Software Safety Assurance – What Is Sufficient?

R.D. Hawkins, T.P. Kelly

Department of Computer Science,
The University of York, York, YO10 5DD
UK

Keywords: Software, Assurance, Arguments, Patterns.

Abstract

It is possible to construct a safety argument for the software aspects of a system in order to demonstrate that the software is acceptably safe to operate. In order to be compelling, it is necessary to justify that the arguments and evidence presented for the software provide sufficient safety assurance. In this paper we consider how assurance may be explicitly considered when developing a software safety argument. We propose a framework for making and justifying decisions about the arguments and evidence required to assure the safety of the software.

1 Introduction

A safety case can be used to demonstrate that a system is acceptably safe to operate. A safety case should contain a structured argument demonstrating how safety claims are supported by a body of evidence. For systems containing software, a safety argument must consider claims that the contribution of the software to the safety of the system is acceptable.

In order to assure the safety of software it is common to adopt a highly prescriptive approach, where safety is demonstrated by showing compliance with the requirements set out as a prescribed process in a standard. Such requirements are generally varied to reflect the criticality or importance of the safety function being performed by the software. This approach is the basis of commonly adopted standards such as IEC 61508 [3] and DO-178B [8].

A software safety argument makes it possible to provide an explicit demonstration that the evidence generated supports the specific safety objectives of the system. Where a prescribed process has been followed, the software safety argument may therefore highlight requirements for additional evidence.

Constructing compelling software safety arguments remains, however, a major challenge. In particular, justifying the sufficiency of the arguments and evidence provided for the software is difficult. In this paper we describe an approach which begins to provide a framework for making and justifying decisions about the arguments and evidence required to assure the safety of the software. We begin by considering the challenges of software safety assurance.

2 Software Safety Assurance

It is inevitable for the software aspects of a system that there will exist inherent uncertainties that affect the assurance with which it is possible to demonstrate the safety of the software. The reason for this is that the amount of information potentially relevant to demonstrating the safety of the system is vast. This may be information relating to the software itself, or to the system within which the software operates. There will also be information relating to the environment and operation of the system, all of which potentially has a role in demonstrating that the software is acceptably safe. It is simply not possible to have complete knowledge about the safety of the software.

This leads to uncertainty, for example through having to make assumptions or accept known limitations in the integrity of the evidence generated, or the strength of support that evidence provides. For this reason it is not normally possible to demonstrate with absolute certainty that the claims made in a software safety argument are true. For a software safety argument to be compelling it must instead establish *sufficient confidence* in the truth of the safety claims that are made. The assurance of a claim is the justified confidence in that claim.

It is worth noting at this point that such uncertainties in demonstrating the safety of the software are always present, but are often, such as when following a highly prescriptive approach, left implicit. The construction of a software safety argument facilitates the explicit identification of such uncertainties, making them easier to reason about, and therefore justify. Reasoning explicitly about the extent and impact of the uncertainties in a safety argument aids in the successful acceptance of the argument as part of a safety case. Any identified residual uncertainty in demonstrating the safety of the software (such as those discussed above) can be considered to be an *assurance deficit*. Assurance deficits can reduce the assurance which is achieved. It is possible to use the construction of a software safety argument to identify how assurance deficits may arise. This is achieved by systematically considering how uncertainty may be introduced at each step in the construction of the argument. We discuss this further in Section 3

By identifying where potential assurance deficits may arise, this approach can be used to inform the decisions that are made on how to construct the argument. We consider how to structure a software safety argument in Section 4

In order to produce a sufficiently compelling software safety argument, all identified assurance deficits must be satisfactorily addressed, or justification must be provided that the impact of the assurance deficit on the claimed safety of the system is acceptable. Section 5 discusses how such justifications might be made.

3 Considering Assurance During Argument Construction

There exists a widely used method for constructing and defining safety arguments, often referred to as the 'six-step method' [4]. The steps of the method are:

1. Identify goals (claims) to be supported
2. Define basis on which goals (claims) are stated
3. Identify strategy (argument approach) to support goals (claims)
4. Define basis on which the strategy (argument approach) is stated
5. Elaborate the strategy (argument)
6. Identify basic solution (evidence)

Steps 1 to 5 are applied cyclically to create the hierarchical structure of claims and sub-claims of the argument. This continues until it is possible to identify evidence to support the claim. At this point step 6 is applied, and development of that leg of the argument stops.

Although this six-step method has been used successfully to develop many different safety arguments, this approach doesn't explicitly consider assurance. This means that the sufficiency of the resulting argument may be difficult to justify. For software safety arguments a more systematic consideration of assurance is required.

As an argument is constructed, decisions are continually being made about the best way in which to proceed. Decisions are made about how goals are stated, the strategies that are going to be adopted, the context and assumptions that are going to be required, and the evidence it is necessary to provide. Each of these decisions has an influence on what is, and is not, addressed by the safety case. The things that are not sufficiently addressed are referred to as assurance deficits.

To extend the existing six-step method to explicitly identify assurance deficits, the potential ways in which assurance may be lost at each of the steps in the method must be considered. 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 uncertainty may be introduced into the argument at that step. This deviation analysis was based upon the widely used HAZOP technique which was originally developed as a way of analysing process plants [1] but has since been developed for use in other applications including the analysis of software [8]. 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.

The HAZOP guidewords were applied and interpreted for each step in the six-step argument development method to consider the assurance deficits that may arise. An example taken from a summary of the results of the analysis is provided in Table 1. By helping to identify where assurance deficits may arise, this approach can be used to inform the decisions that are made on how to construct an argument.

On its own however, such guidance is insufficient, as it provides no specific guidance on the argument structure or nature of the claims required for a software safety argument. This is considered in the next section.

4 Software Safety Argument Patterns

Software safety argument patterns provide a means 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 [4]. As with software design, software safety argument patterns can be used to abstract fundamental argument approaches 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.

There exist a number of examples of safety argument patterns. Kelly developed an example safety case pattern catalogue in [4] which provided a number of generic solutions identified from existing safety cases. Although providing a number of useful generic argument strategies, the author 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 [9], who specifically developed a safety pattern catalogue for software. There were two crucial differences with this catalogue. Firstly, 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 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 [10], 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.

4.1 A Software Safety Argument Pattern Catalogue

We have developed a catalogue of software safety arguments which builds upon the existing work, and also takes account of current good practice for software safety, including from existing standards. 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 broader 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. **Software Safety Requirements identification pattern** - This pattern provides the generic structure for an argument that software safety requirements (SSRs) 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 SSRs at each level of software safety development adequately address all identified potential hazardous failures.
5. **Argument justification software safety argument pattern** - This pattern provides the generic structure for an argument that the software safety argument presented is sufficient.

The argument patterns are captured using the pattern extensions to the Goal Structuring Notation (GSN), described in [4]. When instantiated for the target system, these patterns link together to form a single software safety argument for the software. Here we provide a brief overview of the patterns, full details of the patterns can be viewed at [11]. Key to these arguments is establishing the satisfaction of software safety requirements (SSRs) and the absence of hazardous errors through tiers of development of the software. The number of tiers of development may be different for different software systems, but the general safety considerations at each tier are unchanged.

At the heart of the pattern catalogue is the *software contribution safety argument pattern* shown in Figure 1. This pattern provides the structure for arguments that the contributions made by software to system hazards are acceptably managed. It is at this point in the overall software argument that the software design is considered in detail. The main 'spine' of the argument argues that, at each tier, the SSRs imposed upon the design are met. As can be seen in Figure 1, this can be demonstrated in two ways. Firstly, at each tier it is possible to provide evidence at that tier that the SSRs are satisfied. Secondly, the SSRs may be traced through to the next tier of design. This ensures that traceability is established up through the tiers of development to the system hazards to which the software may contribute.

However, such an argument on its own would be insufficient. In Figure 1 it can be seen that two other software safety argument patterns from the catalogue (SSR identification and hazardous contribution) support this main thread of argument. These patterns consider that it is possible, at any tier of development, to introduce errors into the software as decomposition of the design occurs. The *SSR identification pattern* argues that, at each tier, the SSRs from the previous tier have been adequately allocated, decomposed, apportioned and interpreted. The pattern provides two ways in which it can be argued that this is achieved. Firstly, it can be argued that design decisions that are taken will help to mitigate SSRs. For example, a decision may be taken to have redundant components in the design in order to help satisfy an SSR relating to the availability of an item of data. Secondly, it may be necessary, taking into account the design at that tier, to specify new or additional SSRs upon the components in the design.

The *hazardous contribution pattern* considers possible hazardous failures that may manifest themselves at each tier of software development. There are two aspects to this argument. Firstly the argument must consider at each tier the possible failure behaviour of the software. An argument is provided that such potentially hazardous behaviour is identified at each tier, and that appropriate SSRs have been defined that are sufficient to address them. There are various techniques available for identifying deviations from intended behaviour in software designs (such as Software HAZOP [7]). The particular technique which is most appropriate to use will depend upon the tier being considered, and also upon the nature of the software design itself. Secondly the argument must also consider the possibility that hazardous errors are introduced by the design process adopted at that tier. It is necessary to recognise in the argument that the design process at any tier may be flawed. Although this aspect of the argument is principally concerned with those errors in the design which may lead to hazardous behaviour, the argument is likely to have to involve thinking more generally about how errors are removed from the design. This might include

consideration of the integrity of models generated, or the robustness of languages used to specify the software.

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 crucial to make the correct decisions when instantiating these patterns for a particular system, in order that the resulting argument be considered sufficiently compelling. Making incorrect instantiation decisions when constructing the argument can result in assurance deficits. The argument development approach discussed in section 3 should always therefore be used when instantiating the patterns for a particular system to ensure that assurance is considered throughout instantiation. To be compelling it is necessary to be able to justify that the instantiation decisions taken result in a sufficiently compelling argument for the system under consideration (such as why particular claims are chosen whilst others are not required). Guidance for justifying such decisions is provided in the next section.

5 Justifying Assurance Sufficiency

The discussions in sections 3 and 4 illustrated how assurance deficits may be systematically identified throughout the construction of a software safety argument. The existence of identified assurance deficits raises questions concerning the sufficiency of the argument. Therefore where an assurance deficit is identified, it is necessary to demonstrate that the deficit is either acceptable, or addressed such that it becomes acceptable (for example through the generation of additional relevant information). There will typically be a cost associated with obtaining the information to address an assurance deficit. In theory it would be possible to spend disproportionate sums of money generating sufficient information to address all assurance deficits. However in practice the benefit gained from addressing each assurance deficit will not necessarily justify the cost involved in generating the additional information. In order to assess if the required level of expenditure is warranted, the impact of that assurance deficit on the claimed risk position of the argument must be determined. Firstly we therefore discuss how the impact of an assurance deficit can be assessed.

5.1 Assurance Deficit Impact

The software safety argument establishes a claimed position on the hazard identification, risk estimation, and risk management of the software contribution to system hazards. Since assurance deficits have the potential to undermine the sufficiency of the argument, the impact of any assurance deficit should be assessed in terms of the impact it may have on this claimed position. Is the assurance deficit significant enough that that position can no longer be supported? For example, an assurance deficit may be sufficient to challenge the completeness of hazard identification, or may be sufficient to challenge the estimated residual risk. It may also be possible, for example, that an assurance deficit challenges a claim that the software contribution to system hazards are acceptably managed.

One of the challenges of determining the impact of an assurance deficit is that the activities undertaken to address an assurance deficit (such as generating additional evidence from testing) can only increase confidence in a safety claim, and do not directly reduce risk. In establishing the overall claimed position of the software safety argument, some of the argument claims can, however, be recognised as being more important than others. For example, claims regarding the behaviour of an architectural component (such as a voter), which carries a greater responsibility for risk reduction than other components, are more important to the overall software safety argument. Therefore claims relating to those components would require a greater degree of assurance (more confidence must be established). Safety standards such as [5] and [3] describe how safety integrity requirements may be defined for software functions or components. These safety integrity requirements define the integrity or reliability required in order to support the safety of the system. Where safety integrity requirements have been defined, they can be used as a way of determining the importance of the software safety argument claim to which they relate.

The impact of an assurance deficit should first be determined by considering the importance of the truth of the related claim or claims in establishing the claimed risk position of the safety case (when considered in the overall context of the system). Secondly, the relative importance of the assurance deficit to establishing the truth of that claim must also be considered. One way to approach this is to consider those aspects of the claim that are still assured in the presence of the assurance deficit (due to other evidence or information), and those that are not. Knowing the importance of the truth of the claim in establishing the claimed risk position, *and* the relative importance of the assurance deficit to establishing the truth of that claim, it then becomes possible to reason about the overall impact of the assurance deficit.

In a similar manner to the categorisation of risks within the ALARP approach, the impact of the identified assurance deficits may be usefully classified into three categories. An “intolerable” deficit could be one whose potential impact on the claimed risk position is too high to be justified under any circumstances. At the other extreme, some assurance deficits may be categorised as “broadly acceptable” if the impact of the assurance deficit on the claimed risk position is considered to be negligible i.e. the “missing information” has a negligible impact on the *overall* confidence in the safety argument. In such cases no additional effort to address the assurance deficit need be sought. Finally, a potentially “tolerable” assurance deficit is one whose impact is determined to be too high to be considered negligible, but which is also not necessarily considered to be intolerable. For a potentially “tolerable” assurance deficit it may be considered acceptable only if the cost of taking measures to address the assurance deficit are out of proportion with the

impact of not doing so. The greater the impact of an assurance deficit, the more money system developers may be expected to spend in addressing that deficit.

Note that the impact of an assurance deficit can only be determined on a case-by-case basis for a specific argument relating to a particular system. The same type of assurance deficit (such as a particular assumption) whose impact is categorised as broadly acceptable when present in the software safety argument for one system, may be considered intolerable when present in the argument for a different system. This is because the impact of an assurance deficit considers its impact in terms of the overall safety of the system. It is for this reason that particular argument approaches (such as the patterns discussed in section 4) cannot be stated as sufficient for particular claims, but must be adapted on each use to be appropriate for the particular application.

5.2 Addressing Assurance Deficits

Addressing an assurance deficit requires 'buying' more information or knowledge about the system relevant to the safety claims being made. There will typically be a cost associated with obtaining this information. For those assurance deficits categorised as tolerable, in a manner similar to that adopted for an ALARP assessment process (such as that described in [6]), the value of the information in building confidence in the safety case must be considered when deciding whether to spend that money. In theory it is possible to do a formal cost-benefit analysis based on a quantitative assessment of the costs associated with the available options for addressing the assurance deficit, and the costs associated with the potential impact on the claimed risk position (such as the necessity to provide additional system level mitigations). In many cases however, a qualitative consideration of these issues will suffice. It should be noted that even for ALARP assessments of conventional systems qualitative arguments will often be presented before turning to a quantitative first principles argument [2]. In all cases an explicit justification should be provided as to why the residual assurance deficit is considered acceptable and, wherever appropriate, an argument should be used to provide this justification. Section 4 described how we have provided an argument pattern for constructing an argument to justify that the residual assurance deficits are acceptable.

The approach described above, although similar to ALARP, rather than considering the necessity of adopting measures to directly decrease *risk*, instead considers measures intended to increase the *confidence* that is achieved. As such the framework could be considered to help establish a claimed risk position in the software safety case that is ACARP (As Confident As Reasonably Practicable).

6 Conclusions

Constructing software safety arguments for software can bring many benefits – particularly the ability to explicitly reason about how the evidence generated demonstrates that the software is sufficiently safe. The biggest challenge in constructing a compelling software safety argument is making a judgement as to what is sufficient in order to gain an acceptable level of assurance.

In this paper we have provided a framework for making and justifying decisions about the arguments and evidence required to demonstrate sufficient assurance in the software. This framework includes an argument development approach that systematically considers assurance, utilises a catalogue of software safety argument patterns, and a structured approach for justifying the acceptability of the resulting argument. By using these elements together, we believe it becomes easier to demonstrate that sufficient software safety assurance is achieved.

Acknowledgements

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] CISHEC. "A Guide to Hazard and Operability Studies.", *The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd.*, (1977).

[2] HSE. "The Tolerability of Risk from Nuclear Power Stations", *Health and Safety Executive*, (1988).

[3] IEC. "61508 - Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems", (1998).

[4] T.P. Kelly. "Arguing Safety - A Systematic Approach to Managing Safety Cases." *PhD thesis, Department of Computer Science, The University of York*, (1998).

[5] MoD. "Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems.", *HMSO*, (2007).

[6] Railtrack. "Engineering Safety Management – Yellow Book 3", *Railtrack Plc.*, **volumes 1 and 2**, (2000).

[7] F. Redmill, M Chudleigh, J. Catmur. "System Safety: HAZOP and Software HAZOP". *Wiley*, (1999).

[8] RTCA. "DO-178B - Software Considerations in Airborne Systems and Equipment Certification", (1992).

[9] R.A. Weaver. "The safety of Software - Constructing and Assuring Arguments.", *PhD thesis, Department of Computer Science, The University of York*, (2003).

[10] Fan Ye. "Justifying the Use of COTS Components within Safety Critical Applications.", *PhD thesis, Department of Computer Science, The University of York*, (2005).

[11] http://www.ssei.org.uk

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.	

Table 1: An extract from the argument development assurance analysis summary table.

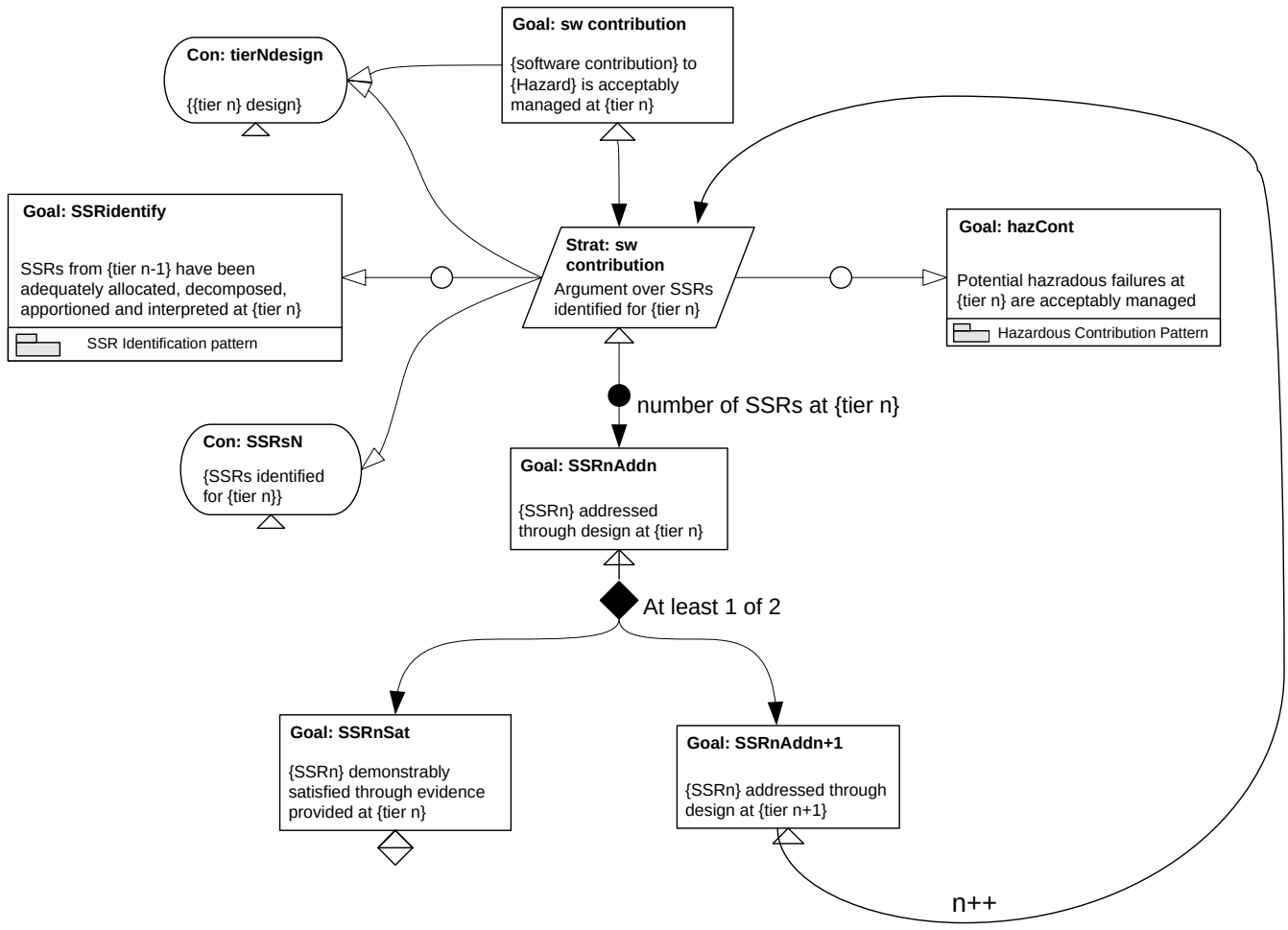

Figure 1: An example software safety argument pattern.