

A Systematic Approach for Developing Software Safety Arguments

R.D. Hawkins, Ph.D.; Software Systems Engineering Initiative; The University of York, York, UK

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

Keywords: Software Safety, Assurance, Safety Cases, Safety Critical Software, Argumentation

Abstract

It is becoming increasingly common to develop safety arguments (also called assurance arguments) to demonstrate that the software aspects of a system are acceptably safe to operate. A software safety argument enables a compelling justification of the sufficiency of the software to be provided, whilst also giving the software developer flexibility to adopt the development approach that is most appropriate for their system.

To be compelling, the safety argument must provide sufficient assurance in the safety claims made about the software. Our investigations have shown that creating compelling software safety arguments remains a major challenge for those developing safety-related software. To help address this challenge we have developed a systematic approach to software safety argument construction which explicitly considers and addresses assurance.

Our approach has two key elements which, when used together, facilitate the construction of compelling software safety arguments. Firstly a method for argument construction is proposed, this method extends an existing method by explicitly considering assurance at each step. Secondly a set of software safety argument patterns have been developed. These patterns document reusable software safety argument structures which may be instantiated for the system under consideration. These patterns again build on existing work, and have been developed such that they highlight as clearly as possible where assurance may be gained and lost during the development of the argument.

Introduction

In order to demonstrate that a system is acceptably safe to operate, it is possible to provide a safety case for that system. A safety case is defined in reference 1 as, "The safety case shall consist of a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment". For systems which contain software, the safety case must consider the contribution of the software to the safety of the system. Constructing a safety argument requires that safety claims are identified. These safety claims capture the objectives of the safety case. The safety argument must demonstrate that the safety claims are supported by the evidence generated. In supporting a safety claim, a hierarchy of sub-claims is constructed which establish the relationship between the evidence and the overall safety case objectives. Creating a compelling, well structured safety argument makes it possible to provide an explicit justification that the system is acceptably safe to operate.

When demonstrating the safety of software it remains common to adopt a prescriptive approach where the developer of the software demonstrates the safety of the system by demonstrating compliance with requirements set out by the appropriate regulatory authority, generally in the form of a prescribed process in a standard. This approach is the basis of the standards most commonly used for developing software used in safety related applications, such as reference 2 and reference 3. Alternatively, with a safety case (or goal-based) approach the regulatory authority does not prescribe a method for demonstrating that the software is acceptably safe. Instead it is the responsibility of the developer of the software to demonstrate to the regulatory authority that the software is acceptably safe to operate. We believe that a goal-based approach has a number of advantages over a prescriptive approach. The philosophy of a prescriptive approach is heavily focused on controlling the processes that are used to develop the software. Generally the processes that are specified in prescriptive standards are very sensible, and the software evidence produced may be of a high level of integrity. The approach relies, however, on the assertion that the processes used, and the evidence generated as a result of following those processes, are adequate to sufficiently control the contribution of the software to the system hazards. This relationship between the prescribed processes and the system hazards is generally tenuous and always implicit. It is generally fairly easy with all prescriptive approaches

to conceive of a situation where the prescribed processes have been followed, but there remain software contributions to hazards which are not sufficiently controlled. This is discussed in more detail in reference 4.

In contrast to this, a goal-based approach does not rely solely on controlling the processes that are used, but instead focuses directly on controlling the contribution of the software to system hazards through the construction of an explicit software safety argument. When using a prescriptive approach there exists an implicit argument (that following the prescribed process will result in an acceptably safe system). By generating an explicit software safety argument, the way in which the evidence supports the objectives of the safety case for the particular system under consideration becomes clear. The development of a software safety argument relies upon the developer of the software to determine the most appropriate way to demonstrate the safety of the software they are developing. A prescriptive approach relies upon the regulatory authority determining the most appropriate way to demonstrate the safety of any system within their domain. Clearly the developer of the software itself is normally the most appropriate person to determine what should be done for their system. The most appropriate role for the regulatory authority is to assess whether what the developer has done is sufficient for their system. A software safety argument approach supports these roles. One further advantage of adopting a goal-based approach is that, because it is not prescriptive about the methods and techniques that should be adopted, it facilitates the use of new approaches and technologies that could bring increased capability and efficiency.

Despite the potential advantages discussed above, our investigations have highlighted that constructing safety arguments for the software aspects of systems (software safety arguments) is challenging. When following a prescriptive approach, the developer of the software knows clearly from the outset the processes that must be followed, and the techniques that must be used. This helps with the planning and management of the development project. In contrast to this, when adopting a safety argument based approach, the necessary activities and processes are not specified up front. Instead the high level objectives are specified, and the developer must determine what techniques and evidence are necessary and sufficient to construct a compelling safety argument. Identifying what evidence will be sufficient to demonstrate that the contribution of the software to the safety of the system is acceptable is a major challenge. In this paper we propose a systematic approach to the development of software safety arguments which begins to address this.

Software safety arguments

As discussed in reference 5, all arguments can be split into two types, deductive and inductive. Deductive arguments are those where if the premises of the argument are true, then the conclusion must also be true. In contrast, an inductive argument is one in which the conclusion of the argument follows from the premises not with absolute certainty, but only with a level of confidence. It is more common to see software safety arguments which are inductive in nature. In order for a software safety argument to be compelling, it is therefore necessary to provide and demonstrate a sufficient level of confidence in the safety claims made. When considering the safety of software, it is common to use the term *assurance*. The assurance of a safety claim is simply the justifiable confidence in that claim.

There are a number of different factors which affect the assurance of software safety claims. Firstly, software failures are systematic in nature - mainly due to errors made in the specification or design. Therefore, unlike random failures, it is extremely difficult to predict when a failure may occur, or what the nature of that failure may be. This places an important limitation on the confidence that can be achieved in a safety claim, which requires knowledge of both how likely a failure is, and also whether that failure may affect the claim being made. It is often necessary to make certain

assumptions relating to a claim, for example, depending on the nature of the claim made, it may be necessary to make assumptions about the independence of different aspects of the system, or the suitability of a particular approach. All assumptions are, by definition, unsupported. Assumptions are taken to be true, and the argument holds on the basis that the assumptions are true. Consequently, if there is a lack of confidence in the truth of the assumptions, then this will result in uncertainty in the truth of the claim as well. In addition, it is never possible to have complete knowledge about the system under consideration and the environment in which that systems will be used. Without this information it is difficult to gain confidence in the claims being made, since it is hard to know how strongly the available evidence supports the claim. The assurance of a safety claim may also be affected by how strongly the sub-claims or evidence give reason to believe that the safety claim is true. Different types of

argument and evidence are more compelling in their support of different types of software safety claim. The trustworthiness (quality) of the evidence itself may also affect the confidence in the claim. Even if a safety claim is strongly supported by an item of evidence, if that evidence is untrustworthy, then the confidence provided in the safety claim will be reduced. It is important to note that all of the uncertainties discussed above are present also in a prescriptive approach, however they are generally left implicit. A safety case approach enables an explicit consideration of the uncertainty to be provided.

In order to be compelling, a software safety argument must demonstrate that sufficient assurance is achieved in the safety claims. The uncertainties in the argument, such as those discussed above, have the potential to undermine the assurance achieved. We describe such uncertainties as *assurance deficits*. A compelling argument must demonstrate that any residual assurance deficits are acceptable, that is that the impact of the assurance deficit can be justified. It is necessary to consider, attempt to address, or justify all the potential assurance deficits in the software safety argument. This requires that throughout the development of the argument the assurance achieved in the safety claims from the argument and evidence provided is explicitly considered.

We propose a systematic approach to software safety argument construct which explicitly considers assurance throughout the development of the argument. This helps to ensure that the resulting software safety argument is sufficiently compelling. The approach we have developed has two parts. Firstly, we have developed a software safety argument pattern catalogue, which suggests argument structures for compelling software safety arguments which can be instantiated for the target system. Secondly, we propose a software safety argument development method which explicitly considers how assurance may be affected at each step in the argument development. We propose that these two parts, when used together, provide a method for developing software safety arguments which are sufficiently compelling.

Software Safety Argument Patterns

Software safety argument patterns provide a way of capturing good practice in software safety arguments. Patterns are widely used within software engineering as a way of abstracting the fundamental design strategies from the details of particular designs. The use of patterns as a way of documenting and reusing successful safety argument structures was pioneered by Kelly in reference 6. As with software design, software safety argument patterns can be used to abstract the fundamental argument strategies from the details of a particular argument. It is then possible to use the patterns to create specific arguments by instantiating the patterns in a manner appropriate to the application. One approach to representing safety arguments is the Goal Structuring Notation (GSN) (ref. 6). The basic GSN symbols are shown in Figure 1.

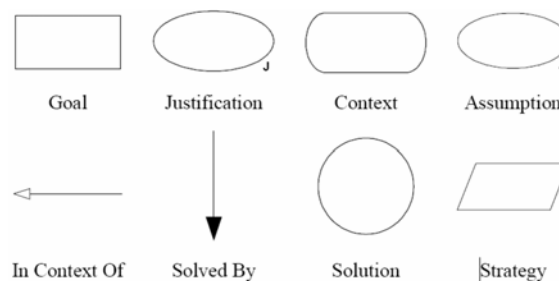
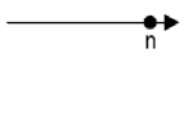
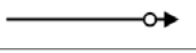



Figure 1 - Main elements of the GSN notation

These symbols can be used to construct an argument by showing how claims (goals) are broken down into sub-claims, until eventually they can be supported by evidence (solutions). The strategies adopted, and the rationale (assumptions and justifications) can be captured, along with the context in which the goals are stated. Kelly proposes extensions to GSN that can be used to support the abstractions necessary to capture patterns of argument. To create patterns, GSN is extended to support multiplicity, optionality and abstraction. The multiplicity extensions shown in figure 2 are used to describe how many instances of one entity relate to another entity. They are annotations on existing GSN relational arrows. The optionality extension is used to denote possible alternative

support. It can represent a 1-of-n or an m-of-n choice. In figure 2, one source node has three possible alternative sink nodes.

	A solid ball is the symbol for many (meaning zero or more). The label next to the ball indicates the cardinality of the relationship.
	A hollow ball indicates "optional" (meaning zero or one).
	A line without multiplicity symbols indicates a one to one relationship (as in conventional GSN).

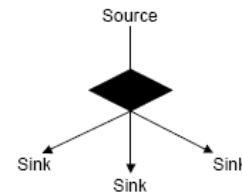


Figure 2 - GSN multiplicity and optionality extensions

The abstraction extensions shown in figure 3 allow GSN elements to be generalised for future instantiation. The uninstantiated entity placeholder denotes that the attached element remains to be instantiated, i.e. at some later stage the abstract entity needs to be replaced with a more concrete instance. The undeveloped entity placeholder denotes that the attached element requires further development, i.e. at some later stage the entity needs to be decomposed and supported by further argument and evidence.



Figure 3 - GSN abstraction extensions

Kelly also introduced modular extensions to GSN (ref. 7), modular safety cases provide a means of organising large or complex safety cases into separate but interrelated component modules of argument and evidence. When splitting an argument into modules it becomes necessary to be able to refer to goals which exist within other modules. To refer to goals in other modules, the GSN element "Away Goal" is used. Each away goal contains a module identifier, which is a reference to the module where the goal can be found. Away goals can be used as a way of providing support for a goal in one module, with a goal in another module and can also be used to provide contextual backing for goals, strategies and solutions (see goals DSSRidentify and hazCont in Figure 4).

There exist a number of examples of safety argument patterns. Kelly himself developed an example safety case pattern catalogue in reference 6 which provided a number of generic solutions identified from existing safety cases. Although providing a number of useful generic argument strategies, Kelly acknowledges that this catalogue does not provide a complete set of patterns for developing a safety argument, it merely represents a cross-section of useful solutions for unconnected parts of arguments. Kelly's pattern catalogue does not deal specifically with any software aspects of the system. The safety argument pattern approach was further developed by Weaver (ref. 8), who specifically developed a safety pattern catalogue for software. The crucial differences with this catalogue were firstly that the set of patterns in the catalogue were specifically designed to connect together in order to form a coherent argument. Secondly the argument patterns were developed specifically to deal with the software aspects of the system. There are a number of weaknesses that have been identified with Weaver's pattern catalogue. Firstly, the argument patterns take a fairly narrow view, focusing on the mitigation of failure modes in the design. Secondly, the patterns present an essentially "one size fits all" approach, with little guidance on alternative strategies, or how the most appropriate option is determined. A software safety pattern catalogue has also been developed by Ye (ref. 9), specifically to consider arguments about the safety of systems including COTS software products. Ye's patterns provide some interesting developments to Weaver's, including patterns for arguing that the evidence is adequate for the assurance level of the claim it is supporting. Although we do not necessarily advocate the use of discrete levels of assurance, the patterns are useful as they support the approach of arguing over both the trustworthiness of the evidence and the extent to which that evidence supports the truth of the claim.

The software safety argument pattern catalogue discussed in this paper builds upon this existing work, and also takes account of current good practice for software safety, including from existing standards. A primary consideration during the development of these patterns has been flexibility and the elimination of system-specific concerns and terminology. Consequently, these patterns can be instantiated for a wide range of systems and under a variety of circumstances. It is therefore crucial to make the correct decisions when instantiating these patterns, in order that the resulting argument be considered sufficiently compelling. It is for this reason that the instantiation of the patterns for a particular system must always be carried out within the framework of the assurance based argument development method discussed later. This ensures that sufficient assurance is achieved from the application of the patterns.

The software safety argument pattern catalogue contains a number of patterns which may be used together in order to construct a software safety argument for the system under consideration. The following argument patterns are currently provided:

1. High-level software safety argument pattern – This pattern provides the high-level structure for a generic software safety argument. The pattern can be used to create the high level structure of a software safety argument either as a stand alone argument or as part of a system safety argument.
2. Software contribution safety argument pattern - This pattern provides the generic structure for an argument that the contributions made by software to system hazards are acceptably managed. This pattern is based upon a generic ‘tiered’ development model in order to make it generally applicable to a broad range of development processes.
3. Derived Software Safety Requirements identification pattern - This pattern provides the generic structure for an argument that derived software safety requirements (DSSRs) are adequately captured at all levels of software development.
4. Hazardous contribution software safety argument pattern – This pattern provides the generic structure for an argument that the identified DSSRs at each level of software safety development adequately address all identified potential hazardous failures.
5. Strategy justification software safety argument pattern - This pattern provides the generic structure for an argument that the strategy which is adopted in a software safety argument is acceptable given the confidence that is required to be achieved in the relevant claim.

When instantiated for the target system, these patterns link together to form a single software safety argument for the software. The high-level software safety argument pattern can be used to create the high level structure of the argument defining the overall objectives of the software safety argument. The high-level argument pattern provides claims that the software is acceptably safe to operate in the defined system. This is supported by arguing that the contribution made by the software to system level hazards is acceptable. It is important that explicit traceability is maintained between the system level hazards (which ultimately must be controlled for the system to be considered safe) and the behaviour of the software. At the high level of the argument, the design of the software itself is not considered, the software is considered as a black box, and the contribution to the system hazard is identified from consideration of the system level functionality in which the software is involved. Such high-level software contributions may be identified, for example, as the base event in a system fault tree for the hazard of interest.

In this paper it is not possible to provide full details on each of these argument patterns, instead we highlight one of the patterns which is of particular importance both to the overall structure of any resulting argument, and to the assurance achieved in the safety of the software – the software contribution safety argument pattern. Figure 4 shows the structure of this pattern represented using the GSN pattern notation introduced earlier.

This pattern provides the structure for arguments that the contributions made by software to system hazards, which were identified in the high-level software safety argument pattern, are acceptably managed. It is at this point in the argument that the software design is considered in detail. It was discussed earlier how the patterns have been

constructed to be as flexible as possible such that they are applicable to a wide range of systems. There are a wide range of different development processes used on different projects, and it is important that the argument pattern may be instantiated no matter what development process is used. The structure of the pattern is therefore based upon a generalized ‘tier’ model of development such as that proposed in reference 10. Each tier corresponds to one level of decomposition of the design. The number of tiers of development may be different for different software systems, but the general safety considerations at each tier are unchanged. In addition, different parts of the design of any software system may be decomposed over a different number of tiers. Note that the term ‘tier’ is used principally to avoid the potential confusion of overloading the term ‘level’.

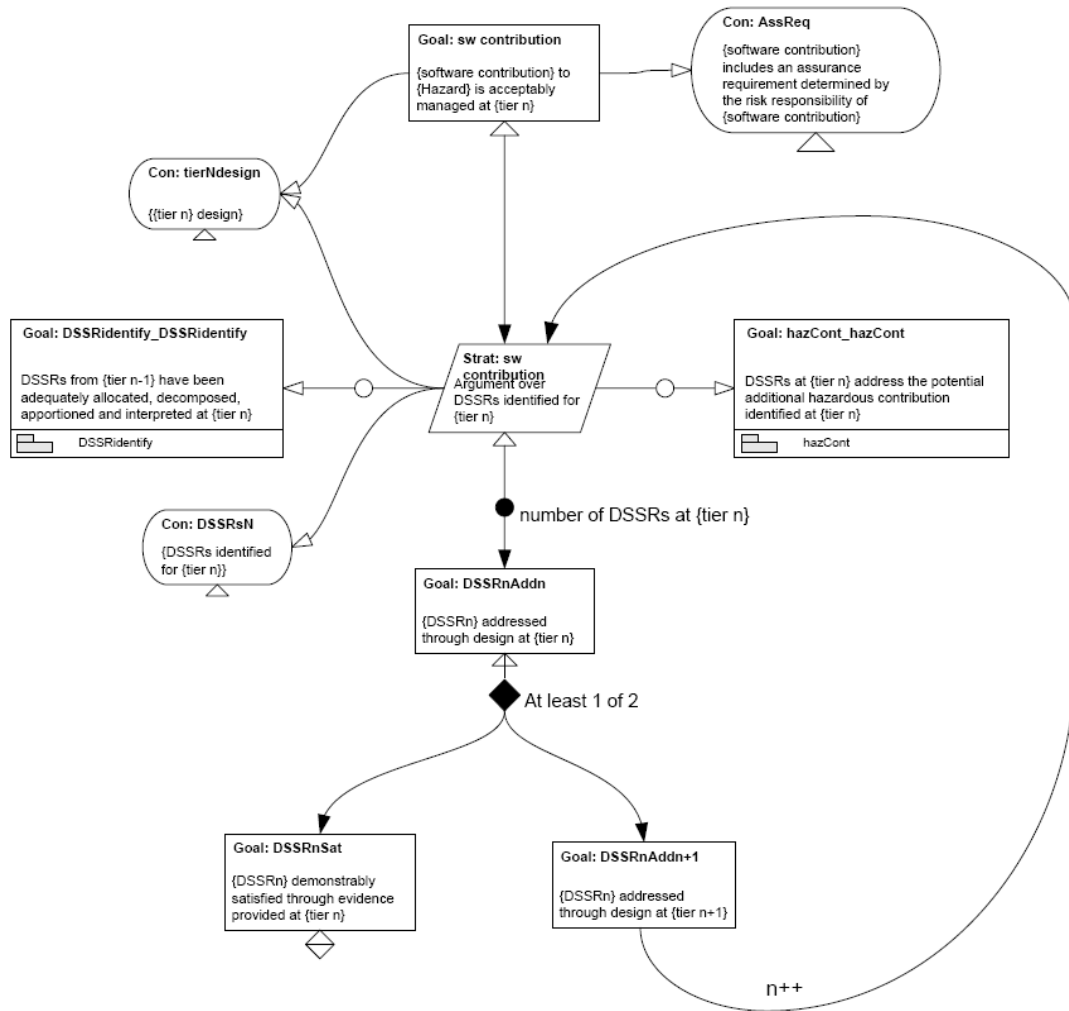


Figure 4 - The structure of the software contribution safety argument pattern

It should be noted when instantiating the pattern shown in Figure 4 that {tier n}, and {tier n+1} etc. must be instantiated with the names of the relevant tier as appropriate for the target system (e.g. class design, high level design, etc.). The term DSSR refers to derived software safety requirements. These are the set of safety requirements which the software must satisfy at each tier. In the pattern the term DSSRn is used to refer to a DSSR at tier n, and should be instantiated with the DSSR itself or a unique identifier for the DSSR.

The starting point for this argument pattern is to make a safety claim relating to each of the potential software contributions identified at the high level of the argument. To make a compelling safety argument for the software, it is important for each contribution that all the ways in which errors may be introduced into the software which could lead to that contribution are considered. At each tier in the development of the software, it is necessary to address

the safety requirements from the previous tier, that is at each tier it is necessary to ensure that the software is designed such that it will meet its safety requirements. By ensuring that each tier meets the safety requirements imposed by the previous tier, it is ensured that there is traceability of safety requirements up through the tiers of development to the system hazard to which the software may contribute. It can be seen in the argument pattern in Figure 4 that there a choice of two ways in which this can be achieved. At each tier it is possible to provide evidence at that tier that the safety requirements are satisfied. In addition to this the safety requirements can be traced through to the next tier. At that point (under Goal: DSSR_nAdd_{n+1}), the pattern returns to repeat the pattern of argument for the next tier (tier $n+1$ becomes tier n). It should be noted that in the instantiated argument, this relationship would not appear as a 'loop', but as a single hierarchical structure, with an instances of each goal created for the relevant tier.

It is possible to introduce errors into the software at every tier of development as decomposition of the design occurs. The argument put forward must take the impact of these errors into account as they can undermine the adopted strategy if not addressed correctly. The potential errors at each tier are addressed through the arguments provided in context to the main strategy (the contextual backing for the strategy). The first argument (Goal: DSSRIdentify) demonstrates that the safety requirements from the previous tier have been adequately allocated, decomposed, apportioned and interpreted for the current tier. It may be possible to achieve this through implementing design decisions for the current tier which mitigate the safety requirement form the previous tier. For example, it may be possible to include detection and handling mechanisms into the tier design which address potential safety requirement breaches. It may also be possible, for example, to design the tier to prevent interference between components. In addition to this it is necessary, for any requirements that aren't fully mitigate through the design at that tier to use the tier design information to specify safety requirements which ensure the safety requirements of the previous tier are met. This may require the definition of one or more new safety requirements upon the components in the tier design. The derived software safety requirements identification pattern has been created to provide the structure of such an argument. At each tier of software development it is possible to introduce errors into the software which could manifest themselves as hazardous failures. The second argument providing context to the main strategy (Goal: hazCont) considers the additional hazardous contributions that may be introduced at each tier. The hazardous contribution software safety argument pattern has been created to provide the structure of such an argument.

Assurance Based Argument Development Method

Even when using the argument patterns described in the previous section to develop the software safety argument for a system, this doesn't necessarily guarantee that the resulting argument will be sufficiently compelling. We discussed earlier how the assurance achieved in the argument can be undermined by the presence of assurance deficits. It is possible that assurance deficits may be introduced into the argument as it is being developed. These assurance deficits will undermine the assurance that is achieved. To ensure that the argument is sufficiently compelling (that sufficient assurance is achieved), it is necessary to manage these assurance deficits. This requires that the assurance deficits are explicitly identified throughout the development of the argument.

There exists a general, and widely used method for constructing and defining arguments. This method was developed by Kelly and is often referred to as the 'six-step method'. The method is described in detail in reference 6. Each of the steps in the process is listed in the first column of Table 1. Steps 1 to 5 are applied cyclically to create the hierarchical structure of the argument. This continues until such a point as evidence may be provided to support the goals. At this point step 6 is applied, and development of that leg of the argument stops. It is possible to apply this existing method to develop a software safety argument, however following such a method alone does not guarantee that the resulting argument will be sufficiently compelling. Instead it is necessary to extend this method in order to explicitly consider assurance at each step by identifying how assurance deficits may be introduced.

In order to achieve this a deviation-style analysis of each of the six steps was performed. This considered the purpose of each of the steps, and then considered the ways in which assurance deficits may be introduced at that step. This deviation analysis is based on the widely-used HAZOP technique, which was originally developed as a way of analysing process plants (ref. 11) but has since been developed for use in other applications including the analysis of software (ref. 12). HAZOP uses a set of guidewords to prompt the identification of deviations from

normal behaviour. The standard HAZOP guidewords are: no or none, more, less, as well as, part of, other than, reverse.

In Table 1 we apply and interpret the HAZOP guidewords for each step in the six-step argument development method to consider the ways in which assurance may be affected. Only those guidewords with a meaningful interpretation are considered for a particular step.

Table 1 - Consideration of Assurance During Argument Construction

Step	Purpose	Assurance impact			
1. Identify goals to be supported	To clearly and unambiguously state the goals to be supported.	More - If in stating the goal, an attempt is made to claim more than it is actually possible to support with the available evidence, then the assurance that can be achieved in that goal will inevitably be low.	Less - The stated goal may claim less than is actually required to support the argument. Although in this case it may be easier to achieve higher confidence in the stated goal, this confidence will not result in the expected assurance in the parent goal, since the claim is insufficient to support the conclusion.	As Well As - A strategy or solution may be erroneously included in the claim. This can inadvertently constrain potential options for addressing assurance deficits.	Other Than - The claim made may not actually be that in which assurance is required. Assurance may be lost through failing to correctly capture the true intent of the claim.
2. Define basis on which goals are stated	To clarify the scope of the claim, to provide definitions of terms used, to interpret the meaning of concepts.	None - Any claim is only true or false over a particular scope. If the scope of the claim is unclear, due to lack of context, then the level of truth or falsity of the claim becomes more difficult to determine. This increases the uncertainty associated with the assurance in that claim, and therefore makes it more difficult to determine the assurance.	More - The scope of the claim as defined by the context may be too narrow. The result of this is that although a certain level of assurance may be achieved over the scope defined by the context, the narrowness of the scope limits that in which confidence is achieved.	Less - The scope of the claim is too loosely defined. The effect of this would be similar to having no context at all, in that it leads to uncertainty, and a corresponding reduction in assurance.	
3. Identify strategy to support goals	To identify how a goal can be adequately supported.	More, Less, Other Than - This step of the safety argument process is the most crucial for the assurance achieved since it is at this step that the decisions are made about which strategy should be adopted to support each claim. Assurance is lost at this step if the proposed strategy does not provide sufficient support to the goal. This could happen for two reasons. <ul style="list-style-type: none"> • Firstly the inductive gap may be too large. If this is the case, then even if the premises are believed, it doesn't provide sufficient confidence in the truth of the conclusion. • Secondly the fundamental beliefs upon which the strategy is based may be open to question. In such a case the premises may not provide confidence in the conclusion. 			
4. Define basis on which strategy is stated	To identify any assumptions upon which the sufficiency of the strategy depends.	No, Less - It is inevitable that some assumptions will be made during the development of any safety argument, however these assumptions may not always be explicitly captured. Any assumptions that are left implicit introduce uncertainty, and reduce assurance.	More - All assumptions are, by definition, unsupported. The argument holds only on the basis that the assumptions are true. If there is a lack of confidence in the truth of the assumptions, then this will also result in a lack of confidence in the truth of the claim. It is therefore recommended, for any assumptions that may be open to any significant doubt, that an argument is presented, rather than an assumption.	Other Than - Assumptions may be stated which are not actually true. Any false assumptions undermine the whole basis upon which the argument is made.	
	To provide justification for why a particular strategy is being used.	No, Less - No justification is provided as to why the adopted strategy is sufficient. This can result in a loss of assurance, since there may be	More - Although not leading to a loss of assurance, it is important to note that providing an argument to justify the strategy chosen in		

		a lack of confidence in the sufficiency of that strategy. It is important, if it's likely that the justification may be unclear, not to leave it implicit, but to explicitly record the justification in the argument.	each decomposition in the argument is not necessary. For many strategies, the justification will be obvious to the reader and may be left implicit.		
5.Elaborate strategy	Specify the goals that implement the chosen strategy.	Less, As Well As, Part Of - The strategy that is actually implemented does not fully and accurately reflect the one that was chosen. Assurance may be lost at this step, even though a chosen strategy may be considered acceptable.			
6. Identify basic solution	Identify the solutions which provide adequate support to the goal.	Less - The solution provides less confidence in the goal being supported than is required. Assurance is lost at this step if it is unclear why the evidence gives confidence in the goal being supported. It may be unclear because: there may be an inductive gap between the claim and the evidence (the nature of the evidence does not provide a compelling reason to believe the claim is true) there is uncertainty about the trustworthiness of the evidence itself. Note that evidence which is untrustworthy will undermine assurance even in the situation where there is a deductive relationship between the claim and the evidence.	Other than - Counter evidence is any evidence which undermines the confidence in the claim being made. The presence of counterevidence does not necessarily mean that the argument is inadequate. It simply means that the confidence in the claim may now be lower than it was before the counter evidence was identified. It is necessary to determine the impact of the counter evidence on the claim's assurance. In many cases it may still be possible to make a sufficiently compelling argument despite the identification of counter evidence, particularly where there are mitigations which limit the uncertainty caused by the counter evidence.		

It is not realistically possible to remove all assurance deficits from the argument produced. The amount of information relevant to the argument being made is simply too large. For the argument to be sufficiently compelling however, it is not necessary to remove all assurance deficits. Instead it is necessary to be able to justify that any residual assurance deficits are acceptable. The acceptability of an assurance deficit will depend upon the impact of the deficit on the overall safety of the system. The impact of an assurance deficit will be specific to the system under consideration, and must consider the effect of the assurance deficit on system risk. It is possible to provide an argument to justify the acceptability of residual assurance deficits. In other work, the authors are currently developing an approach to justifying the acceptability of assurance deficits, this is not developed further in this paper.

As the patterns capture good practice for argument construction, they can themselves be used to identify assurance deficits in an argument. Any elements from the software safety arguments that are not reflected in the argument produced for the system are potential assurance deficits and must be considered and justified. When instantiating the patterns, the assurance based development discussed above must be applied to ensure that justifiable instantiation decisions are made.

Conclusions

In this paper we have described a systematic approach for developing compelling software safety arguments. Our approach is based upon two elements. A catalogue of software safety argument patterns has been developed, which capture good practice for structuring software safety arguments. Also, an assurance based development method has been proposed, which explicitly considers how assurance deficits may be introduced into the safety argument. By using these two elements together, it is possible to develop compelling safety arguments for the software aspects of systems.

Acknowledgments

The authors would like to thank the U.K. Ministry of Defence for their support and funding. This work is undertaken as part of the research activity within the Software Systems Engineering Initiative (SSEI), www.ssei.org.uk.

References

1. MoD. Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems. HMSO, 2007.
2. RTCA. DO-178B - Software Considerations in Airborne Systems and Equipment Certification. Radio and Technical Commission for Aeronautics, 1992.
3. IEC. 61508 - Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems. International Electrotechnical Commission, 1998.
4. J. A. McDermid. Software safety: Where's the evidence? In Australian Workshop on Industrial Experience with Safety Critical Systems and Software, 2001.
5. J. Baggini and P.S. Fosl. The Philosopher's Toolkit - A Compendium of Philosophical Concepts and Methods. Blackwell, 2003.
6. Tim Kelly. Arguing Safety - A Systematic Approach to Managing Safety Cases. PhD thesis, Department of Computer Science, The University of York, 1998.
7. Tim Kelly. Concepts and principles of compositional safety case construction. Technical Report COMSA/2001/1/1, The University of York, 2001.
8. R. A. Weaver. The safety of Software - Constructing and Assuring Arguments. PhD thesis, Department of Computer Science, The University of York, 2003.
9. Fan Ye. Justifying the Use of COTS Components within Safety Critical Applications. PhD thesis, Department of Computer Science, The University of York, 2005.
10. M.S. Jaffe, R. Busser, D. Daniels, H. Delseny, and G. Romanski. Progress report on some proposed upgrades to the conceptual underpinnings of DO-178B/ED-12B. In Proceedings of the 3rd IET International Conference on System Safety, 2008.
11. CISHEC. A Guide to Hazard and Operability Studies. The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd., 1977.
12. F. Redmill, Morris Chudleigh, and James Catmur. System Safety: HAZOP and Software HAZOP. Wiley, 1999.

Biography

R.D. Hawkins, Ph.D., Department of Computer Science, The University of York, York, YO10 5DD, UK, telephone +44 (0) 1904 567836, e-mail – richard.hawkins@cs.york.ac.uk.

Dr. Richard Hawkins is a research associate for the MoD Software Systems Engineering Initiative at the University of York. His current research is developing guidance on the construction of software safety arguments, and developing an approach for reasoning about argument and evidence assurance. He has previously worked as a software safety engineer for BAE Systems and was involved in research into modular and incremental certification as part of the Industrial Avionics Working Group (IAWG).

T.P. Kelly, Ph.D., Department of Computer Science, The University of York, York, YO10 5DD, UK, telephone +44(0) 01904 432764, facsimile +44 (0)1904 432708, e-mail – tim.kelly@cs.york.ac.uk.

Dr Tim Kelly is a senior lecturer in the Department of Computer Science at the University of York. His main area of research is the development and justification of complex (computer based) safety-critical systems. He has provided extensive consultative and facilitative support in the production of acceptable safety cases for companies from the medical, aerospace, railway and power generation sectors. He has published over 60 papers on high-integrity systems engineering in international journals and conferences.