Software Safety: Why is there no Consensus?

John A McDermid, David J Pumfrey; University of York, Heslington, York, YO10 5DD, UK

Keywords: software safety, software assurance, software safety evidence

Abstract

Development and assessment of safety critical software is governed by many standards. Given the growing dependence on software in a range of industries, one might expect to see these standards reflecting a growing maturity in processes for development and assessment of safety critical software, and an international consensus on best practise. In reality, whilst there are commonalities in the standards, there are also major variations in the standards among sectors and countries. There are even greater variations in industrial practices. This leads us to consider why the variation exists and if any steps can be taken to obtain greater consensus.

In this paper we start by clarifying the role of software in system safety, and briefly review the cost and effectiveness of current software development and assurance processes. We then investigate why there is such divergence in standards and practices, and consider the implications of this lack of commonality. We present some comparisons with other technologies to look for relevant insights. We then suggest some principles on which it might be possible to develop a cross-sector and international consensus. Our aim is to stimulate debate, not to present a "definitive" approach to achieving safety of systems including software.

The meaning of software safety

Software is becoming an increasingly important element of many safety-critical and safety-related systems. In many cases, software is the major determinant of system capability, e.g., in aircraft flight control systems, body electronics for cars, air traffic control, pacemakers and nuclear power plant control. For such systems, the software is often a major factor in the costs and risks of achieving and assuring safety. Costs can be of the order of $10M for each system on a modern aircraft.

Some people argue that the term "software safety" is a misnomer as software, in and of itself, is not hazardous, i.e., it is not toxic, does not have high kinetic energy, and so on. Here we use the term "software safety" simply as a shorthand for "the contribution of software to safety in its system context". Software can contribute to hazards through inappropriate control of a system particularly where it has full authority over some hazardous action. By "full authority" we mean that no other system or person can over-ride the software. Software can also contribute to hazards when it's behaviour misleads system operators, and the operators thereby take inappropriate actions. Misleading operators is most critical if they have no means of cross-checking presented information, or have learnt to trust the software even when there are discrepancies with other sources of data.

Described this way, software is much like any other technology, except that it can only contribute to unsafe conditions due to systematic causes, i.e., there is no "random" failure or "wear-out" mechanism for software. Systematic failures arise from flaws or limitations in the software requirements, in the design, or in implementation. Thus, software safety involves the consideration of how we can eliminate such flaws and how we can know whether or not we have eliminated said flaws. We refer to these two concerns as *achieving* and *assuring* safety.

Why is there a concern?

There is considerable debate on software safety in industry, academia, and government circles. This debate may seem slightly surprising, as software has a remarkably good track record. There have been several high-profile accidents, e.g., Ariane 5 ([1]) and Therac 25 ([2]), and in aerospace the Cali accident has been attributed to software (more strictly data ([3])), but a study of over 1,000 apparently "computer related" deaths ([4]) found that only 34 could be attributed to software issues. The critical failure rate of software in aerospace appears to be around 10^{-7} per hour ([5]), which is sufficient for it to have full authority over a hazardous/severe major event and still meet certification targets. In fact, most aircraft accidents stem from mechanical or electrical causes, so why is there a concern?

We believe the concern arises out of four related factors. First, there is some scepticism related to the accident data. It is widely believed that many accidents put down to human error were actually the result of operators (e.g., pilots) being misled by the software. Also, software failures typically leave no trace, and so may not be seen as contributory causes to accidents (e.g., the controversy over the Chinook crash on the Mull of Kintyre ([6])). Further, much commercial software is unreliable, leading to a general distrust of software. Many critical systems have a long history of "nuisance" failures, which suggests that more problems would arise if software had greater authority. Second, systems and software are growing in complexity and authority at an unprecedented rate, and there is little confidence that current techniques for analysing and testing software will "keep up." There have already been instances of projects where "cutting edge" design proposals have had to be rejected or scaled down because of the lack of suitable techniques for assuring the safety of the product. Third, we do not know how to measure software safety; thus it is hard to manage projects to know what are the best and most effective techniques to apply, or when "enough has been done". Fourth, safety critical software is perceived to cost too much. This is both in relation to commercial software, and in relation to the cost of the rest of the system. In modern applications, e.g., car or aircraft systems, it may represent the majority of the development costs. These issues are inter-related; that is, costs will rise as software complexity increases.

The cost and effectiveness of current practises

It is difficult to obtain accurate data for the cost and effectiveness of software development and assessment processes as such information is sensitive. The data in the following sections are based on figures from a range of critical application projects primarily from aerospace projects based in Europe and the USA; however we are not at liberty to quote specific sources.

Costs: Costs from software requirements to the end of unit testing are the most readily comparable between projects. Typically 1-5 lines of code (LoC) are produced per man day, with more recent projects being near the higher figure. Salary costs vary, but we calculate a mid-point of around \$150 to \$250 per LoC, or \$25M for a system containing 100 kLoC of code.

Typically, testing is the primary means of gaining assurance. Although the costs of tests vary enormously, e.g. with hardware design, testing frequently consumes more than half the development and assessment budget.

Also, in many projects, change traffic is high. We know of projects where, in effect, the whole of the software is built to certification standards three times. In general, the rework is due to late discovery of requirements or design flaws.

Flaws and failure rates: From a safety perspective, we are concerned about the rate of occurrence of hazardous events or accidents during the life of the system. As indicated above, aerospace systems seem to achieve around 10^{-7} failures per hour. We are also aware of systems which have over 10^{7} hours of hazard free operation, although there have been losses of availability. However, there is no practical way of measuring such figures prior to putting the software into service ([7]) and, in practice, the best that can be measured pre-operationally is about 10^{-3} or 10^{-4} failures per hour.

As an alternative to evaluating failure rates, we can try to measure the flaws in programs. We define a flaw as a deviation from intent. We are not concerned about all flaw types, so it is usual to categorise them, e.g., safety critical (sufficient to cause a hazard), safety related (can only cause a hazard with another failure), and so on. On this basis, so far as we can obtain data, anything less than 1 flaw per kLoC is world class. The best we have encountered is 0.1 per kLoC for Shuttle code. Some observations are in order.

First, these figures are for known flaws; by definition, we do not know how many unknown flaws there are. We might expect all known flaws to be removed; however, removing flaws is an error prone process, and thus there comes a point where the risks of further change outweigh the benefits. Second, it is unclear how to relate flaw density to failure rate. There is evidence of a fairly strong correlation for some systems ([8]). In general, however, the correlation will depend on where the flaws are in the program. A system with a fairly high flaw density may have a low failure rate and vice versa depending on the distribution of flaws and demands (inputs). Third, commercial software has much higher flaw densities: perhaps 30-100 per kLoC; as much as two orders of magnitude higher!

We can also analyse where flaws are introduced and where they are removed to try to assess the effectiveness of processes. The available data suggest that more than 70% of the flaws found after unit testing are requirements errors. We have heard figures as high as 85% ([9]). Late discovery of requirements errors is a major source of change, and of cost.

Cost/effectiveness conclusions: The available data are insufficient to answer the most important questions, namely which approaches are most effective in terms of achieving safety (e.g., fewest safety-related flaws in the software) or most cost-effective. There are too many other factors (size of system, complexity, change density etc.) to make meaningful comparisons among the few data points available.

It is not even possible to provide an objective answer to the question "does safety critical software cost too much?" as software is typically used to implement functionality that would be infeasible in other technologies. Thus there are few cases in which we can directly compare the costs of achieving a given level of safety in software with the costs of achieving an equivalent level using other technologies.

Standards

It is impractical to summarise all relevant standards here. A brief critical survey of standards in the transport sector is presented in [10], and a far broader survey is given in [11].

Most software "standards" are not standards as other industries and engineering disciplines would recognise them. There are few software standards that present anything as concrete as the product-related design codes, rules, and objective tests which are found in structural, mechanical, and similar standards. Software standards are almost exclusively *process* specifications aimed at achievement of safety. There are few, if any, mandatory requirements or restrictions placed on the product itself.

It is also important to recognise that standards vary in their "ambition". There is an obvious contrast between sector-specific (e.g., rail, automotive, aerospace) standards and those that are – or attempt to be – sector independent (e.g., IEC61508 ([12])). There is also a fairly clear distinction between standards which capture current best practice, and those which make a deliberate attempt to promote technological advances by requiring processes and analyses which are beyond state-of-the-art.

Perhaps the most fundamental area in which standards vary is in the way in which they relate software integrity requirements to system level safety properties. Many of the standards categorise software in terms of its criticality to system safety. MilStd 882C ([13]) did this on the basis of software authority. A similar principle underlies the four development assurance levels (DALs) in DO178B ([14]). IEC 61508 uses the requisite level of risk reduction to allocate safety integrity levels (SILs). In the defence sector, DS 00-56 ([15]) allocates SILs based on hazard severity, with adjustments for probability. DefAust 5679 ([16]) has a more complex scheme, also introducing the notion of levels of trust. These standards also vary in whether or not they relate SIL or DAL to a failure rate. None of them make a link to flaw density. Queries have been raised about the soundness of the allocation processes in these standards (e.g., [17-19]).

Many standards identify processes to be followed at different SILs or DALs; however, there is a high level of variation among the standards. To oversimplify to make the point, standards such as DO178B say little about how software safety is achieved, and focus on the use of testing and human review to assure safety. In particular, DO178B requires evidence in the form of an accomplishment summary. Others define processes for achieving software safety. Standards such as DS 00-55 ([20]) and DefAust 5679 emphasise the use of formal methods, i.e., the use of mathematics to specify and analyse software. IEC 61508 recommends a large range of techniques, including formal methods.

None of the standards presents any evidence why we should be able to infer product integrity from a process, although many standards do include requirements for direct analysis of the software product. In general, no real distinction is made between achievement and assurance of safety or integrity, and there is an implicit assumption that following the process delivers the required integrity. More strongly, the use of a process becomes a self-fulfilling prophecy – "I need SIL 4, I've followed the SIL 4 process, so I've got SIL 4." As there is no definition of what the SILs mean independent of the process, this form of circularity is inescapable and we do not have the ability to reject ineffective standards.

Also many of the standards seem to be more "quality" standards than safety standards. The standards seem to identify which software engineering methods to apply, but don't require investigation of the ways in which software can contribute to hazards, or hazard control.

Industrial practises and experiences

Industrial practice is strongly influenced by standards, especially when the system needs to undergo formal certification; however, there are a number of factors that raise questions about the standards. These factors include a concern that the standards do not address the way industry actually works. For example, software standards invariably define activities for development of completely new systems, and generally ignore change and development of existing systems. There are also worries that what the standards legislate are not actually the most important issues in the production of safe software.

Experience shows that the allocation of SILs or DALs is problematic. The allocation is typically done at the system level, so the allocation rules apply to the whole computer "box". Whilst many standards allow partitioning of code within a box, and variation of SILs and/or DALs, it is rare for this to be done in practice. Normally, the whole system is developed to the level of the most critical function(s). This is a significant source of cost, especially if only a small proportion of the code is critical.

There are also problems simply in meeting the standards. For example, DO178B requires tests against the multiple-condition/decision (MC/DC) test coverage criterion With MC/DC a program containing a `statement IF A > 0 or B >0 Then,` would be tested `with A > 0`, `B > 0` (and both), and the "`THEN` part" could not simply be executed once. Meeting MC/DC is expensive and may constitute half the testing cost – but it finds relatively few problems. This result may mean that the criterion is inappropriate. However, it has been suggested that MC/DC is sometimes used in a "check box" manner, i.e., that the tests are run simply to meet the criterion, and that there is inadequate checking of the test results. Thus, it is hard to draw clear conclusions from the concerns with MC/DC testing.

The choice of programming languages or subsets to be used in developing critical software has long been a subject of debate. There is evidence that languages such as SPARK ([21]) help to eliminate certain classes of flaws; however, there are some systems written in assembler which have good safety records. Also, some C programs produced by code generators have low flaw densities. It is thus unclear what advice should be given in this area. Similar observations could be made about other aspects of software development, e.g. specification notations.

Perhaps more interestingly, there are a number of anecdotes which suggest that the development technology is not the key factor in achieving software safety. These anecdotes suggest that knowledge of the domain has a crucial bearing on safety (i.e., if you want safe software for controlling a landing gear, get it from people who understand landing gear). A second significant factor is the maturity of system design. Where one system is a close derivative of another, then it is likely to be effective. These observations should not be surprising – we identified requirements as the biggest source of problems for most systems. Employing people with good domain knowledge and evolving a successful product are both good ways of ameliorating requirements problems.

Finally, techniques for developing and assessing safety critical software are evolving. In particular, there is considerable interest in the use of code generators that produce executable code from higher-level design representations, e.g., state machines. These tools eliminate manual coding and hence reduce costs; however, the real saving would come from eliminating unit testing. This sort of technology change is hard to reconcile with current standards that are based on a more classical view of software development. Thus the current trends in software development pose difficulties in interpreting and applying standards, and may strengthen the view that particular techniques, e.g., MC/DC testing, are not cost-effective.

Lack of consensus

In examining the implications of the lack of consensus in standards and practices, we address two main questions: what are the sources of variation, and what are the problems that arise from the lack of consensus?

Sources of lack of consensus: The fundamental issue seems to be lack of data or knowledge. It is hard to obtain suitable data due to the commercial sensitivity of the data. Also any data we are able to get relates to old software technology because the actual failure rates of safety critical software systems are very low. Thus, if the writers of standards wish to be "up to date," they have to draw on what they perceive as being sound principles rather than building on knowledge of what has worked in the past. As a consequence, many of the current standards represent unverified research and not a consolidation of best practice. This result is probably the main source of the variation in the standards: no hard data exist to reject a theory about how to develop critical software, so each community develops its own theory. The fact that theories are unverified is a major problem, and one which cannot readily be overcome; thus, we seek to stimulate debate on this point.

A second set of issues is more "cultural." Diversity of approach to software development among industry sectors is an inevitable product of different historical backgrounds, and the different roles in which software has been deployed. For example, the control/protection system distinction typical in the process industries imposes quite different demands on software to the "single correct output" required of an aircraft flight control system. These differing requirements are (rightly) reflected in sector-specific standards. Unfortunately, the "sector-specifics" can also be seen in standards with ambitions to be sector-independent, e.g. IEC 61508. This can lead to considerable difficulty in applying such standards in sectors with which their authors were not familiar.

There are also significant international variations in industrial practices, which are also reflected in standards. The USA, at least in the military arena, has a "can do" attitude which is discernible in the relatively low level of prescription in standards and the willingness to try new ideas (e.g., the use of MatrixX code "out of the box" on the X38 project). The European military approach, on the other hand, tends to be more analytical and to seek to apply the relevant science to software, hence the concentration on formal methods and static analysis.

Conversely, in civil applications, the USA is often seen as being rather more conservative, waiting until new technology has been successfully demonstrated in Europe or other "high-tech" markets such as Japan before adopting it. To some extent, these differences are the result of different approaches to risk reduction. In the UK and in some other European countries, there is a legal requirement to reduce risk as low as reasonably practicable (ALARP). This requirement allows a designer to omit some possible risk reduction action because the costs are believed to be (grossly) disproportionate to the benefits. ALARP judgements in assuring safety are necessarily subjective, as we do not know what we will find unless we apply a given technique. There is no similar legal framework in the USA, to our knowledge; indeed it seems unacceptable to use economic judgements in risk reduction arguments in the USA.

Another significant "cultural" issue is how closely governments regulate different industries. Whilst some industries (e.g., nuclear) are tightly controlled in all countries, there is considerable variation in the treatment of other sectors, and there are frequently extensive and relatively rapid changes, often driven by the public reaction to accidents. This response can be seen, for example, in the changes in the UK government's position on rail safety regulation following recent highly-publicised accidents such as the Southall crash ([22]). Such responses clearly have direct implications for standards and practices in the industry concerned.

A further complication is the question of protecting national interests or, at a corporate level, intellectual property rights to processes and techniques that are perceived as giving significant competitive advantages. Whilst this situation is relatively rare in the field of safety, there are still a few examples (particularly of detailed analysis techniques) which have not gained international acceptance because of restricted publication. Fortunately, with safety issues, wide dissemination and discussion is generally seen as key to achieving broad-based acceptance of new approaches,

In the context of sector and international cultural diversity, some differences in the detail of requirements and processes are reasonable and justifiable. However the lack of fundamental agreement on what constitutes software safety, and how software safety fits in to the overall system safety process, is extremely worrying.

<u>Consequences of lack of consensus:</u> There are two key areas in which lack of consensus presents substantial problems for the safety

critical software industry. The first is in the technical and financial burdens that the lack of consensus imposes on industry and government. The absence of readily demonstrable equivalence between standards means that developers working on international projects often have difficulties when reconciling different standards. It can also be extremely difficult to take software developed in one industrial/national context and sell or re-use it in another. For example, working to DO178B means that the product does not meet the requirements of DS 00-55; this discrepancy raises the question of whether or not the software is "safe enough". The UK experience of applying static analysis to code developed to DO178B is that, in general, further flaws are found this way (e.g., for one system a flaw density of 23 per kLoC was found, although only 6% of these were critical). However, this analysis was done at great expense (around $50 per LoC) and it is unclear whether or not removing these flaws would significantly reduce the incident rate in service (although clearly it reduces risk).

For developers, the diversity in standards can limit the potential market for their products; for both vendors and customers, diversity presents significant costs in re-assessment or even re-working of products to the standards applicable in the new context.

There are also a number of less obvious costs, such as the need to train staff to work to different standards, and the cost of acquiring additional tools to support different processes.

The second area of concern is that of public perception. Although it has not yet become a major issue, at least in the UK, the challenge that "the industry can't agree within itself, so how are we to believe the industry is delivering safe products?" would be difficult to answer.

This discussion could perhaps be summarised by saying that software safety engineering is not a mature discipline. Further, due to cultural differences and the rate of change of technology, it is hard to see how to obtain data from system operation to prove the value of development and assessment techniques, and to show that the discipline is fully mature.

It has been suggested that aiming for consensus may, in itself, be dangerous, in that consensus would stifle the diversity of approach which leads to improvement. There is some merit in this view. However, we would argue that improvement of software safety standards through "survival of the fittest" cannot be expected, since organisations rarely have a choice of which standard(s) to apply on a given project. Also, project time frames are so long and accident rates are so low that there is insufficient data on which to base a rational selection between competing software processes.

Lessons from other technologies

All other technologies can suffer from systematic failures due to requirements or design problems as well as random failures. However, the possible effects of systematic as well as random failures are (at least in part) controlled by use of margins. For example, it is common practice to design structures to withstand more than the predicted maximum load. There is no obvious analogy for software, as there is no "continuous physics" of software which would enable us to measure the "strength" of a piece of code, or build something "stronger" than is required.

However, there are some aspects of the way in which the safety of other technologies is assessed from which we can derive lessons for software. The first is that, as safety is being assessed with respect to *intent*, not specification, systematic failures are identified and resolved as part of the process of safety analysis, e.g., a design flaw which gives rise to a hazardous state should show up in an FMEA or a fault tree. Second, the level of rigour and depth of analysis applied in assessing the design varies according to the criticality of individual components, rather than treating a whole system at one level. This makes the process more cost-effective, but it is dependent on the ability to assess components independently. Third, complex safety-critical components are commonly produced in an evolutionary manner, with one generation being an extension of the capability of an earlier one, e.g., increasing the capacity of an electrical generator by 10%. Fourth, critical components are subject to rigorous inspections and analysis to ensure that they are made of materials of adequate quality, have sufficiently low flaw density, and so on.

Principles of software safety

We indicated above the need to consider both achievement and assessment of safety. In fact we view a software safety process as having three main elements. First, it is necessary to identify software safety requirements, and we believe that there is broad consensus that the way to do this is by extending classical safety analysis down to the level of identifiable software failure modes. Second, we must consider how to achieve safety, with an emphasis on design to facilitate demonstration of safety, rather than prescriptive demands or constraints on the product or design process. Third, assurance must be provided that the software safety requirements have been met. This may employ classical software engineering techniques, but in a way in which the results can be linked into the system safety process.

Software safety requirements

Modern safety engineering processes, such as that in ARP 4761 ([23]) have the notion of "flowing down" (derived) safety requirements from the hazards to the level of components. Thus a hazard of "loss of ability to lower landing gear" with a required probability of 10^{-7} per flight hour would give rise to a number of lower level safety requirements, e.g., "loss of main hydraulics" of 10^{-4} per flight hour. This idea can be extended to software by setting requirements on specific software failure modes.

We cannot use code level failure modes, e.g., "taking a wrong branch" or "divide by zero" as they are not relevant at the level of software architecture where we wish to specify safety requirements. Instead, we suggest extending functional failure analysis (see ARP4761), to provide a form of "software HAZOP" ([24]). These techniques investigate the effects of failure types such as omission or late delivery of data items. There is now quite wide experience in applying these ideas to software ([25-27]), albeit mainly in the UK defence industry.

There may be concern about putting probabilities on software – but note that here we are placing a probability on occurrence of a failure event relating to the software, and we are not saying that we can evaluate directly a failure rate of, say, 10^{-7} per hour for software. In the above example, we might derive a software-related requirement that "inability to command lowering of landing gear" must have a probability of no worse than 10^{-5} per flight hour. Note that this might seem "obvious", but the examples in ARP 4761 still treat software at the "box level," and put failure probabilities of 0 in fault trees. There will also be other derived software safety requirements, e.g., for new functions to detect and mitigate hardware failures. It is also very important that such safety requirements are validated via simulation. However, this is largely a systems engineering issue, which is outside the scope of this paper.

Achieving software safety (economically)

As indicated above, experience suggests that domain knowledge and product maturity are more significant than development technology – some of the systems which have the best safety records are written in assembler from natural language specifications. Also technology changes very quickly so technology prescriptions or proscriptions will rapidly become obsolescent. However we can identify some principles of which will certainly aid assessment of safety and, we believe, aid achievement.

First, it is desirable to partition the software into components of different criticalities within one processor. Standards such as DO178B allow for this, but partitioning is rarely seen in practice. Our experience has shown that it is possible to verify partitioning even for complex multi-processor computers ([28]), and the effort was relatively small (about 2 man years for safety analysis and testing) in comparison with the economic gain. If a processor contained 100 kLoC, and as a result of partitioning half could be treated as non safety-critical, this could save several man-decades of effort. Proper hardware design is needed to support partitioning. The capabilities required are not out of the ordinary.

Second, the software needs to be designed so as to minimise the cost of assessment. In general, the more detailed the level at which assessment is undertaken, e.g., object code rather than source, the greater the cost. An ideal design representation would be sufficiently complete and precise to enable analysis to be done at the design level. Again this sounds obvious, but few design representations have this property, e.g., it is often not possible to determine schedulable units of software from the design descriptions. This characteristic can be achieved by restricting some of the widely used notations such as Statecharts ([29]) and control law diagrams.

Thus an ideal design representation has to be complete (i.e., cover all feasible combinations of inputs), deterministic, fully describe the mapping to hardware, and so on. Partial solutions include the use of well-defined subsets of notations ([30]), and the use of model checking to validate the design. Such a design representation should also support hazard analysis using methods such as SHARD or Deviation Analysis to identify failure modes of components. This would enable the safety requirements can be mapped down to the software design, i.e., so that the system level fault trees contain failure modes of software components as base events. For this to be fully effective, it is important that the mapping from design notation to eventual implementation is completely understood and controlled.

There is also value in extending the design representation to include information on failures and failure propagation, then generating the relevant parts of the system fault tree from the design description. Early work to show the feasibility of such an approach is reported in [10], and more recent work has shown how fault trees can be generated from enhanced Matlab/ Simulink models of control functions.

Third, the use of family analysis ([31]) gives a systematic way of identifying the evolution of a system from one generation to the next, and thus enabling a systematic reuse programme. This in turn facilitates the growth of design maturity, in a manner similar to that in other engineering disciplines. Our application of these ideas to safety critical software have been successful in terms of reducing the costs of the development process ([32]). However we have yet to be able to demonstrate that they improve safety, or that it is possible to link reusable verification evidence to the components in the family reuse library.

Fourth, we need some “guarantees” that the implemented program will execute without exceptions, i.e., it will not give rise to memory violations, divide by zero, violate the dynamic semantics of the language, and so on. We refer to this as program integrity. Integrity can be achieved, at least in part, by construction, but may also require assessment, see below. Program integrity is a necessary pre-requisite to be able to assess safety at the level of the software design. This is an area where static analysis and focused testing can be effective.

Assuring safety

Assuring safety requires evidence that all the safety requirements, on the design components, have been met. However, showing that the requirements are met is complicated because of the dependencies between components, the fact that they share computing infrastructure, and the issue of the required level of rigour. We deal with the issue of rigour first.

In our view, the notion of SILs is too complex, and matches too poorly with standard safety engineering practices, e.g. fault trees, to be useful. Instead we say that some failure event (associated with a software safety requirement) is either safety critical or safety related. It is safety critical if it appears in a system fault tree cut set of size one, leading to a hazard which endangers life, and it is safety related if it is in a cut set of size two or more. We require diverse evidence, e.g., from testing and static analysis, for safety critical events, but will accept single sources of safety evidence for safety related events. Note that here we are talking about *events*; a single software component may have the capacity to cause more then one safety critical or safety related failure event.

This approach is novel but we think simple and justifiable. We also believe that, in general, it will lead to a reduction in costs.

We now need to discuss the types of evidence to be collected. We note that a software component can fail because of an internal problem or due to the behaviour of other components. In fault tree terms, the failure of the component itself is a primary failure; that of a component which supplies data is a secondary failure. A failure of the infrastructure, e.g., scheduling, will be viewed as a command failure. Thus, we can use fault tree concepts in identifying the requisite safety evidence. We note that we need not concern ourselves with the effects of components which are not causally linked to the primary component. Partitioning and program integrity will ensure that there is no “interference.”

The types of evidence will vary with the types of failure modes. For example, absence of omission in the primary component might be shown by means of inspection of the program control flow graph or by static analysis. On the other hand, absence of late service provision might be shown by static worst case execution time (WCET) analysis or by timing testing. The generation of evidence of this nature can be seen as verifying the specification of failure propagation at the design level. Choice of the form of evidence to

employ is an area where explicit consensus is needed – and we return to this point below.
In principle evidence can be gathered by working back from a software safety requirement, through the design to the inputs from outside the computer. If there are no critical "bugs" (found) in the software, the analysis will show which input failure(s) will cause the program to violate the safety requirement. If there is a critical "bug" then the analysis will show the circumstances, e.g., combination of normal inputs and program state, which will lead to the software safety requirements being violated.

In practice, if this approach were followed, each component would be assessed many times for evidence about its contribution to each hazard. To make the process more efficient, the evidence requirements should be evaluated as outlined above, then consolidated to provide a software safety verification plan.

The approach we have outlined seeks to gather direct, product-based evidence, which we feel is more compelling than process based evidence on its own. However, process based evidence is still needed, e.g., to show that the configuration of the system as analysed is the same as that which has been deployed.

Observations

The concepts we have outlined are, we believe, relatively simple and compelling. However it is difficult to convert them into guidelines for practitioners, or standards – see below.

The approach we have outlined is rather at variance with modern standards – but perhaps not with older ones! We believe that it is very similar to the way in which MilStd 882C was used on some projects. Also we are aware of projects in military aerospace that have used such approaches. Finally it is really just common sense – we are simply suggesting continuing the safety analysis process down into the software, and using software engineering techniques to provide evidence relating to base events in the fault trees, much as we would use mechanical engineering data for mechanical components.
Some work is under way to put these principles into practice. In the UK work for the Civil Aviation Authority (CAA) has produced a draft standard for use on ground based systems, based on the principles outlined above ([33]). Also a project with the European Space Agency (ESA) is developing similar guidelines for assessing space-borne software ([34]). The ESA work is intended to be evaluated experimentally before being applied in practice, but the CAA standard is being applied to current systems.

Also, our approach towards achievement of safety bears strong similarity to the ideas in SpecTRM developed by Nancy Leveson ([35]). Indeed Leveson's approach could be seen as a way of meeting some of the requirements of the approach we have outlined.

Towards international consensus

We have tried to illuminate some of the reasons why there is no consensus on software safety – so why should we think that our proposals might form the basis of international consensus? As indicated above, we think that what we have proposed is "common sense," is compatible with some existing standards, and links well with system safety practises – but that is not enough!

There is no way that these ideas will get adopted and imposed "top down" – there are simply too many international standards bodies, and the process takes too long. Instead we would hope to see a "bottom up" acceptance.

It is planned that the ESA work discussed earlier will culminate in some experiments to apply the approach outlined on some representative space software, with a measurement programme to help assess effectiveness and cost-effectiveness. If the approach is seen to be cost-effective, then this will generate interest.

We would also hope to encourage adoption where it is already practical. We believe that the ideas are fully compatible with the aerospace safety standard ARP 4761, and that there is enough flexibility in what can be presented in a DO 178B accomplishment summary, that the ideas could be used in this context without difficulty. Also, in the UK, the MoD is engaged in redeveloping the defence safety standards, so it is possible that these ideas might be adopted within that framework. If there can be "local" successes such as this we believe that there is sufficient interest in revising the way in which software safety is assessed that it will be possible to get wider take-up of the ideas.

However there are still some technical issues to resolve, primarily what is viewed as acceptable evidence for each class of failure mode. We believe that it may be easier to get consensus on such issues as we are dealing with narrower issues, e.g., ways of showing the absence of omission failures, rather than getting agreement to a "whole software process." Nonetheless there are some difficulties here, and there is no obvious forum for discussing the issues and gaining consensus.

Conclusions

We have identified some of the problems and costs associated with demonstrating software safety, including outlining the variances between some standards and industrial practises. We have used an analysis of some of the perceived problems to outline a proposed "new" set of principles to apply to software.

We hope that these principles can gain wider acceptance, as they are based on the hopefully uncontentious idea of applying classical safety engineering ideas more fully to software safety assessment. There are many issues we have been unable to address within the paper, some of which have been considered in other, more extensive works, e.g., [33, 34].

As indicated in the introduction, we are not claiming to have a complete solution to the issues of how to show software safety cost-effectively, but we hope that the ideas we have presented will stimulate debate, and help towards achieving greater international consensus.

Acknowledgements

Our thanks to our colleagues at York for their contributions to the work presented here. Thanks also to our industrial partners for funding, and for unique insights into the "real world" of industrial, safety-critical software engineering. We must, however, emphasise that this paper represents our own views, which are not necessarily those of our partners! Also, Susan Donohue provided some useful comments on an earlier draft of the paper.

References

1. Lions, J.L., *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. 1996, Paris: European Space Agency.

2. Leveson, N.G., *Safeware: System Safety and Computers*. 1995, Reading, Mass.: Addison Wesley. 680.

3. Aeronautica Civil of the Republic of Colombia, *Controlled Flight into Terrain, American Airlines Flight 965, Boeing 757-223, N651AA, near Cali, Colombia, December 20 1995*. 1996, Santafe de Bogota.

4. MacKenzie, D., *Computer-Related Accidental Death: An Empirical Exploration*. Science and Public Policy, 1994. **21**(4): p. 233-248.

5. Shooman, M.L., *Avionics Software Problem Occurrence Rates*. ?, 1996. **?**(?): p. 53-64.

6. Air Accidents Investigation Branch, *Accident to Chinook HC2 Registration ZD576 at Mull of Kintyre, Scotland, on 2 June 1994*, . 1995: Farnborough, Hants. p. 66.

7. Butler, R.W. and G.B. Finelli. *The Infeasibility of Experimental Quantification of Life-Critical Software Reliability*. in *ACM SIGSOFT '91 Conference on Software for Critical Systems*. 1991. New Orelans, Louisiana: ACM Press.

8. Bishop, P.G. and R.E. Bloomfield. *A Conservative Theory for Long-term Reliability Growth Prediction*. in *Proc. Seventh International Symposium on Software Reliability Engineering*. 1996. White Plains, NY: IEEE.

9. F22 Program Office, . 1998.

10. Papadopoulos, Y. and J. McDermid. *A New Method for Safety Analysis and Mechanical Synthesis of Fault Trees in Complex Systems*. in *12th International Conference on Software and Systems Engineering and their Applications*. 1999.

11. Hermann, D., *Software Safety and Reliability*. 1999: IEEE Computer Society Press.

12. International Electrotechnical Commission, *IEC 61508: Fundamental Safety of Electrical / Electronic / Programmable Electronic Safety Related Systems*. 1999.

13. US Department of Defense, *Military Standard 882C (Change Notice 1): System Safety Program Requirements*. 1996.

14. Radio Technical Commission for Aeronautics, *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. 1992.

15. UK Ministry of Defence, *Defence Standard 00-56 Issue 2: Safety Management Requirements for Defence Systems*. 1996.

16. Australian Department of Defence, *Australian Defence Standard Def(Aust) 5679:*

Procurement of Computer-based Safety Critical Systems. 1998.
17. Lindsay, P.A. and J. McDermid. *A Systematic Approach to Software Safety Integrity Levels*. in *Proceedings of the 16th International Conference on Computer Safety*. 1997. Berlin: Springer.
18. Fowler, D. *Application of IEC61508 to Air Traffic Management and Similar Complex Critical Systems - Methods and Mythology*. in *Lessons in System Safety: Proceedings of the Eighth Safety-critical Systems Symposium*. 2000. Southampton, UK.
19. Redmill, F. *Safety Integrity Levels - Theory and Problems*. in *Lessons in System Safety: Proceedings of the Eighth Safety-critical Systems Symposium*. 2000. Southampton, UK: Springer.
20. UK Ministry of Defence, *Defence Standard 00-55: Requirements for Safety Related Software in Defence Equipment*. 1996.
21. Barnes, J., *High Integrity Ada - The SPARK Approach*. 1997: Addison-Wesley.
22. Uff, J., *The Southall Rail Accident Inquiry Report*. 2000: HSE Books.
23. SAE, *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. 1996, Warrendale, PA: Society of Automotive Engineers, Inc. 331.
24. Kletz, T., *Hazop and Hazan: Identifying and Assessing Process Industry Hazards*. Third ed. 1992: Institution of Chemical Engineers.
25. Redmill, F., M. Chudleigh, and J. Catmur, *System Safety: HAZOP and Software HAZOP*. 1999, Chichester: John Wiley & Sons Ltd. 248.
26. McDermid, J.A. and D.J. Pumfrey. *A Development of Hazard Analysis to aid Software Design*. in *COMPASS '94: Proceedings of the Ninth Annual Conference on Computer Assurance*. 1994. NIST Gaithersburg MD: IEEE, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 0855-1331.
27. McDermid, J.A., *et al.*, *Experience with the Application of HAZOP to Computer-Based Systems*. COMPASS '95: Proceedings of the Tenth Annual Conference on Computer Assurance, 1995: p. 37-48.
28. McDermid, J.A. and D.J. Pumfrey. *Safety Analysis of Hardware / Software Interactions in Complex Systems*. in *Proceedings of the 16th International System Safety Conference*. 1998. Seattle, Washington: System Safety Society, P.O. Box 70, Unionville, VA 22567-0070.
29. Harel, D., *StateCharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 1987. **8**(3): p. 231-276.
30. Burton, S., *et al. Automated Verification and Validation for High Integrity Systems: A Targeted Formal Methods Approach*. in *NASA Langley Formal Methods Workshop*. Langley, USA: NASA.
31. Coplien, J., D. Hoffman, and D. Weiss, *Commonality and Variability in Software Engineering*. IEEE Software, 1998. **15**(6): p. 37-45.
32. Stephenson, A., D.L. Buttle, and J.A. McDermid. *Extending Commonality Analysis for Embedded Control System Families*. in *Third International Workshop on Software Architectures for Product Families*. 2000: LNCS 1951.
33. Civil Aviation Authority, *CAP670: Air Traffic Services Safety Requirements*. 1998.
34. McDermid, J.A., S.C.B. Wills, and G.K. Henry, *Combining Various Approaches for the Development of Critical Software in Space Systems*, . 2000, European Space Agency.
35. Leveson, N., *Safeware Engineering Corporation - SpecTRM,* http://www.safeware-eng.com/.

Biographies

John McDermid, Department of Computer Science, University of York, Heslington, York, YO10 5DD, England. Tel: +44 1904 432726; Fax: +44 1904 432708
Email: john.mcdermid@cs.york.ac.uk

John McDermid is Professor of Software Engineering at the University of York. He directs the High Integrity Systems Engineering group at the University which is a recognised research centre in safety critical computing for the UK aerospace industry. He directs the Rolls-Royce University Technology Centre in Systems and Software Engineering and the BAE SYSTEMS Dependable Computing Systems Centre. He has published 6 books and over 250 papers

David Pumfrey, Department of Computer Science, University of York, Heslington, York, YO10 5DD, England. Tel: +44 1904 432735; Fax: +44 1904 432708
Email: david.pumfrey@cs.york.ac.uk

David Pumfrey is a research and teaching fellow in the DCSC, with research interests in HAZOP, safety analysis of software/hardware interactions, and the development of improved system safety processes. Together with John McDermid he is also one of the presenters of a highly successful series of short courses on system safety and safety cases.