Software safety: relating software assurance and software integrity

Ibrahim Habli*, Richard Hawkins and Tim Kelly

High Integrity Systems Engineering Research Group,
Department of Computer Science,
The University of York,
York, YO10 5DD, UK
Fax: +44 (0) 1904 432708
E-mail: Ibrahim.Habli@cs.york.ac.uk
E-mail: Richard.Hawkins@cs.york.ac.uk
E-mail: Tim.Kelly@cs.york.ac.uk
*Corresponding author

Abstract: The overall safety integrity of a safety critical system, comprising both software and hardware, is typically specified quantitatively, e.g., in terms of failure rates. However, for software, it is widely accepted that there is a limit on what can be quantitatively demonstrated, e.g., by means of statistical testing and operational experience. To address this limitation, many software standards appeal instead to the quality of the process to assure the sufficient implementation of the software. In this paper, we contend that there is a large inductive gap between the quantitative software integrity required for a safety function and the assurance of the software development process for that function. We propose that this large inductive gap between software integrity and software process assurance could be narrowed down by an explicit definition of a product-based software argument. The role of this argument is to justify the transition from arguing about software integrity to arguing about software assurance by showing how the evidence, in the context of the software product-based argument, provides assurance which is commensurate with the required integrity.

Keywords: software safety; software reliability; safety critical systems; safety critical software; software safety standards; software assurance; software integrity; software quantification; safety cases; safety arguments; goal structuring notation.

Reference to this paper should be made as follows: Habli, I., Hawkins, R. and Kelly, T. (2010) 'Software safety: relating software assurance and software integrity', *Int. J. Critical Computer-Based Systems*, Vol. 1, No. 4, pp.364–383.

Biographical notes: Ibrahim Habli is a Research and Teaching Fellow at the High Integrity Systems Engineering (HISE) Research Group at the University of York. He has previously worked as a Research Associate at the Rolls-Royce University Technology Centre at the University of York. His main research interests include safety case development, software safety assurance, software architecture design and safety-critical product-lines.

Richard Hawkins is a Research Associate for the Software Systems Engineering Initiative (SSEI) at the University of York. He is a member of the High Integrity Systems Engineering (HISE) Research Group. His current research involves establishing a systematic approach to the development of software safety cases. In particular, he is developing guidance on the construction of software safety arguments and developing an approach for reasoning about argument and evidence assurance.

Tim Kelly is a Senior Lecturer in Software and Safety Engineering within the Department of Computer Science at the University of York. His expertise lies predominantly in the areas of safety case development and management. He has published over 70 papers on safety case development in international journals and conferences and has been an Invited Panel Speaker on software safety issues.

1 Introduction

For software systems, it is widely accepted that there is a limit on what can be quantitatively demonstrated, e.g., by means of statistical testing and operational experience. However, to meet regulatory requirements, system safety engineers often have to describe the safety integrity of the overall system quantitatively, e.g., in terms of system failure rates. This is mostly the case even when the system embodies and depends on software components to perform safety-critical operations. To this end, it is difficult to exclude software components from the allocation of quantitative safety integrity requirements, i.e., merely because of the preserved limitation of demonstrating the achievement of these requirements by means of quantitative evidence. Many safety standards acknowledge the difficulty of demonstrating, quantitatively, low software failure rates and instead appeal to process-based arguments, which are mostly based on qualitative software assessment techniques.

In this paper, we discuss the limitations of demonstrating the satisfaction of safety integrity requirements allocated to software by means of quantitative evidence. We also discuss the weaknesses of addressing these limitations by directly appealing to process assurance, through compliance with the process defined in prescriptive software standards. We contend that there is a large inductive gap between the quantitative software integrity required for a safety function and the assurance of the software development process for that function, i.e., good tools, techniques and methods do not necessarily lead to the achievement of the required integrity (e.g., in the form of failure rates). We propose that this large inductive gap between software integrity and software process assurance could be narrowed down by an explicit definition of a product-based software argument. The role of this argument is to justify the transition from arguing about software integrity to arguing about software assurance by showing how the evidence, in the context of the software product argument, provides assurance which is commensurate with the required integrity. By openly recognising and reasoning about, this inductive gap, the sufficiency of the evidence, produced by the process through testing and analysis, can be more easily determined and justified. Using a consideration of product assurance to drive the development processes provides advantages over a prescriptive approach, which results in evidence which may or may not be appropriate and relevant to the claims concerning safety functions required, and may or may not be of the required assurance.

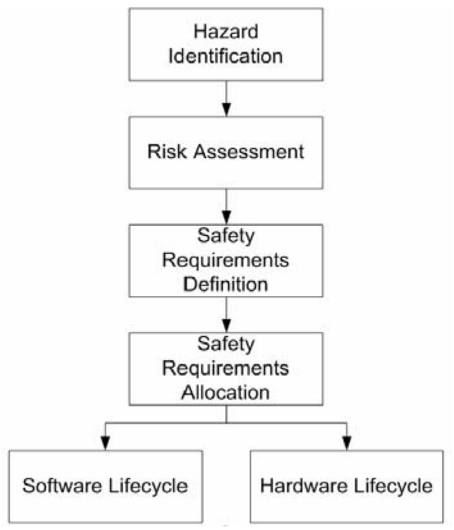
The rest of the paper is organised as follows. Section 2 presents an overview of system safety activities that lead to the definition of quantitative software safety integrity requirements. The limitations of demonstrating the achievement of these requirements by quantitative software evidence is discussed in Section 3, followed by discussing the weaknesses of appealing instead to the quality of the software process (Section 4). Relating safety integrity to assurance, in the context of an explicit product-based software safety argument is then presented in Sections 5, 6 and 7, followed by describing the relationship between software product assurance and software process assurance (Section 8). Finally, conclusions are discussed in Section 9.

2 Definition of safety and integrity requirements

In this section, we briefly explore the safety lifecycle activities which typically lead to the generation and allocation of safety requirements. These activities include (Figure 1):

- hazard identification
- risk assessment
- definition of safety requirements
- allocation of safety requirements to system components.

Figure 1 Allocation of safety requirements



Before the hazard identification activity can commence, the system and its intended environment should be described and understood. In many domains, the description of the system is specified in terms of the physical and operational environment and the system's boundary, functions and inter-functional dependencies (SAE, 1994) (IEC, 1998). Hazards associated with the system and its environment are then identified and typically recorded in a hazard log. The hazard log is a live artefact which tracks the management of hazards throughout the system, hardware and software safety lifecycles.

For each hazard recorded in the hazard log, the likelihood and severity of the hazard's consequences are then determined and the risk associated with the hazard is estimated accordingly. Hazard identification and risk assessment are iterative activities. The understanding of hazards and their estimated risks evolves in parallel to the evolution of the system development and deployment, and later evolves based on data gathered from the system's operation.

Safety requirements are then defined, specifying the required risk reduction associated with each identified hazard. Despite variation in terminology, there are generally three types of safety requirements (MoD, 2007):

- *safety requirements* allocated to the system as a whole
- *derived safety requirements* generated from decisions made during development phases, e.g., design and implementation decisions
- *safety integrity requirements* specifying the failure rates with which the safety requirements and the derived safety requirements should be achieved.

Most standards require a quantitative approach to defining safety integrity requirements. For example, the UK Defence Standard 00-56 states that "quantitative safety integrity requirements should be defined for safety related complex electronic elements" (MoD, 2007) – the higher the importance of the safety requirements to system safety, the more stringent the quantitative safety integrity requirements are. There are different ways in which quantitative safety integrity requirements can be specified. For example, they can be specified in terms of mean-time-to-failure, probability of failure-free operation or unavailability time (Littlewood and Strigini, 1993). The generic functional safety standard IEC 61508 (IEC, 1998), for instance, emphasises the distinction between two categories of failure rates: probability of failure to perform low demand functions and probability of failure to perform high demand/continuous functions. Each range of failure rates for these two categories is associated with one of four safety integrity levels (SILs) as shown in Table 1.

Table 1 SILs in IEC 61508

Integrity level	*Continuous mode probability of a dangerous failure per hour*	*On demand mode probability of failure to perform the design function*
4	$10^{-9} < P \leq 10^{-8}$	$10^{-4} < P \leq 10^{-5}$
3	$10^{-8} < P \leq 10^{-7}$	$10^{-3} < P \leq 10^{-4}$
2	$10^{-7} < P \leq 10^{-6}$	$10^{-2} < P \leq 10^{-3}$
1	$10^{-6} < P \leq 10^{-5}$	$10^{-1} < P \leq 10^{-2}$

Source: IEC (1998)

Safety requirements and their associated safety integrity requirements are then allocated to system components within the overall system architecture. If a system component comprises hardware and software components, the safety requirements and their associated safety integrity requirements are refined and allocated to these components. For software and hardware components which implement more than one safety requirement of different safety integrity targets, these elements should be developed to the highest allocated safety integrity target, unless sufficient separation and partitioning is assured.

3 Satisfying safety integrity requirements

At the hardware level, there are accepted means for demonstrating the achievement of the quantitative safety integrity requirements allocated to hardware components. This is mainly because hardware components fail randomly, as a result of physical factors such as corrosion and wear-out. To this end, statistical-based reliability techniques can be applied, supported by operational experience and industry databases, to estimate and demonstrate, prior to deployment, the failure rates of the hardware components.

At the software level, on the other hand, and because software is an abstraction, software failures are systematic – mainly due to specification and design faults. Because software is not physical, there is a limit on what can be quantitatively demonstrated by means of statistical testing and operational experience. Littlewood and Strigini (1993) list the following as the main factors that distinguish software reliability from hardware reliability:

- software failures are caused by design faults which are difficult to avoid
- software is usually used to implement relatively new systems, making it difficult to exploit knowledge from previous experiences
- software systems implement 'discontinuous input-to-output mappings' which are complex to be captured in simple mathematical models.

In the safety domain, statistical testing can generally demonstrate software failure rates of 10^{-3} to 10^{-4} per hour, prior to release to service (McDermid and Kelly, 2006). However, this is far from sufficient in domains such as civil aerospace, where DAL A software, whose failure could contribute to catastrophic failures, corresponds to a failure rate of 10^{-9} per flying hour. Despite case studies suggesting the achievement of failure rates of less than 10^{-7} per hour (Shooman, 1996), based on software operational data, there is a large consensus that it is infeasible to demonstrate such low failure rates prior to deployment, where demonstrating safety integrity is most needed, e.g., to obtain the approval of certification authorities.

Butler and Finelli (1991) in their landmark paper argue that it is infeasible to quantify the reliability of life-critical software systems, regardless of the form of verification, whether it is a black-box or white-box examination. For example, they estimate that the test duration for quantifying a software failure rate of less than 10^{-9} per hour would take 114,155 years. Littlewood and Strigini (1993) discuss three issues related to software reliability, namely:

- specifying reliability targets
- designing software systems which can achieve these reliability targets
- evaluating the achievement of the reliability targets by the software systems.

While allocating reliability targets in the form of safety integrity requirements is adequately addressed in many safety assessment processes, e.g., (SAE, 1994) (IEC, 1998), and while there are established design mechanisms to address these integrity requirements, the fundamental limitation lies in evaluating the achievement of the reliability targets in a quantitative manner. In terms of safety argumentation, we know how to make quantitative claims regarding safety integrity requirements. We also know how to make claims concerning the suitability of the chosen design measures. However, we often fail to produce evidence, based on quantitative techniques, which could substantiate the safety integrity requirements and design claims.

To meet various national and international regulatory requirements, system safety engineers have to describe the safety integrity of the overall system quantitatively. This is mostly the case even when the system embodies, and depends on, software components to perform safety-critical operations. To this end, it is difficult to exclude software components from the allocation of quantitative safety integrity requirements, i.e., merely because of the preserved limitation of demonstrating the achievement of these requirements by means of quantitative evidence. Many safety standards acknowledge the difficulty of demonstrating, quantitatively, low software failure rates and instead appeal to process-based arguments, which are mostly based on qualitative software assessment techniques. In the next section, we discuss fundamental problems relating the achievement of quantitative safety integrity requirements to qualitative process-based evidence. In particular, we focus on the way in which appealing to process-based evidence has been addressed in software certification standards.

4 Problems of existing approaches to satisfying safety integrity requirements for software

The limitations of demonstrating quantitative integrity requirements for software have long been acknowledged in most software certification standards. Existing certification approaches, such as those defined in the commonly used standards such as IEC 61508 (IEC, 1998) and DO178B (RTCA, 1992), do not set out to demonstrate the achievement of qualitative targets for software. For example, DO178B explicitly states that, "development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software levels or software reliability rates based on software levels cannot be used by the system safety assessment process as can hardware failure rates."

Instead, such standards adopt a highly prescriptive approach, based around a demonstration of compliance with a defined process. The process in DO178B is defined as a set of objectives. The number of objectives which needs to be met and the level of independence with which they must achieved is determined by the allocated software level. For DO178B, the component which implements a safety function is allocated a software level based upon an assessment of the safety impact of the failure of that function.

The IEC 61508 standard allocates a SIL to the system implementing the safety requirement. The SIL represents the required integrity as one of four discrete levels. The SIL is determined based upon the responsibility of that function for risk reduction at the system level. In a similar manner to that described above for DO178B, the rigour of the process followed in developing the software varies according to the assurance level associated with that software.

So, although it is often difficult to directly claim that the safety functions of the system are implemented to the required integrity by the software, complying with prescriptive software standards such as those discussed above enables an alternative claim to instead be made. That claim is that having demonstrated compliance with the standard we are sufficiently assured that the safety function is implemented. This reasoning underpins most highly prescriptive standards, and is the basis upon which they are accepted. It is important at this point to consider carefully both the fact that we are no longer dealing with demonstrating integrity, but with demonstrating confidence, and also how we can be sure that achieved confidence is sufficient.

When we consider integrity, we are dealing with aleatoric uncertainty. By changing to a consideration of confidence, we are dealing with a different type of uncertainty, epistemic uncertainty. Epistemic uncertainty is characterised by the limitations of knowledge. To be able to use an approach to software safety which is based upon confidence, 'it is necessary to justify that the level of epistemic uncertainty demonstrated in the safety function is commensurate with the aleatoric uncertainty which was determined for that function'.

When considering safety, it is common to use the term assurance rather than confidence. Assurance is simply the level of confidence which can be justified. In this paper, we contend that it is possible to justify the achievement of an integrity requirement through a consideration of assurance. However, prescriptive approaches, such as those discussed above, are insufficient to justify that the integrity requirement is sufficiently addressed. Firstly it should be noted that a prescriptive approach deals with process assurance; confidence in the rigour of the process followed. The first challenge here is whether a 'good' process necessarily leads to a 'good' product. A number of people have questioned this correlation. In relation to IEC 61508, Redmill (2000) notes that, "the processes defined as being appropriate to the various SILs are the result of value judgements regarding what needs to be done in support of a reasonable claim to have met a particular SIL. However, the development processes used, however good, appropriate, and carefully adhered to, do not necessarily lead to the achievement of the defined SIL." This position is supported by McDermid (2001) who, whilst acknowledging that his assessment cannot be taken as conclusive, claims that, "the evidence does not support the assumptions that the processes for the higher SILs/DALs produce software with lower failure rates. At minimum the assumption seems questionable."

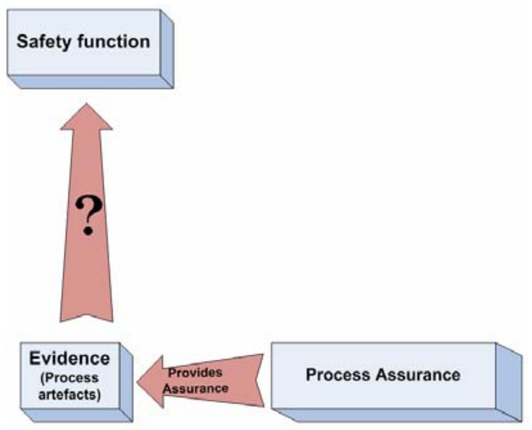
This suggests that, to be compelling, rather than process assurance, what is really needed is product assurance; confidence in the behaviour of the software product itself. Product assurance can be more easily directly related to product integrity. In fact, despite some of the pessimistic views expressed above, highly prescriptive approaches do have the capability of providing some information relating to product assurance. Following a rigorous process may not guarantee software integrity, but it does provide assurance relating to the quality of, and trustworthiness in, the outputs from that process (e.g., test results or mathematical proofs).

5 Relating software integrity to software assurance

To be able to demonstrate elements relating to the integrity requirements associated with a safety function, it is necessary to generate assurance in that safety function. Although assurance in a safety function and integrity of a safety function are not equivalent measures, we will show that it is possible to demonstrate the sufficiency of such a comparison.

In the previous section we discussed how a prescriptive approach can provide assurance in the process artefacts. The problem with this is that the assurance of the process artefacts does not necessarily provide the assurance that is required in the safety functions. The process artefacts generated from following the process do not necessarily demonstrate that the safety functions are achieved to the required integrity. This is illustrated in Figure 2. The diagram indicates that the process assurance activities can demonstrate assurance of the quality of the process artefacts. There is a missing argument here which is needed to justify how the evidence produced by the process can provide a compelling demonstration of the assurance of the safety function, as indicated by the question mark in Figure 2.

Figure 2 The role of process assurance (see online version for colours)



It is necessary, rather than relying on the outputs of a prescribed process, to instead explicitly consider the type of evidence that could be used to demonstrate that the specific safety functions required of the software have been achieved. A structured, product-based safety argument provides a way in which this can be demonstrated.

6 Constructing product-based software safety arguments

A product-based software safety argument demonstrates how the available evidence can be reasonably interpreted as indicating assurance in achieving allocated safety requirements. The degree of this assurance should be commensurate with the required integrity. By generating an explicit product-based software safety argument, the way in which the evidence supports the safety integrity claims made in the software safety case for the particular system under consideration becomes clear.

Before we elaborate further on the key role of the product-based software safety argument, it is important to highlight the distinction between two types of argument: deductive and inductive arguments. Baggini and Fosl (2003) characterise deductive arguments as those where, if the premises are true, then the conclusion must also be true. In contrast, an inductive argument is characterised as one where the conclusion follows from the premises not with necessity, but only with probability. Safety arguments are rarely provable deductive arguments. Instead they are more commonly inductive due to the complexity and high-level of uncertainty in the software specification and design.

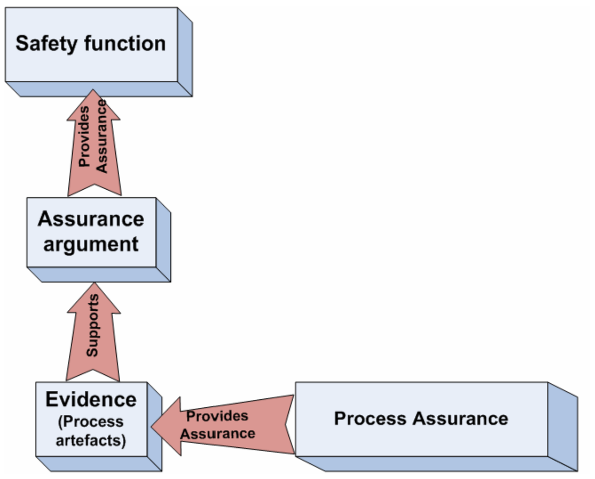
Regarding the term assurance in the context of a safety argument, it is used to refer to the justified confidence that a claim in the safety argument is true. The assurance of a safety claim is related to:

- the assurance of the premises (supporting claims or evidence)
- the extent to which those premises give reason to believe the conclusion is true.

Due to the inductive nature of most safety arguments, determination of assurance is always going to be subjective. What is important is that agreement can be reached between the safety argument provider, and the safety argument reader, that the subjective position is acceptable. This can be achieved by demonstrating that the argument is sufficient. We have already seen how this subjectivity is also present in a prescriptive approach. Figure 2 is essentially indicating a large inductive ‘gap’ between the assurance in the process artefacts and the assurance of the safety function. By reasoning about the subjectivity explicitly in a safety argument, the sufficiency of the evidence can be more easily determined and justified.

Figure 3 shows how a product-based safety argument can be used to provide the required link between the evidence and the safety function, thus demonstrating assurance not just in the process, but also directly in the product (i.e., the safety function of interest). Figure 3 indicates how the assurance of the evidence used to support the product-based safety argument can still be provided by process assurance in a similar manner to that discussed for a prescriptive approach earlier. This is discussed in more detail in the last section of this paper.

Figure 3 The role of an assurance argument (revisited) (see online version for colours)



The approach described above provides a way, through the creation of a product-based software safety argument, of determining the evidence that is required. It is then possible to ensure that processes are put in place to generate that evidence with the required assurance. Using a consideration of product assurance to drive the development processes provides advantages over a prescriptive approach, which results in evidence which may or may not be appropriate and relevant to the claims concerning safety functions required, and may or may not be of the required assurance.

We have so far discussed how an explicit demonstration of the assurance of safety functions can be achieved through a focus on product assurance. We have yet to fully discuss the relationship between the assurance of safety functions and the demonstration of integrity of safety functions (the achievement of safety requirements). This is the subject of the next section.

7 The transition from software product integrity to software product assurance

It is important in constructing the software safety argument to correctly capture the claims which need to be supported. The top level claims in the argument should reflect the overall safety objectives. For a software safety argument, these objectives are that the software safety requirements are valid, traceable and satisfied. This includes demonstrating both the functional and associated integrity requirements. We discussed earlier how directly demonstrating integrity for software is very difficult, however there are commonly adopted strategies which can have a direct impact on the failure rates of software product. For example, there are a number of software architectural approaches which can directly reduce the expected failure rate due to the presence of redundancy. Such software architectures, when implemented with sufficient diversity, can reduce the probability of the manifestation of a functional failure. It is possible to include such architectural mitigations within the argument to directly address a claim relating to integrity. Similarly, the use of other features such as the implementation of exception handling, can directly affect the probability of a functional failure. Again such features may be considered as part of the safety argument to address claims relating to functional integrity requirements. It is important to note that it is not the objective of this paper to provide a comprehensive list of architectural mitigation measures. Interested readers can refer to Avizienis and Laprie (1986) and Wu and Kelly (2004).

The argument fragment shown in Figure 4 illustrates how an argument regarding diversity in the software architecture for a long range air-to-air missile (LRAAM) can be used to support a claim concerning the role of software in preventing premature launch. Further details of the LRAAM system are provided in Weaver (2003). In particular, the argument in Figure 4 focuses on the diverse implementation of the interlock handler components. As shown in Figure 5, the interlock handler components can cause the premature launch failure mode. The argument in Figure 4 is represented using the goal structuring notation (Kelly, 1998). GSN explicitly represents the individual elements of goal-based arguments (requirements, goals, evidence and context) and (perhaps more significantly) the relationships that exist between these elements (i.e., how individual requirements are supported by specific claims, how claims are supported by evidence and the assumed context that is defined for the argument). The principal purpose of any argument is to show how goals (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence. As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g., adopting a quantitative or qualitative approach), the rationale for the approach and the context in which goals are stated (e.g., the system scope or the assumed operational role).

Figure 4 An argument over software diversity

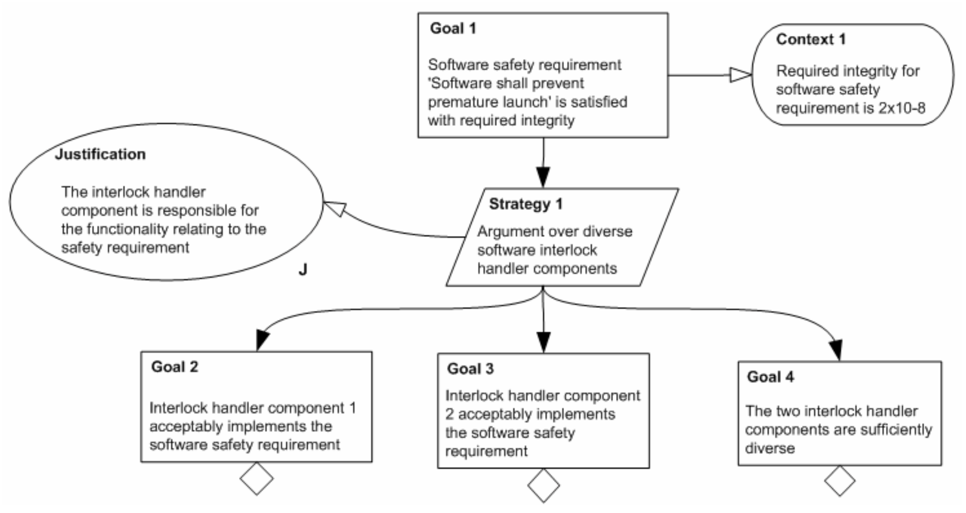
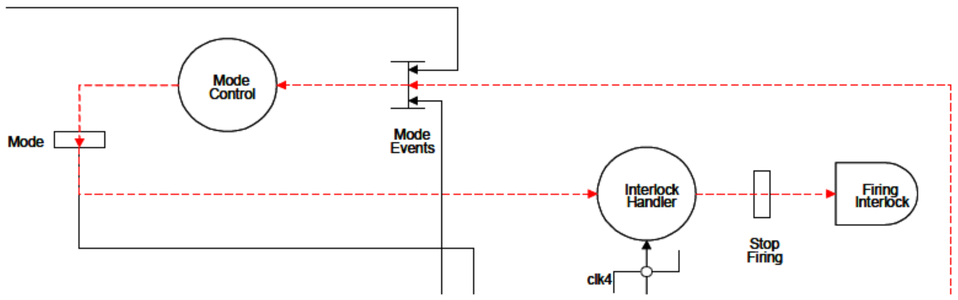

Goal 1
Software safety requirement 'Software shall prevent premature launch' is satisfied with required integrity
Context 1
Required integrity for software safety requirement is 2x10-8
Justification
The interlock handler component is responsible for the functionality relating to the safety requirement
J
Strategy 1
Argument over diverse software interlock handler components
Goal 2
Interlock handler component 1 acceptably implements the software safety requirement
Goal 3
Interlock handler component 2 acceptably implements the software safety requirement
Goal 4
The two interlock handler components are sufficiently diverse

Figure 5 Interlock and mode controller components in the LRAAM software architecture (see online version for colours)

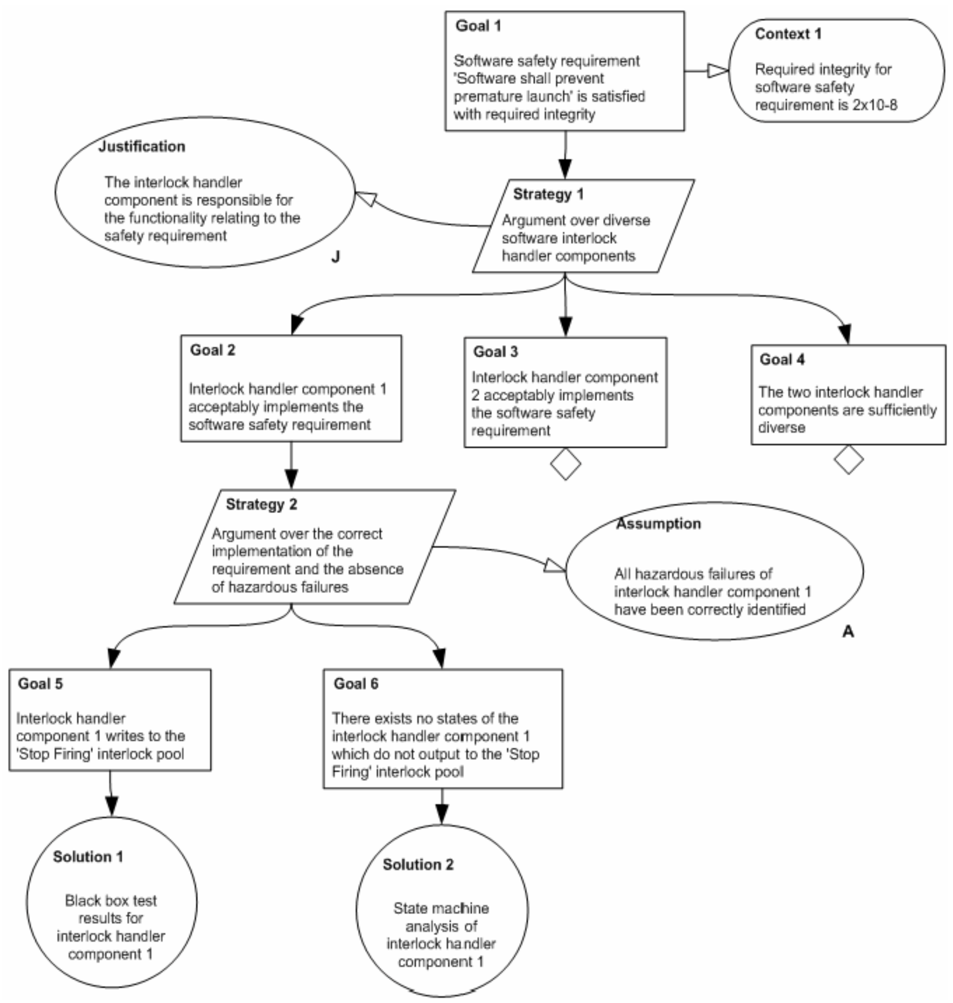


Source: Weaver (2003)

Note that the diamond symbol beneath a goal, as depicted in Figure 4, is used to symbolise that the goal requires further development (support by argument and evidence). The way in which these goals are developed is considered later. Note the importance of Goal 4. If it were not possible to provide an argument to support Goal 4, then the strategy of having the two software components would not have the required effect of addressing the integrity with which the safety requirement is achieved. Note that the strategy adopted in this argument means that the integrity requirement for Goal 2 and 3 is reduced from 2×10^{-8}. As has already been discussed, providing a precise value for the revised integrity requirements is problematic. However, as we shall see later, the knowledge that a lower integrity is required is still useful in demonstrating the sufficiency of the final argument.

So, although it is possible to make safety argument claims which relate directly to integrity, most of the software safety arguments that can be made are not about how good the software product is, but about how well we know whether the product is good or not, i.e., the confidence we have in that product. It is therefore inevitable that the claims that are put forward in the software safety argument, although they may initially deal with integrity, are likely to ultimately deal only with assurance. The key to a compelling argument is to ensure that once the argument claims switch solely to assurance claims, the link to integrity (the ultimate objective) is not lost.

Figure 6 Providing assurance in Goal 2


Goal 1
Software safety requirement 'Software shall prevent premature launch' is satisfied with required integrity
Context 1
Required integrity for software safety requirement is 2x10-8
Justification
The interlock handler component is responsible for the functionality relating to the safety requirement
J
Strategy 1
Argument over diverse software interlock handler components
Goal 2
Interlock handler component 1 acceptably implements the software safety requirement
Goal 3
Interlock handler component 2 acceptably implements the software safety requirement
Goal 4
The two interlock handler components are sufficiently diverse
Strategy 2
Argument over the correct implementation of the requirement and the absence of hazardous failures
Assumption
All hazardous failures of interlock handler component 1 have been correctly identified
A
Goal 5
Interlock handler component 1 writes to the 'Stop Firing' interlock pool
Goal 6
There exists no states of the interlock handler component 1 which do not output to the 'Stop Firing' interlock pool
Solution 1
Black box test results for interlock handler component 1
Solution 2
State machine analysis of interlock handler component 1

If we return to Figure 4, Goals 2 and 3 require to be supported by argument and evidence to demonstrate that software safety requirement is implemented by two diverse interlock handler components. Whereas it was possible in supporting Goal 1 to directly address the integrity requirement, Goals 2 and 3 can instead only be addressed through consideration of assurance (at this point no further strategies for directly impacting the achieved integrity have been identified). Figure 6 shows an argument for how Goal 2 may be supported. It can be seen in Figure 6 that the argument and evidence presented to support Goal 2 are providing assurance in that goal (confidence that the claim made is true), but the approach used, based on testing and analysis of the component, does not directly influence the integrity associated with the safety requirement in the way that was seen in Figure 4.

It should be noted in Figure 6 that the argument relies on the assumption that all the hazardous failures have been correctly identified. In the fully developed argument it would be expected that an argument would instead be presented to provide assurance in the truth of this assumption. In Figure 6, Goal 2 represents the point at which the argument switches from a consideration of both integrity and assurance, to purely a demonstration of assurance. It is therefore necessary to demonstrate that the assurance achieved in Goal 2 is commensurate with the integrity requirement at that point.

Determining the assurance achieved in a goal by the argument provided depends upon a number of factors including the assurance of the supporting evidence, and the extent to which the supporting argument and evidence gives reason to believe that the goal is true. Menon et al. (2009) provide a framework for assessing the assurance achieved based on a number of factors including the scope and independence of the supporting claims. Producing an explicit argument (such as through the use of GSN) makes it easier to understand the structure of the argument and therefore assess the assurance achieved in the safety claims. The assurance of the evidence can be assessed based on a consideration of the processes used to generate the evidence. This is discussed in more detail in the next section.

For the assurance of a safety claim to be considered sufficient, that assurance must be commensurate with the integrity requirement associated with that claim. This is a principle which is already well established. As we saw earlier in this paper, in many prescriptive approaches, the association of a requirement for assurance with an integrity requirement is a common practise. This is also seen in standards which are less prescriptive in nature. For example the UK Defence Standard 00-56 (MoD, 2007) rather than prescribing processes to be followed, instead sets out a small number of higher-level objectives, which include a requirement for the production of a safety argument. This requirement states that, ‘the argument shall be commensurate with the potential risk posed by the system’.

8 The role of software process assurance

Although we emphasised in the previous sections the fundamental role of product-based software arguments, the role of the process should not be underestimated in the overall software safety case. A software safety case comprises both product-based arguments and process-based arguments. That said, the relationship between these two types of arguments should be carefully maintained. This relationship is based on the sufficiency of the process-based argument to demonstrate the trustworthiness of the evidence used in the product-based argument. This evidence is typically generated from review, analysis and testing. That is, the foundation of any product-based argument, i.e., the evidence, depends on the verification artefacts generated from the software process.

However, this process can fail to deliver its expected artefacts to the required assurance and consequently contribute to the generation of untrustworthy evidence. The process may fail due to ambiguous and unsuitable notations, unreliable tool-support, flawed methods and techniques or incompetent personnel. In other words, assurance in the software safety case may be undermined by weaknesses or uncertainties about the quality and adequacy of the process that has generated the evidence used in the product-based argument. The trustworthiness of this evidence depends on the quality and adequacy of the software lifecycle process to produce the evidence to the intended assurance (i.e., the simple question: why should I trust the evidence?).

Disregarded flaws in the process activities and process resources may propagate into software system itself. The risk of these process flaws should be identified and mitigated through focused and targeted assessment of a valid model of the software lifecycle process. This assessment should provide evidence that the risk of the process failing to deliver the required evidence to the intended assurance is acceptable. If the process assessment uncovers unacceptable risks, additional risk reduction mechanisms may have to be integrated into the process in order to reduce the risk of producing untrustworthy evidence. The quality and adequacy of the process should be explicitly communicated in the form of a process-based argument that demonstrates the trustworthiness of items of evidence in a product-based argument.

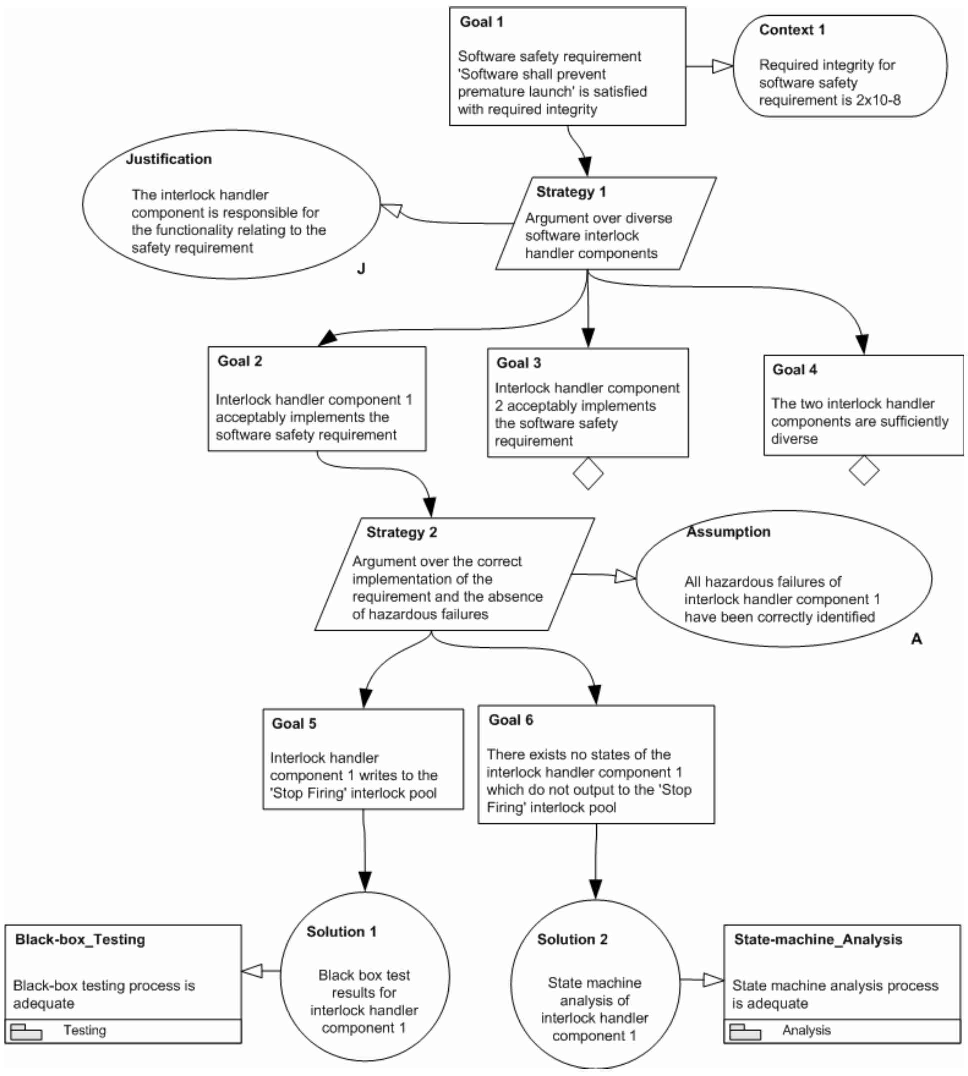
However, not all process activities pose the same level of risk and therefore require the same degree of rigour. The degree of rigour in the process activities should be proportionate to the level of assurance in the evidence as required by the product-based safety argument. This is crucial for software as software failures relate to the degree of freedom from systematic errors in the design – introduced through failings in the software lifecycle process. The key difference between the approach presented in the paper and that of prescriptive software standards is that the rigour in the process in our approach is dictated by the level of assurance placed on the evidence as required by the product-based safety argument – the higher the required confidence in the evidence, the more demanding are the requirements on the software production process. This offers the software engineers and the safety analysts the flexibility of selecting methods and techniques which they can justify to be suitable for the generation of evidence to the required level of assurance, rather than mere compliance with a prescribed, 'one size fits all' process.

We will illustrate the relationship between software product assurance and software process assurance by revisiting the argument depicted in Figure 6. This argument, which is product-based, lacks a clear reference to any process assurance that addresses the trustworthiness of the product evidence (i.e., black-box testing and state machine analysis). Firstly, black-box testing ('Solution 1') is an effective verification technique for showing the satisfaction of safety requirements. However, confidence in the black-box testing depends on assuring the testing process. For example, testing factors that need to be addressed by the process-based argument include issues such as:

- Is the testing team independent from the design team?
- Is the process of generating, executing and analysing test cases carried out systematically and thoroughly (i.e., adequacy of test data)?
- Is the traceability between safety requirements and test cases well-established and documented?

Similarly, state machine analysis (Solution 2) is a powerful formal method for specification and verification. Nevertheless, process justification is required to reveal the mathematical competence of the verification engineers and their ability to demonstrate, for instance, the correspondence between the mathematical model and the software behaviour at run-time (Hall, 1990). Mistakes can be made in formal proofs in the same way that they can be made in coding (more on that in the last section). Therefore, the quality of the verification process by means of formal methods can be as important as the deterministic results such methods produce.

Figure 7 Integrated product and process argument


Goal 1
Software safety requirement 'Software shall prevent premature launch' is satisfied with required integrity
Context 1
Required integrity for software safety requirement is 2x10-8
Justification
The interlock handler component is responsible for the functionality relating to the safety requirement
J
Strategy 1
Argument over diverse software interlock handler components
Goal 2
Interlock handler component 1 acceptably implements the software safety requirement
Goal 3
Interlock handler component 2 acceptably implements the software safety requirement
Goal 4
The two interlock handler components are sufficiently diverse
Strategy 2
Argument over the correct implementation of the requirement and the absence of hazardous failures
Assumption
All hazardous failures of interlock handler component 1 have been correctly identified
A
Goal 5
Interlock handler component 1 writes to the 'Stop Firing' interlock pool
Goal 6
There exists no states of the interlock handler component 1 which do not output to the 'Stop Firing' interlock pool
Black-box_Testing
Black-box testing process is adequate
Testing
Solution 1
Black box test results for interlock handler component 1
Solution 2
State machine analysis of interlock handler component 1
State-machine_Analysis
State machine analysis process is adequate
Analysis

To tackle the above limitations, we propose to address process uncertainty through linking process-based arguments to the items of evidence used in the product-based safety argument. Such process-based arguments address issues of tool and method integrity, competency of personnel, and configuration management. Figure 7 shows a modified version of the argument previously depicted in Figure 6. This version uses an extension to GSN – the 'away' Goal' (e.g., Black-box_Testing and State-machine_Analysis) to attach process-based arguments to the items of evidence in product-based argument. Away goals are used within the arguments to denote claims that must be supported but whose supporting arguments are located in another part of the safety case.

Figure 8 Black-box process argument

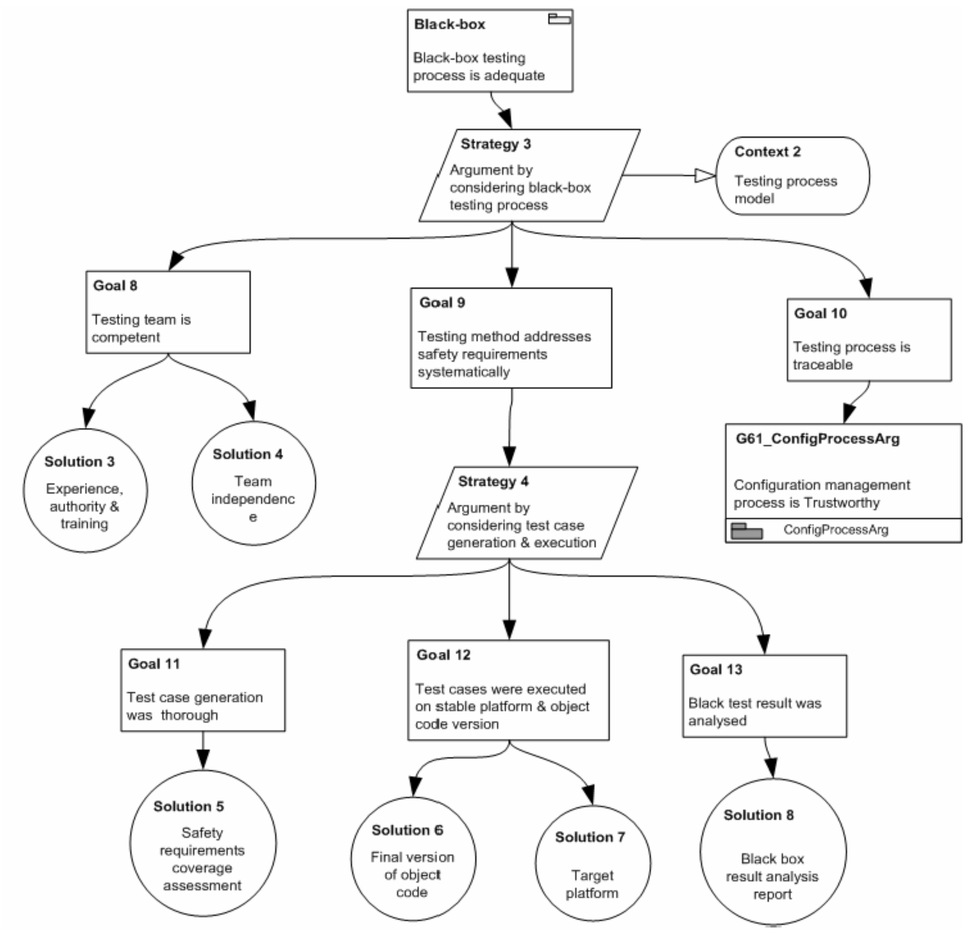

Black-box
Black-box testing process is adequate
Strategy 3
Argument by considering black-box testing process
Context 2
Testing process model
Goal 8
Testing team is competent
Goal 9
Testing method addresses safety requirements systematically
Goal 10
Testing process is traceable
Solution 3
Experience, authority & training
Solution 4
Team independence
Strategy 4
Argument by considering test case generation & execution
G61_ConfigProcessArg
Configuration management process is Trustworthy
ConfigProcessArg
Goal 11
Test case generation was thorough
Goal 12
Test cases were executed on stable platform & object code version
Goal 13
Black test result was analysed
Solution 5
Safety requirements coverage assessment
Solution 6
Final version of object code
Solution 7
Target platform
Solution 8
Black box result analysis report

Figure 8 shows the process-argument for the Black-box_Testing away goal. Here, the argument stresses the importance of process assurance to justify the trustworthiness of the black-box testing evidence. The process assurance addresses team competency, test case generation, execution and analysis, and testing traceability. Firstly, the competency of the testing team (Goal 8) is supported by claims about the team's qualifications and independence from the design team. Secondly, the argument contains a claim that the process of generating, executing and analysing test cases is systematic (Goal 9). This claim is supported by items of evidence showing that the test cases cover all defined safety requirements and executed on the final source code and target platform. Finally, the argument shows that the black-box testing process is traceable. However, in order to avoid complicating the representation of the argument, the justification argument for traceability is documented elsewhere (module: ‘ConfigProcessArg’).

In short, without focused and explicit process-based arguments which justify the level of assurance needed for the items of evidence in the product-based argument, the overall software safety case may be undermined by highlighting uncertainties about the provenance of these items of evidence. It is also important for the process argument to show that the degree of rigour in the process activities is proportionate to the level of assurance in the evidence as required by the product-based safety argument. Prescriptive process-based software standards offer guidance on ‘good practise’ software engineering methods and techniques and the way in which factors such as independence in the process can improve confidence. However, in the context of the approach presented in this paper, the use of these methods and techniques to produce evidence should be justified against the degree of assurance as defined in the product-based argument. In other words, the use of these methods and techniques should not be linked with the direct achievement of the quantitative safety integrity requirements.

9 Discussions and conclusions

In this paper, we discussed the limitations of demonstrating the satisfaction of safety integrity requirements allocated to software by means of quantitative evidence. We also discussed the weaknesses of addressing these limitations by directly appealing to process assurance, through compliance with the process defined in prescriptive software standards. We contended that there is a ‘large inductive gap’ between the quantitative integrity required for a safety function and the assurance of the development process for that function, i.e., good tools, techniques and methods do not necessarily lead to the achievement of the required integrity (e.g., in the form of failure rates). The direct correlation between the quality of the process and the failure rate of the safety function is almost infeasible to justify.

We proposed that this large inductive gap between software integrity and software process assurance could be narrowed down by an explicit definition of a product-based software argument. The role of this argument is to justify the transition from arguing about software integrity to arguing about software assurance by showing how the evidence, in the context of the software product argument, provides assurance which is commensurate with the required integrity. By openly recognising and reasoning about, this inductive gap, the sufficiency of the evidence, produced by the process through testing and analysis, can be more easily determined and justified.

That said, it is reasonable to pose the following three questions:

Question one: Can the software engineer be able to confirm to the systems engineer that the allocated integrity requirements have been achieved by the software?

One of the problems that is currently experienced by systems engineers is that the utility of what is presented back to the system level by the software engineer is often unclear. In previous sections we described how integrity requirements are allocated to software based on system level analysis. Due to the problems that have been discussed, the software engineer will rarely be able to confirm to the systems engineer that such an integrity requirement has been achieved. Instead, as a result of following a prescriptive approach, the software engineer is only able to confirm that the processes defined for that level of software have been followed. Such information, although perhaps of comfort to software regulatory authorities, is in fact is of little relevance with relation to overall system safety. To be of more relevance, the software safety process must provide information which is more closely related to the original requirement allocated from the system level. We feel that an approach based on product assurance is more likely to provide such information, since the claims that can be made for the software relate specifically to the properties of the software in which the integrity is required (the safety functions).

Question two: What role can existing highly prescriptive standards play in assuring the adequacy of the process in producing intended evidence?

Highly prescriptive certification approaches, such as DO178B, provide valuable guidance on how to implement high-quality and repeatable software engineering processes. However, the problem lies in explaining the rationale as to why the achievement of the software safety integrity requirements is assured by compliance with the prescribed techniques and methods that the standards associate with these integrity requirements. The processes prescribed in these standards can be used if the relationship between the required software integrity and the assurance in the evidence produced by these processes is justified in the context of the properties of the software product.

For example, the safety argument developed for the software will often consider traceability and verification of software requirements. In such cases, much of the evidence generated through following a standard such as DO178B may be relevant to supporting claims made in the argument. However, the primary focus for the safety argument is upon demonstrating the required safety properties of the software. Therefore, rather than providing general claims about requirements traceability and verification, the software safety argument must provide specific claims relating to the safety requirements identified for the software. In addition, to be compelling, the safety argument must consider all aspects of the safety of the software which may undermine assurance in the specific safety claims made about the product. This may, for example, involve analysing the potential failure modes of the software design. The software safety argument may therefore lead to a requirement for evidence additional to that which is generated from following any particular standard. It is only through constructing the software safety argument that the sufficiency of any existing approach can be determined.

It is important to note here that moving away from prescription to a safety case approach will involve the development of clear guidelines for developing and reviewing software safety arguments. Rather than merely showing the satisfaction of a list of objectives which are associated with the achievement of a particular SIL, it is the responsibility of the software and safety engineers, in a safety case approach, to present an argument as to why the contribution of their software to system safety is acceptable. Similarly, it is the responsibility of the certification authorities to be able to objectively review the suitably of the presented software safety argument, e.g., uncovering fallacious reasoning and counter-evidence rather than merely auditing compliance with the process defined in applicable standards. In other words, a safety case approach to software assurance will require a shift from "a tick-box mentality to argument-based mind-set" (Penny et al., 2001). Any new guidelines for software safety arguments should provide worked examples and patterns, based on actual successful software safety cases, illustrating how good arguments can be constructed and supported by trustworthy items of evidence. However, the development of such guidelines may be hindered by two factors. Firstly, many developers may regard these worked examples and patterns as the preferred means for compliance rather than an example means for compliance. Secondly, it is often difficult to publish successful software safety cases as they include commercially sensitive data.

Question three: Integrity in hardware is often demonstrated quantitatively. How does hardware integrity relate to assurance? Do we need to assure hardware integrity?

In this paper, we addressed the justification of the transition from software integrity to software assurance mainly because software failures are purely systematic, due to design faults which are difficult to avoid because of the complexity of the software design. In hardware, and despite the ability to quantity hardware failure rates, complex hardware systems also suffer from systematic faults, which are also hard to quantify due to the complexity of the hardware design. So, applying standard statistical hardware reliability techniques to complex hardware systems may not be sufficient to estimate the actual failure rates, i.e., considering both random and systematic failures. For example, Littlewood and Strigini argue that "the 'solution' we sometimes hear to the software problem – 'build it in hardware instead' - is usually no solution at all. The problem of design dependability arises because the complexity of the systems is so great that we cannot simply postulate the absence of design faults". Therefore, even for demonstrating the integrity of complex hardware designs, there is a need for assuring the quantitative evidence by explicitly addressing potential sources of systematic failures such incompetent personnel, ambiguous notations or flawed tool-support. In fact, the same can be said about software systems which are formally verified, e.g., by means of mathematical proofs. These proofs can demonstrate mathematically the satisfaction of safety requirements (i.e., not just meeting, but rather exceeding the required reliability targets). However, complex designs verified mathematically using formal methods are also prone to design faults, i.e., mistakes in proofs, flaws in automated theorem provers or weaknesses in the correspondence between the system and the mathematical model representing it. So, assuring the mathematical evidence, e.g., resulting from model checkers and theorem provers, will often be necessary. In summary, whenever there is complexity, in both the problem and the solution, and regardless of the implementation technology (hardware or software) and methodology (testing or formal methods), assurance has to be considered.

References

Avizienis, A. and Laprie, J. (1986) 'Dependable computing: from concepts to design diversity', *Proceedings of the IEEE*, Vol. 74, No. 5, pp.629–638.

Baggini, J. and Fosl, P. (2003) *The Philosopher's Toolkit – A Compendium of Philosophical Concepts and Methods*, Blackwell.

Butler, R. and Finelli, G. (1991) 'The infeasibility of experimental quantification of life-critical software reliability', *ACM SIGSOFT Software Engineering Notes*, Vol. 16, pp.66–76.

Hall, A. (1990) 'Seven myths of formal methods', *IEEE Software Archive*, Vol. 7, No. 5.

IEC (1998) *IEC 61508 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, International Electrotechnical Commission.

Kelly, T.P. (1998) 'Arguing safety – a systematic approach to managing safety cases', PhD thesis, Department of Computer Science, The University of York.

Littlewood, B. and Strigini, L. (1993) 'Validation of ultrahigh dependability for software-based systems', *Communications of the ACM*, Vol. 36, No. 11, pp.69–80.

Menon, C., Hawkins, R. and McDermid, J.A. (2009) 'Defence standard 00-56 issue 4: towards evidence-based safety standards', *Proceedings of the Seventeenth Safety-critical Systems Symposium*.

McDermid, J. (2001) 'Software safety: where's the evidence?', *Australian Workshop on Industrial Experience with Safety Critical Systems and Software*.

McDermid, J.A. and Kelly, T.P. (2006) 'Software in safety critical systems: achievement and prediction', *Nuclear Future, Thomas Telford Journals*.

MoD (2007) *Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems*, UK Ministry of Defence.

Penny, J., Eaton, A., Bishop, P.G. and Bloomfield, R.E. (2001) 'The practicalities of goal-based safety regulation', *Proceedings of the Ninth Safety-Critical Systems Symposium*.

Redmill, F. (2000) 'Understanding the use, misuse and abuse of safety integrity levels', *Proceedings of the Eighth Safety-critical Systems Symposium*.

RTCA (1992) *DO 178B – Software Considerations in Airborne Systems and Equipment Certification*, Radio and Technical Commission for Aeronautics.

SAE (1994) *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, Society of Automotive Engineers, 400 Commonwealth Drive, USA.

Shooman, M.L. (1996) 'Avionics software problem occurrence rates', *Proceedings of the 7th International Symposium on Software Reliability Engineering*, White Plains, NY, pp.53–64.

Weaver, R. (2003) 'The safety of software – constructing and assuring arguments', PhD thesis, Department of Computer Science, The University of York.

Wu, W. and Kelly, T. (2004), 'Safety tactics for software architecture design', *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, IEEE Computer Society.