the overall safety of the system. But failures can be random or systematic, where random failures are typically the result of physical changes in system components, because of wear, corrosion, contamination, ageing or other physical stresses. In contrast, systematic failures are intrinsic in the system design; they will always occur whenever particular circumstances co-exist and the system enters a state that has not been correctly designed or implemented. Historically, safety engineers assumed that systematic failures would be eliminated by following established engineering best practices, so the possible effects of
systematic failures were ignored when calculating failure probabilities for electro-mechanical control systems.

Software does not age in the way that physical components do (though it may become corrupted in store, for example by maintenance actions or corruption of its physical media, and it may be affected by changes in other software or hardware, such as a new microprocessor or a new compiler). Software failures are, in general, the result of software faults that were created when the system was specified, designed and built and these faults will cause a failure whenever certain circumstances occur. They are systematic failures but they are sufficiently common that it would be difficult to justify ignoring them in the safety certification of software-based systems.

Some engineers have argued that it is a category mistake to talk about software failing randomly; they argue that software cannot be viewed as having a probability of failure. Their reasoning is that because every bug that leads to failure is certain to do so whenever it is encountered in similar circumstances, there is nothing random about it, so software does not fail randomly. This argument is wrong, because any real-world system operates in an uncertain environment, where the conditions that the system encounters and has to process will arise randomly and the resulting failures will occur randomly; it is therefore perfectly reasonable to treat the failures of software-based systems as random.

There are two sources of uncertainty about when and how often software will fail: uncertainty about the inputs it will encounter in a given period because the inputs occur randomly (aleatoric uncertainty) and uncertainty about whether a particular sequence of inputs will trigger a failure because of limited knowledge about the total behaviour of a complex program (epistemic uncertainty). Probability is the mathematical tool that is used to reason about uncertainty; engineers therefore talk about the \( pfd \) (probability of failure on demand – used for systems that are rarely executed, such as the protection systems in nuclear reactors) and the \( pfh \) (probability of failure per hour – used for systems that operate continuously, such as the control software in a modern car or aircraft).

It is a symptom of the immaturity of the software engineering profession that the debate about whether or not software failures can be regarded as random still continues on professional and academic mail groups, despite two decades of authoritative, peer reviewed papers on software failure probabilities.

As we have seen in previous lectures, the primary ways that software developers use to gain confidence in their programs are through testing and operational experience, so let us now consider what we can learn from these methods.

**How reliable is a program that has run for more than a year without failing?**

A year is 8760 hours, so let us assume that we have achieved 10,000 hours of successful operation; this should give us some confidence in our program, but just how much? What can we claim as its probability of failure per hour (\( pfh \))?

When calculating probabilities from evidence, it is necessary to decide how confident you need to be that the answer is correct. It is self evident that the more evidence you have, the more confident you can be: if you toss a coin 100 times and get roughly the same number of heads as tails, you may believe it is a unbiased coin, if the numbers are close after 1 million tosses, your confidence should be much higher. In the same way, to know how much evidence you need to support a particular claim for a probability, you must first specify the degree of confidence you need to have that you are right.

For a required failure rate of once in 10,000 hours (a \( pfh \) of 1/10000 or 10^{-4}), 10,000 hours without a failure provides 50% confidence that the target has been met; but if you need 99% confidence that the \( pfh \) is at least 10^{-4} you would need 46,000 hours of tests with no failures. As an engineer’s rule-of-thumb, these multipliers can be used to indicate the hours of fault-free testing or operational experience that you need to justify a claim that your \( pfh \) (or the number of demands if you are claiming a \( pfd \) – the maths come out close enough for the relevant probability distributions). So to claim a \( pfh \) or \( pfd \) of no greater than 10^{-n}, you need at least 10^n failure free hours or demands if you are happy with 50% confidence that your claim is correct. If you want to be 99% confident then you need at least \( 4.6 \times 10^n \) failure free hours or demands. (Similar calculations can be done to determine the amount of evidence needed if the program has failed exactly once, twice, or any other number of times. The increase is relatively small if the number of failures is small. For details see the referenced papers).

The reasoning above means that if a program has run for a year without failing, you should only have 50% confidence that it will not fail in the following year.

These probability calculations depend on three critical assumptions:

- That the distribution of inputs to the program in the future will be the same as it was over the period during which the test or operational evidence was collected. So evidence of fault-free working in one environment should not be transferred to a different environment or application.
- That all dangerous failures were correctly detected and recorded.
• That no changes have been made to the software during the test or operational period.

These are onerous assumptions and if any one of them cannot be shown to be true then you cannot validly use the number of hours of successful operation to conclude that a program is highly reliable. This means that you must be able to justify very high confidence in your ability to predict the future operating environment for your software and, if the software is changed in any way, all the previous evidence from testing or operation should be regarded as irrelevant unless it can be proved that the software changes cannot affect the system failure probability. Most arguments for the re-use of software in safety-critical applications fail all three of the above criteria, so safety arguments based on re-used software should be treated with a high degree of scepticism, though such proven in use arguments are often put forward (and far too often accepted by customers and regulators).

We saw earlier that the certification guidance for flight-critical software in aircraft is a $p_{f}h$ of 10-9 which would require 4.6 billion failure free hours (which is around 500,000 years) to give 99% confidence. In contrast, the certification of the software based primary protection system for the Sizewell B nuclear reactor only required a $p_{f}d$ of 10-3, which was feasible to demonstrate statistically. At the other extreme, it is reported that some railway signalling applications require failure rates no worse than 10-12 per hour, which is far in excess of anything for which a valid software safety argument could be constructed.

All these considerations mean that software developers have a major problem in generating valid evidence that their software is safe enough to meet certification requirements or ALARP. Yet safety-critical software is widely used and accident rates seem to be acceptable. Does this mean that the certification requirements have been set far too strictly? Or have we just been fortunate that most faults have not yet encountered the conditions that would trigger a failure? Should the increasing threat of cyber-attack change the way that we answer these questions?

### N-version Programming

Some academics and some industry groups have tried to overcome the above difficulties by arguing that if safety-critical software is written several times, by several independent groups, then each version of the software will contain independent faults and therefore fail independently. On this assumption, the different versions could all be run in parallel, processing the same inputs, and their outputs could be combined through a voting system. On the assumption of independence and a perfect voter, three 10-3 systems could be combined to give one 10-9 system. N-version programming is a software analogue to multi-channel architectures that are widely used to mitigate hardware failures.

N-version programming has strong advocates but it has been shown to be flawed. An empirical study by John Knight and Nancy Leveson undermined the assumption of independence that is critical to the approach, and other analyses, both empirical and theoretical, have reached the same conclusion. Professors Knight and Leveson have responded robustly to critics of their experimental work. Despite these flaws, the idea remains intuitively attractive and it is still influential.

One interesting approach has been to combine two software control channels, one of which is assumed to be "possibly perfect", perhaps because it has been developed using mathematically formal methods and proved to meet its specification and to be type safe. A paper by Bev Littlewood and John Rushby, called *Reasoning about the Reliability Of Diverse Two-Channel Systems In which One Channel is "Possibly Perfect"*, analyses this approach.

### Standards for developing Safety Critical Systems: DO-178 and IEC 61508

The leading international standards for software that implements safety-critical functions (DO-178C for aircraft software, and IEC 61508 and its industry-specific derivatives) do not attempt to provide scientifically valid evidence for failure probabilities as low as 10-9 per hour or even 10-6 per hour. Instead, they define a hierarchy of **Software Integrity Levels** (SILs – the IEC 61508 term) or **Development Assurance Level** (DAL – the DO-178 term) depending on the required reliability of the safety function that is providing protection from a hazard. The standards recommend or require that the software is specified, designed, implemented, documented and tested in specified ways, with stricter requirements applying where a lower probability of failure is required.

Many criticisms can be levelled at this approach.

• There is at best a weak correlation between the methods used to develop software and its failure rate in service. The exception to this is where the software has been mathematically specified and analysed and proved to implement the specification and to be devoid of faults that could cause a failure at runtime (and even here there is insufficient experience of enough systems to establish it empirically). The standards allow this approach but do not require it.
Some of the recommended and required activities are costly and time-consuming (such as the DO178 Level A requirement to carry out MC/DC testing) and although though they may detect errors that would not otherwise have been found, it is not clear that these activities are a good use of safety assurance resources.

There is no evidential basis for the way that recommended engineering practices have been assigned to SILs or DALs. For example, IEC 61508 specifies four SILs for safety functions that operate continuously, as below

**Safety Integrity Level - Probability of dangerous failure per hour**

| SIL 4 | >=10^-9 to <10^-8 |
| SIL 3 | >=10^-8 to <10^-7 |
| SIL 2 | >=10^-7 to <10^-6 |
| SIL 1 | >=10^-6 to <10^-5 |

Since (as we have seen) there is no way to demonstrate empirically that even SIL 1 pfh has been achieved, it is unclear how there could be evidence that additional development practices would reduce the failure rate by factors of 10, 100, or 1000. Indeed, there is no evidence that employing all the practices recommended for SIL 4 would actually achieve even SIL 1 with any certainty.

- The standards do not adequately address the safety threats from security vulnerabilities.

One evaluation of software developed in accordance with DO-178 showed no discernable difference in the defect levels found in DAL A software and lower DAL software. I discussed this evaluation in my first Gresham lecture, *Should We Trust Computers?*.

A 2007 report from the United States National Academies: *Software for Dependable Systems: Sufficient Evidence?* makes three strong recommendations to anyone claiming that software is dependable:

Firstly on the **need to be explicit about what is being claimed**.

No system can be dependable in all respects and under all conditions. So to be useful, a claim of dependability must be explicit. It must articulate precisely the properties the system is expected to exhibit and the assumptions about the system’s environment on which the claim is contingent. The claim should also make explicit the level of dependability claimed, preferably in quantitative terms. Different properties may be assured to different levels of dependability.

Secondly on **Evidence**.

For a system to be regarded as dependable, concrete evidence must be present that substantiates the dependability claim. This evidence will take the form of a “dependability case,” arguing that the required properties follow from the combination of the properties of the system itself (that is, the implementation) and the environmental assumptions. So that independent parties can evaluate it, the dependability case must be perspicuous and well-structured; as a rule of thumb, the cost of reviewing the case should be at least an order of magnitude less than the cost of constructing it. Because testing alone is usually insufficient to establish properties, the case will typically combine evidence from testing with evidence from analysis. In addition, the case will inevitably involve appeals to the process by which the software was developed—for example, to argue that the software deployed in the field is the same software that was subjected to analysis or testing.

Thirdly on **Expertise**.

Expertise—in software development, in the domain under consideration, and in the broader systems context, among other things—is necessary to achieve dependable systems. Flexibility is an important advantage of the proposed approach; in particular the developer is not required to follow any particular process or use any particular method or technology. This flexibility provides experts the freedom to employ new techniques and to tailor the approach to their application and domain. However, the requirement to produce evidence is extremely demanding and likely to stretch today’s best practices to their limit. It will therefore be essential that the developers are familiar with best practices and diverge from them only with good reason. Expertise and skill will be needed to effectively utilize the flexibility the approach provides and discern which best practices are appropriate for the system under consideration and how to apply them.

The recommended approach leaves the selection of software engineering and safety assurance methods under the control of the developers. It is a goal-based approach, very much consistent with the philosophy of the Health and Safety at Work Act, that those who create a hazard have the duty to control it and should be held
accountable for the outcome, rather than being told what they have to do and then being able to claim that any accidents are not their fault because they followed instructions.

I know of only one published standard for safety-critical software that follows this goal-based approach, CAP 670 SW01 Regulatory Objectives for Software Safety Assurance in Air Traffic Service Equipment from the Safety Regulation Group of the UK Civil Aviation Authority. They have also published an accompanying guide on Acceptable Means of Compliance.

Final Observations

Safety-critical computer systems are used widely and, on currently available evidence, many of them seem to be fit for purpose. Where systems have been certified or claimed to have a specified probability of failure, the evidence to support that claim is rarely, if ever available for independent review and in the case of claims of $pfh$ lower than 10-4 per hour (which includes all claims for SILs 1,2,3 and 4 using the IEC 61508 international standard or its derivatives) it appears to be scientifically infeasible to produce valid evidence for such a claim.

Current standards also treat the security threat to safety inadequately, if at all, and yet the possibility of a serious cyberattack is a Tier One Threat on the national risk register.

It seems certain that many more safety-critical software-based systems will be introduced in the future, and it is essential that the software industry adopts engineering methods that can provide strong evidence that the risks from both systematic failure and cyberattack have been reduced so far as reasonably practicable. (Indeed, as we have seen, this could be said to be a legal obligation in Britain under Sections 2 and 3 of the Health and Safety at Work Act). Unfortunately, the history of engineering suggests that major improvements in new engineering methods usually only come about after major accidents, when an official investigation or public inquiry compels industry to change.

© Professor Martyn Thomas, 2017