Software Safety Arguments: Towards a Systematic Categorisation of Evidence

R. A. Weaver; Department of Computer Science, University of York; York, UK

J. A. McDermid; Department of Computer Science, University of York; York, UK

T. P. Kelly; Department of Computer Science, University of York; York, UK

Keywords: safety arguments, safety case patterns, evidence characteristics

Abstract

The development of software for safety critical systems is guided by standards. Most standards identify processes for different safety integrity levels (SILs) or development assurance levels (DALs). Software is shown to be fit for use primarily by appeal to the standards, supported with appropriate evidence, e.g. from testing. The assumption is that software developed against the requirements of higher SILs will be less prone to critical failures. A paper at the last ISSC questioned this assumption, and proposed instead that an "evidence-based" approach be taken to software. To implement this type of approach requires arguments to reflect the contribution of software to safety in the context of the system. We believe that an "evidence-based" approach can be implemented by using a framework for articulating software safety arguments, based on categorisation of evidence, which is largely independent of the development process. This paper outlines our approach, and shows how the ideas can be presented within a safety case, without precluding the use of existing standards. A key motivation in producing the paper is to expose these rather unconventional views to critical review, and to seek to build acceptance of the principles.

Introduction

Most standards used to guide the development of software for safety critical systems, e.g. DO178B (ref. 1), DS 00-55 (ref. 2), and Part 3 of IEC 61508 (ref. 3), identify processes for different safety integrity levels (SILs) or development assurance levels (DALs). We have previously questioned the extent to which there is evidence that the approaches advocated by these standards are effective in practice (ref. 4-5). It is also widely agreed that these standards do not deal adequately with certain practical situations, e.g. use of legacy or commercial off the shelf (COTS) software. Also they do not deal with some modern practices, e.g. code generators. This is not to say that the standards give no guidance, merely that their emphasis is on the development of "new" software by conventional means. The aim in this paper is to show how to produce a framework for software safety evidence which can be applied independently of the process used for software development, but which doesn't preclude the use of existing standards. The ideas presented are intended to be generic, but they were motivated by work in the aerospace sector.

Current Standards: Current standards recommend a set of techniques to be used at each SIL or DAL. Both the developer and assessor accept that, by following the process of applying these techniques and developing evidence, the software achieves the required level of safety. However, the safety evidence generated does not necessarily give a quantitative demonstration that the SIL or DAL has been achieved. Also, due to the difficulty in detecting software failures in accidents, the commercial sensitivity of failure data, and the extremely high safety levels required of software, it is difficult to determine the operational levels of safety for developed software. Thus it is often not possible to assess before or during operation whether software produced to a SIL or DAL process attains the required level of safety.

Previous research has shown the difficulties in quantitatively proving a software failure rate due to the systematic nature of software failures (ref. 6). Also there is some evidence to show that software developed by a process-based approach may not always meet the required level of assurance (ref. 7). This paper introduces a product-based framework for software safety evidence, instead of following a prescriptive process approach. The intention of this framework is to circumvent the problems associated with a process-based method described in the current standards.

Safety Cases

Our approach is to consider the issues from the point of view of producing a safety case (ref. 8). A safety case includes an argument as to why the system is believed to be safe to deploy in its intended operational context. The safety case is based upon a hierarchy of safety requirements, e.g. to show the tolerable rate of occurrence of some hazard, which we refer to as goals. These goals provide the "spine" of a well-organized safety argument. The main argument is linked to assumptions and contextual information. The goals at the leaves of the safety argument, which cannot be decomposed further, are supported by safety evidence. For example, the evidence may be an FMEA showing that no single point of failure gives rise to a hazard.

In general, no one item of evidence will be sufficient to support the safety argument. Indeed, there may be many different combinations of evidence which, together, could support the argument, and an integral part of the safety process is selecting an appropriate combination of techniques. Justification of the selection is an important part of the safety argument for both the developer and the assessor. For the assessor, it is necessary to show that the evidence is comprehensive and compelling. The developer will be concerned to produce the evidence cost-effectively, e.g. by avoiding unnecessary redundancy in the evidence.

Defining Software Safety Requirements

Safety is a property of a complete system (platform) in a given context. Thus, analysis of deviations from design intent and failures in subsystems and components, including software, must always be considered in terms of their contributions to system level effects (ref. 9). Software Safety Requirements (SSRs) may be allocated from the system level where the software implements safety functions, e.g. detecting the loss of a rotational speed sensor, and synthesizing a speed value from the current to the drive motor. SSRs are also concerned with software contributions to hazards, e.g. the failure of the software to instigate the lowering of an aircraft undercarriage, when this is commanded from the cockpit. The research presented in this paper focuses primarily on arguments for SSRs which relate to software contribution to system level hazards.

Classification of Software Failure Modes: Classification of software failure modes, e.g. into early and late delivery of outputs, allows them to be analysed more effectively. A common form of safety argument can be generated for a specific class of failures, e.g. the use of worst case execution times (WCETs) as part of the argument that late delivery of outputs has been avoided (assuming the hardware is functioning correctly). The development of generic arguments allows both reuse within and between safety arguments, and greater guidance to be given on the initial construction of the argument. The classification used in this framework is based upon Pumfrey's (ref. 9) which is set in the context of the system delivering a service (both internally and externally), and divides software failure modes into:

- Service Provision (Omission and Commission);
- Service Timing (Early and Late);
- Value.

Subdivisions of these failure modes, e.g. into detectable and undetectable value domain failures, may be useful in some contexts, however experience suggests that this is a general "top level" classification.

Constructing the Software Safety Argument

Once all SSRs have been identified an argument is required to show that the SSRs have been met. However this is not sufficient to demonstrate the acceptability of the software, and we identify three types of evidence that are required for a complete software safety argument:

Requirements Validation	Demonstration that the set of SSRs is complete and "accurate", e.g. cover all hazards to which the software can contribute.
Requirements Satisfaction	Demonstration that all Software Safety Requirements have been met.
Requirements Traceability	Demonstration that all Software Safety Requirements have been tracked throughout all stages of System Development and Safety Analysis.

Validation and Traceability arguments will be similar for each SSR (e.g. demonstration of the thoroughness of hazard analysis and the inclusion of traceability matrices). This means that generic arguments can be produced for these two forms of evidence and used for all SSRs. An argument for Satisfaction is dependent upon the type of SSR under consideration. However, as stated above, it is possible to develop generic arguments using the type of Software Failure Mode on which the SSR is based, see figure 1.

Figure 1 shows the top level of the argument framework and has been produced using the Goal Structuring Notation (GSN) (ref. 8). GSN provides a graphical notation for the representation of a safety argument structure. It details the dependencies between requirements and evidence, through a hierarchy of goals, showing assumptions made and providing other contextual information. Figure 1 shows a generic structure for the top level of a software safety argument.

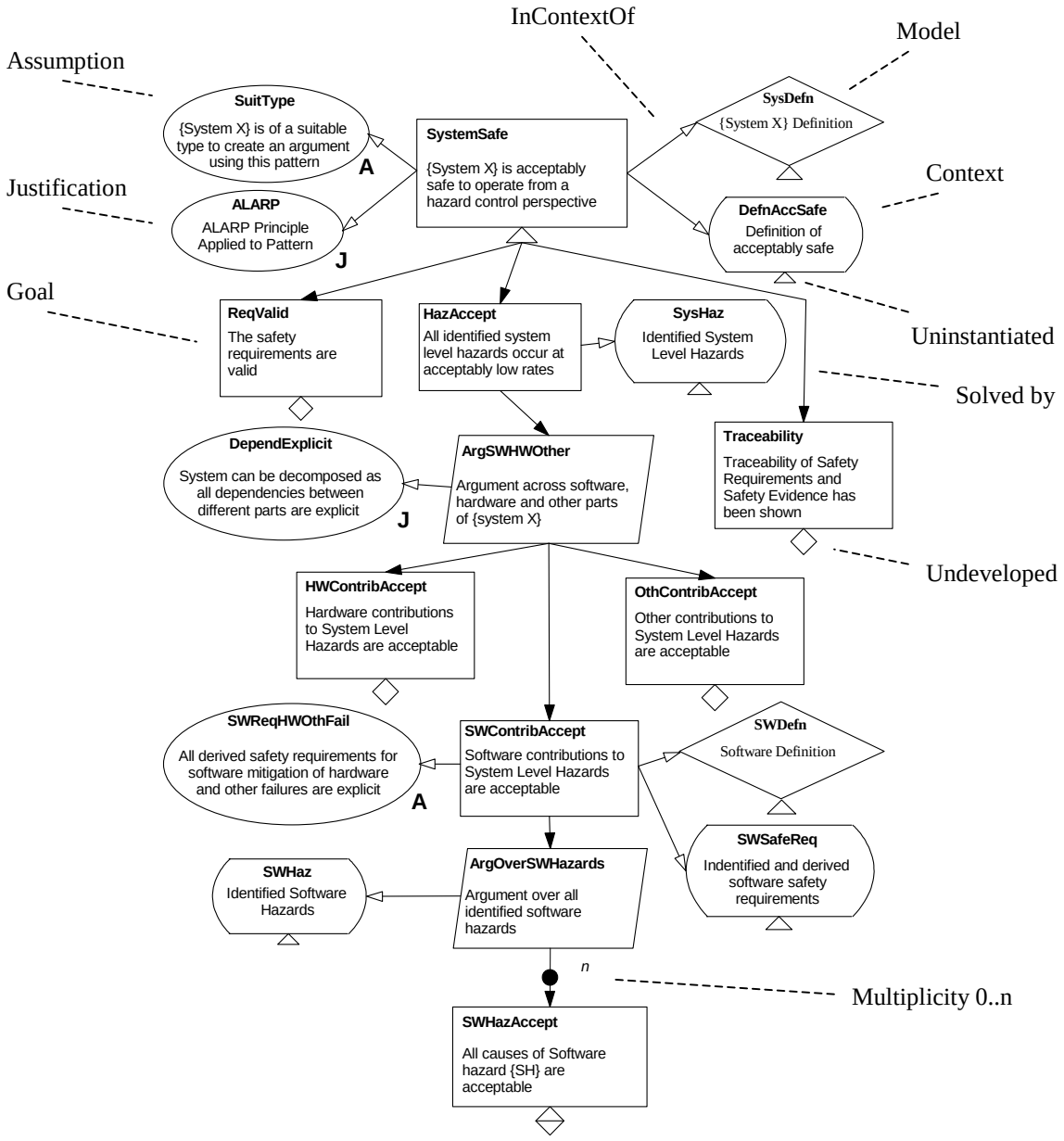

Figure 1 – Top Level Software Safety Argument

Figure 1 starts at the system level with a general goal **SystemSafe**. To show **SystemSafe** it is sufficient to show that the three immediate sub-goals **ReqValid**, **HazAccept** and **Traceability** are met, in other words the goal decomposition can be thought of as an "and tree". Note that these three goals correspond to the three generic safety arguments identified above. The figure also shows the assumptions made in constructing the argument, e.g. **SuitType**, the justifications made in proceeding from one level of goal to another, e.g. **ALARP**, and other contextual information.

The figures presented in this paper all form generic solutions to parts of the software safety argument. Safety Case Patterns (ref. 8) provide generic safety arguments in the form of GSN, which can be instantiated for a specific safety critical system.

A Pattern Catalogue is being generated to represent the framework described. It consists of a collection of highly interrelated patterns which can be combined to form a software safety argument. Dependent on the system being assessed, a selection of patterns can be made. These patterns can then be instantiated and joined together to develop a specific safety argument. The pattern catalogue contains patterns for the different categories of evidence and failure types.

For example, an argument for satisfaction of a SSR through the mitigation of the software failure modes can be created using a generic argument specific to the software failure mode class. The arguments are based on showing one, or a combination, of an acceptable probability of occurrence, absence, or handling of the potentially hazardous failures within the software.

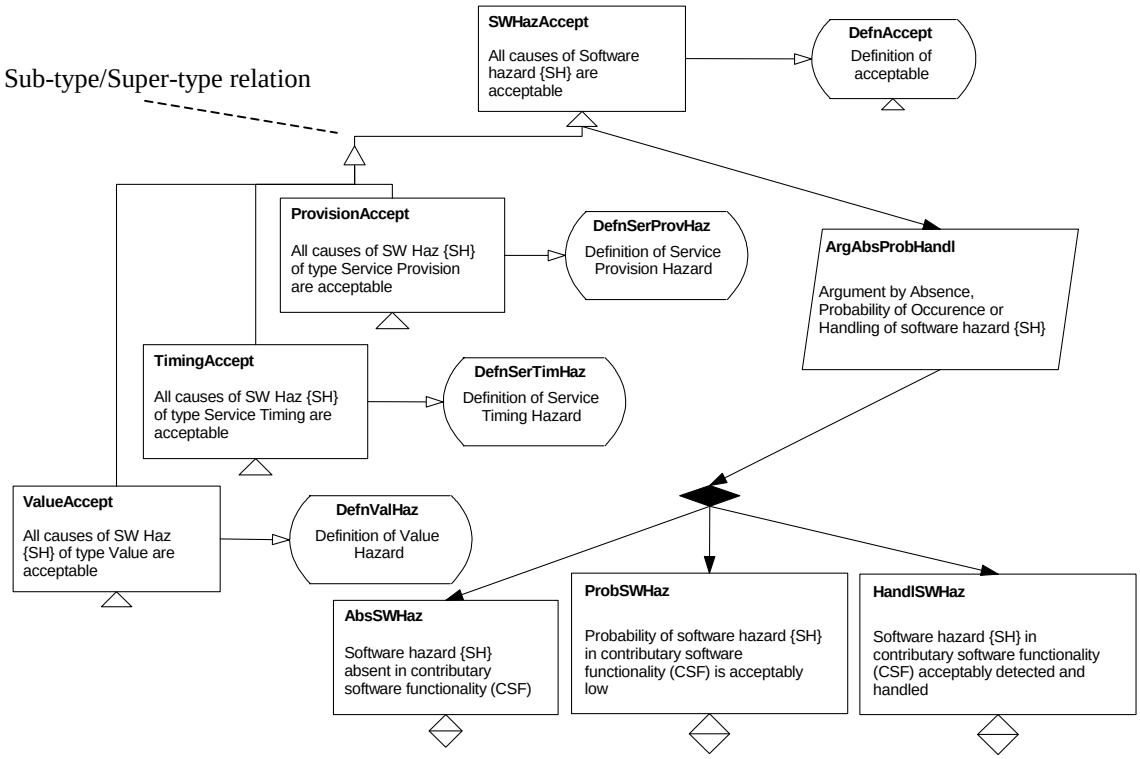

Figure 2 - Argument Across Software Contribution to System Hazards

Figure 2 shows the pattern for the different classes of failure mode, and extends the argument structure shown in figure 1. Any argument that a hazardous software failure mode (software hazard for short) is acceptable is either an argument of the type service provision, service timing or value. Thus, for example, if a software hazard is a late failure, then the goal **SWHazAccept** is replaced (instantiated) by **TimingAccept**.

Note that only one of the goals on the left can be used to instantiate **SWHazAccept**, in a particular argument

The right hand half of the argument shows that the goal can be met in one of three ways:

- The potential software hazard is completely absent (absolute evidence).
- The probability of the software hazard is acceptably low.
- The software hazard is handled.

The latter two arguments may be used together. As the argument is developed, the most appropriate form of goal is chosen – to satisfy the assessor, and to produce the evidence in the most cost-effective manner.

In the case of a late failure, absence might be shown by static analysis of the code (perhaps at object level), plus analysis of the scheduling. (This is rather more complex in practice, but the principle is valid.) In the case of a probabilistic argument might be produced by a combination of testing, and analysis of the distribution of inputs to the system. An argument about detection and handling might involve analysis of a watchdog timer, and also some evidence that the rate of occurrence of timing failures is low enough to be tolerable, e.g. it does not make the control system unstable.

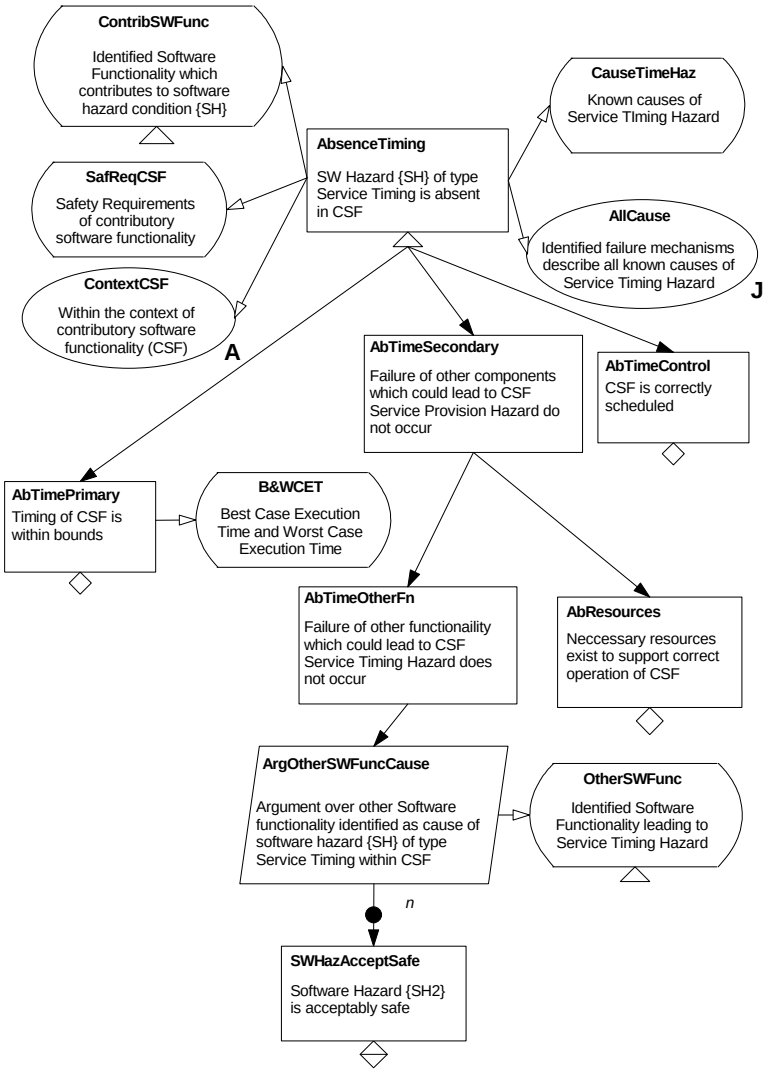

Figure 3 – Failure Mode-based Argument for a Software Safety Requirement

As indicated above, timing arguments will in fact be more complex than the simple example given. Figure 3 presents the amplified goal structure for absence of timing failures **AbsenceTiming**. In essence there is (can be) no timing failure if:

- The primary software component of interest is free of timing errors, e.g. its WCET is within bounds;
- There are no secondary causes of timing failures, e.g. the software components which supply it with inputs do so on time;
- There are no scheduling failures.

Note that the decomposition here is into primary, secondary and command failures – one of the standard decomposition tools used in fault trees. To show the requirement is met for the primary component necessitates similar requirements on secondary components so there is "recursion" in the argument except that the repeated goals, e.g. **SWHazAcceptSafe**, now apply to different software components.

Finally, note that the form of the argument will depend on the software architecture. If a component is triggered by the arrival of data, e.g. from a sensor, then the late delivery of the input can cause the late delivery of its output. In this case, **SWHazAcceptSafe** would have to be applied to software components which provide inputs. However if the components communicate via shared data, and there is no triggering of one component by another, then the source components cannot cause late failures, and they can be eliminated from the argument (this flexibility is not shown in figure 3 above). Of course, running the component with stale data can have deleterious effects, but this will be addressed elsewhere in the argument as it is a value domain issue.

Figures 1, 2 and 3 show how the goals are decomposed from system level to produce component level safety requirements. It is now necessary to consider how to produce evidence to support these requirements, i.e. to show that they have been met.

Supporting the Software Safety Argument

For each bottom level goal derived in the software safety argument (e.g. Timing of Contributory Software Functionality is within bounds), evidence must be developed which is both appropriate and sufficient. One of the primary roles of the safety argument must be to present a rationale as to why the evidence fulfills the goal. The evidence supporting a safety goal may be made up of several items, each of which addresses a different aspect of the goal. This evidence will arise from some form of assessment technique e.g. an inspection, a set of tests or WCET analysis. When selecting techniques to produce evidence it is important to produce sufficient weight of evidence. This can be achieved by considering the role the evidence performs within the complete safety argument and the characteristics of the techniques. Three characteristics: Relevance, Coverage and Independence can be used to show the sufficiency of the evidence.

Relevance	The extent to which the evidence directly addresses the software safety goal.
Coverage	The proportion of the software safety goal which the evidence addresses.
Independence	The extent to which complementary evidence follow diverse approaches in fulfilling the requirement for evidence.

The idea of evidence characteristics is important, but unusual, so it is appropriate to describe these concepts in some detail.

Relating Evidence to the Goal

Relevance of Evidence: It is possible to identify how pertinent the software safety evidence is in addressing the software safety goal. The relevance gives the developer and assessor an understanding of the role that the evidence plays. Software Safety Evidence (SSE) can be of many types, e.g. test results or competency of the development personnel. Again we introduce a classification, this time of evidence types:

Direct	Evidence that directly shows software meets the safety goal.
Backing	Evidence that shows that the direct evidence is soundly based.
Reinforcement	Evidence which shows that direct evidence can be extrapolated to meet higher integrity requirements than can be met with the direct evidence alone.

We take the view that direct evidence is the most relevant in showing that an SSR is met, i.e. it carries the greatest weight. For example, the results of WCET analysis is direct evidence. Backing evidence shows that the direct evidence is soundly based. Backing evidence addresses issues concerned with the system development and analysis process, for example that the WCET results relate to the system as delivered (configuration control) and that the development personnel were adequately competent. Reinforcement evidence allows the developer to extend the direct evidence, so as to meet a goal which cannot be supported by the direct evidence alone. Like direct evidence, reinforcement evidence is product-based, for example historical data relating to in service safety performance of current systems.

Normally it will be necessary to have both direct and backing evidence for any "leaf" goal in a software safety case. Reinforcement of evidence will only be required when the direct and backing evidence is judged to have insufficient weight, see the observations below.

Coverage of Evidence: Coverage is a term often used with respect to testing. We focus here on software aspects of coverage. The higher the test coverage, the greater the confidence in the results. The concept of coverage can be applied to all aspects of safety evidence. The coverage of evidence identifies the proportion of the software, or the property of interest, for which safety goal has (demonstrably) been met. For example, the requirement for availability of resources **AbResources** in Figure 3 should be applied to all resources, e.g. input-output buffers and device registers, on which a software component depends. Thus, if a software component depends on four resources, some analysis technique may only deal with two of them, giving 50% coverage.

Where appropriate a coverage structure can be developed based on the causally related parts of the software under consideration. In general, coverage is being used here to identify the proportion of causally relevant items which have been investigated, not lower level issues as with the use of MC/DC coverage in DO178B (reference 1).

Relating Multiple Items of Evidence

It will be rare that one item of evidence will adequately support a software safety goal. In this situation multiple items of evidence are used to support the goal. It is important to consider the relationship between the items of evidence, to show that together they satisfy the goal.

For example, it should be noted that two items of evidence each having 50% coverage of a goal, does not necessarily lead to 100% coverage of the goal. This highlights the necessity of developing a complete coverage structure so as to identify all related components within the software or property of interest. Defining coverage criteria will vary from system to system, and "good practice" coverage criteria will also vary, but it is important to identify the criteria explicitly as it can then be used to show that the argument is complete.

Independence of Evidence: In considering software safety evidence it is necessary to identify ways in which the evidence can be undermined. There is an analogy with common cause failures. Items of evidence intended to support the same safety goal in complementary ways must be independent. Independence may be undermined if they both rely on the same incorrect base data, e.g. a program control flow graph. Independence can be either Mechanistic or Conceptual. Conceptually different approaches are based on different underlying theories. For example static and dynamic analysis are conceptually different approaches to developing evidence (one involves running the program; the other does not). Mechanistically different approaches implement the same underlying theory in different ways. For example testing techniques in a host using a coverage analyser in comparison to a hardware target are mechanistically

different. As a general rule conceptual independence is more significant than mechanistic independence, thus an order can be created as follows:

Conceptual and Mechanistic Independence
Conceptual but not mechanistic Independence
Mechanistic but not conceptual Independence
No Independence

The level of independence required depends on the criticality of the items of evidence, see below.

Observations

The framework has been developed to allow the completeness of the safety argument to be shown. From the point of view of the assessor it is as important to be able to see what has not been covered as what has been covered. If the framework described in this paper is used, this task should become easier. From the point of view of the developer it is possible to demonstrate both what the evidence shows and the level of confidence that can be gained in the evidence.

One of the potential attractions of the "evidence based" approach is that it may offer a means of dealing with existing (legacy) software, commercial off the shelf (COTS) software, or modern practices (e.g. code generators) on an equal footing. In other words, by developing an approach which requires evidence of the safety of the software product in its system context, we can avoid the difficulties of how we assess the safety of software not developed to the requirements of the existing standards. When developing arguments about systems for which limited evidence is available, the evidence characteristics and framework can be used to identify what objectives can be shown to be satisfied and what objectives cannot. From this, approaches (e.g. wrapping) can be developed which specifically cover objectives that cannot be met by evidence.

The discussion above has avoided issues such as SILs, and the use, or not, of coverage criteria such as MC/DC. This is deliberate, as we believe that they are of limited value in a software safety case. Space does not permit a full articulation of the arguments, but we briefly consider the issues.

Instead of using SILs or DALs we would recommend that any safety critical goal be supported with at least two items of independent evidence, preferably with conceptual and mechanistic independence, and at least conceptual independent (for the satisfaction evidence). For safety related goals we would accept simplex evidence; as more that one safety related goal will have to be violated before a hazard arises, thus we still have independent evidence against any safety critical requirements. This is fundamentally a more logical standpoint to argue from, providing a level of consistency, and is much simpler, and easier to explain, than SILs and DALs.

As for MC/DC and similar requirements, we observe that they are not stated in terms of hazards. The approach outlined here naturally leads to requirements for hazard-directed evidence, thus the coverage issue becomes more one of "have we analysed all causally relevant components", not what is the coverage of the code. For example, if we are concerned with an omission failure (a form of service provision failure) then we need to know if there are paths through the code of the component where the relevant function is not called, if the component might not be scheduled, etc. The framework described in this paper makes explicit the practical benefit of issues such as MC/DC of which we have previously questioned the value (ref. 4). However we have been investigating a further characteristic of "weight of evidence" which we variously call integrity and thoroughness. We will amplify on this concept, should it prove necessary.

Direct application of the patterns outlined above might seem to make arguments very complex, and require large quantities of evidence. However the approach need not produce a great burden on the developer. Our intent is that developers use the safety case patterns to identify the evidence required for the set of software components. Once this is done it will be discovered that there is a lot of overlap in the requirements, i.e. that the same item of evidence is needed for more than one purpose. Thus a verification and validation plan can be produced, which will result in the required evidence being generated as efficiently as possible. However,

no unnecessary evidence should be produced, and the goal structures will show the impact if it proves impractical to produce some piece of evidence that is more likely to gain sufficient coverage.

Although we have described our approach in general terms, we are involved in a number of experimental applications of the ideas. One application is an aerospace project which involves using the approach to set out requirements on subcontractors for the production of software safety evidence. This is being used as an alternative to reliance on existing standards, e.g. DS 00-55. We are also undertaking work on behalf of the European Space Agency (ESA), with the aim of using this evidence framework to identify alternative ways of producing safety evidence. If the current phase of the work is successful ESA expect to carry out some comparative experiments which, hopefully, will lead to a greater understanding of the underlying principles, and how to present the ideas in a practical form for use in industry. Finally, an early version of these ideas underlies the CAA's SW01 software standard (ref. 10).

Future Research

Due to the stage of development of the current research it is useful at this point to give a brief overview of future research that will be undertaken. The evidence characteristics described require further development, so as to give additional guidance on the use of Relevance, Coverage and Independence, and possibly the inclusion of another concept Thoroughness (or Integrity) mentioned above. While a number of Safety Case Patterns have been developed, some remain to be designed before a complete pattern catalogue can be used as assistance for the development of software safety arguments. In attempt to validate the framework described in this paper, a direct comparison with current software safety standards is planned which would aid identification of the flaws in both the Framework and current standards. Finally further consideration of the application of the structure with respect to developing arguments for COTS/Legacy systems is planned.

Summary

This paper has presented an evidence-based framework for generating software safety arguments, an overview of which is shown in Figure 4. Both the underlying concepts and a method of implementation have been described. The benefits of this approach in comparison to current practices have been briefly discussed, and due to the stage of current development, further research of the framework has been outlined, most importantly the need to validate the approach. If compared with current standards and best practices, the proposal presented is an unconventional approach to the development software safety arguments. Critical review is invited of this approach as we believe it to be a framework that can generate arguments which justify the acceptable safe nature of software to a greater degree than current practices.

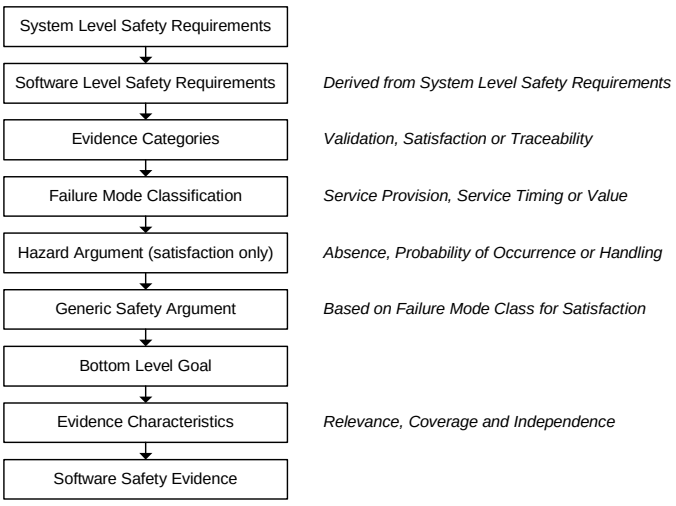

Figure 4 – An Overview of the Framework for Constructing Software Safety Arguments

References

1. RTCA and EUROCAE. Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics RTCA DO-17B/EUROCAE ED-12B, 1992.
2. UK MoD. 00-55 Requirements of Safety Related Software in Defence Equipment. Ministry of Defence, Defence Standard, 1996.
3. IEC. IEC-61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. International Electrotechnical Commission, 1999.
4. J. A. McDermid, D. J. Pumfrey. Software Safety: Why is there no Consensus?. in Proceedings of the 19th International System Safety Conference, Huntsvillee, System Safety Society, 2001.
5. J. A. McDermid. Software Safety: Where's the Evidence?, in Proc. 6th Australian Workshop on Industrial Experience with Safety Systems and Software, Australian Computer Society, 2001.
6. B. Littlewood, L. Strigini. Validation of Ultrahigh Dependability for Software-based Systems. in Communications of the ACM 36(11), p69-80, 1993.
7. K. J. Harrison. Static Code Analysis on the C-130J Hercules Safety Critical Software. Aerosystems International, UK,.
8. T. P. Kelly. Arguing Safety – A Systematic Approach to Managing Safety Cases. Department of Computer Science, University of York, York, UK, 1999.
9. D. J. Pumfrey. The Principled Design of Computer System Safety Analysis. Department of Computer Science, University of York, York, UK, 2000.
10. CAA. Regulatory Objective for Software Safety Assurance in Air Traffic Service Equipment SW01. Civil Aviation Authority,.

Biography

R. A. Weaver, M.Eng., Research Student, Department of Computer Science, University of York, York, UK, telephone - +44(0)1904 433388, e-mail – rob.weaver@cs.york.ac.uk

Rob Weaver has been a Research Student in the BAE SYSTEMS funded Dependable Computing Systems Centre at the University of York since November 1999. Before beginning his Ph.D., funded by the Engineering and Physical Sciences Research Council (EPSRC) and BAE SYSTEMS, he attained an M.Eng. in Computer Systems and Software Engineering from the University of York.

J. A. McDermid, Ph.D., Professor of Software Engineering, Department of Computer Science, University of York, York, UK, telephone - +44(0)1904 432726, facsimilie - +44(0)1904 2708 e-mail – john.mcdermid@cs.york.ac.uk

John, McDermid has been Professor of Software Engineering at the University of York since 1987 where he runs the high integrity systems engineering (HISE) research group. HISE studies a broad range of issues in systems, software and safety engineering, and works closely with the UK aerospace industry. Professor McDermid is the Director of the Rolls-Royce funded University Technology Centre (UTC) in Systems and Software Engineering and the BAE SYSTEMS-funded Dependable Computing System Centre (DCSC). He is author or editor of 6 book, and has published about 250 papers.

T.P. Kelly, D.Phil., Lecturer, Department of Computer Science, University of York, York, UK, telephone - +44(0)1904 432764, facsimilie - +44(0)1904 2767 e-mail – tim.kelly@cs.york.ac.uk

Tim Kelly is a lecturer in software and safety engineering at the University of York. He is also Deputy Director of the Rolls-Royce funded UTC. His expertise lies predominantly in the areas of safety case development and management. Tim has provided extensive consultative and facilitative support in the production of acceptable safety cases for the defence, railways and power generation sectors. As part of the HISE research group in York, Tim has trained over 1000 people representing over 50 organisations in system safety engineering topics. He has published a number of papers on safety case development in international journals and conferences and has been an invited panel speaker on software safety issues.