Principled Construction of Software Safety Cases

Richard Hawkins, Ibrahim Habli, Tim Kelly

Department of Computer Science, University of York, UK

Abstract. A small, manageable number of common software safety assurance principles can be observed from software assurance standards and industry best practice. We briefly describe these assurance principles and explain how they can be used as the basis for creating software safety arguments.

Keywords. Software safety, assurance, safety cases, certification.

1 Introduction

We have previously presented a set of software safety assurance principles [1]. The principles are common across most domains, and can be regarded as the immutable core of any software safety justification. In order to demonstrate that a system is acceptably safe, it is increasingly common to provide a safety case for that system.

A safety case comprises "*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*" [2]. For systems that contain software, the safety case must consider the contribution of the software to the safety of the system. Creating a clear safety argument helps to provide explicit safety justification, making it easier to understand, review and criticise the reasoning and evidence presented.

Software safety arguments are challenging to create. Bloomfield and Bishop [3] discussed the current practice and uptake of safety cases for software-based systems. They concluded that, while the application to complex systems is a significant undertaking, the use of assurance cases for software is very appealing, supporting as it does innovation and flexibility. Understanding how the principles of software safety assurance relate to software safety cases makes it easier to understand the required aspects of the safety case, and determine which of those aspects are covered by existing software assurance processes. In this paper, we briefly describe the software safety assurance principles (Section 2) and discuss how these principles can be used as the basis for developing software safety arguments (Section 3).

2 Software Safety Assurance Principles

The principles presented in this section can help maintain understanding of the 'big picture' of software safety issues whilst examining and negotiating the detail of individual standards. Recognising these principles does not remove the obligation to comply with domain-specific standards. However, the principles can provide a reference model for cross-sector certification.

Principle 1: Software safety requirements shall be defined to address the software contribution to system hazards

The assessment and mitigation of hazards is central to the engineering of safety-critical systems. Software, although conceptual, can contribute to these hazards through the system control or monitoring functions it implements (e.g. software implementing anti-lock braking or aircraft warning functions). Hazardous software contributions, identified through a safety process, should be addressed by the definition of safety requirements to mitigate these contributions. It is important for these contributions to be defined in a concrete and verifiable manner, i.e. describing the specific software failure modes that can lead to hazards. Otherwise, we will be in danger of defining generic software safety requirements, or simply correctness requirements, that fail to address the specific hazardous failure modes that affect the safety of the system.

Principle 2: The intent of the software safety requirements shall be maintained throughout requirements decomposition.

As the software development lifecycle progresses, requirements and design are progressively elaborated and a more detailed software design is created. Having established software safety requirements at the highest (most abstract) level of design (see Principle 1), the intent of those requirements must be maintained as the software safety requirements are decomposed. Simply looking at requirements satisfaction is insufficient. The notion of '*intent*' is very important here. It is necessary to consider what was meant by the high level requirement, including implied semantics. It is common for a lot of information to remain unstated or deliberately undefined. A theoretical solution to this problem is to ensure that all the required information is captured in the initial high-level requirement. In practice however this would be impossible to achieve. Design decisions will always be made later in the software development lifecycle that require greater detail in requirements. This detail cannot be properly known until that design decision has been made.

Principle 3: Software safety requirements shall be satisfied.

Once a set of 'valid' software safety requirements is defined, either in the form of allocated software safety requirements (Principle 1) or refined or derived software safety requirements (Principle 2), it is essential to verify that these requirements have been satisfied. The principal challenge for demonstrating that the software safety requirements have been satisfied resides in the fundamental limitations of the evidence obtained from the adopted verification techniques. The source of the difficulties lies in the nature of the problem space. For testing and analysis techniques alike, there are issues with completeness given the complexity of software systems.

Principle 4: Hazardous behaviour of the software shall be identified and mitigated.

Although the software safety requirements established for a software design can capture the intent of the high-level safety requirements, this cannot guarantee that the requirements have taken account of all the potentially hazardous ways in which the

software might behave. There will often be unintended behaviour of the software, resulting as a side-effect from the way in which the software has been designed and developed, that could not be appreciated through simple requirements decomposition. These hazardous software behaviours could result from either *unanticipated behaviours and interactions* arising from software design decisions (side effects of the software design) or *systematic errors* introduced during the software development process.

Principle 4+1: The confidence established in addressing the software safety principles shall be commensurate to the contribution of the software to system risk.
It is necessary to provide evidence to demonstrate that each of the principles described above has been established. The evidence may take numerous forms based upon the nature of the software system itself, the hazards that are present, and the principle that is being demonstrated, and may vary hugely in quantity and rigour. It must be ensured that the confidence achieved from the evidence provided is commensurate to the contribution that the software makes to system risk. This approach is widely observed in current practice, with many standards using notions of integrity or assurance levels to capture the confidence required in a particular software function.

3 Developing a Software Safety Argument

Figures 1 presents, using the Goal Structuring Notation (GSN) [4], the generic structure of a software safety argument that could be created for systems containing software. The argument structure is presented in the form of a safety argument pattern [4]. A fully documented catalogue of patterns from which Figure 1 is extracted is provided in [5]. In [6] we provided a fully developed example of a software safety argument for an aircraft wheel braking system that uses this argument pattern. In the rest of this section we explain how the software safety assurance principles are explicitly addressed through a safety argument created using the pattern from Figure 1.

Principle 1 – The instantiation of the pattern in Figure 1 starts by creating an instance of the '*Goal: sw contribution*' for each identified contribution that the software could make to system hazards. This is to ensure that the software safety argument links to the system safety case by providing explicit traceability to system hazards. Note that there might be more than one contribution that the software could make to each system hazard. For example, one hazard such as 'incorrect altitude displayed' may be associated with multiple software contributions, including software providing incorrect data values or failing to pass data values. Justifying in the safety argument that all the software contributions have been identified is key. Typically, a combination of software Functional Failure Analysis and System Fault Tree Analysis is used to identify these software contributions.

Principle 2 – To address Principle 2 in the argument we need to be demonstrate that the defined Software Safety Requirements (SSRs) correctly reflect the software con-

tributions that were identified at the top level, but also that the SSRs are correct at each level of software design decomposition. The term 'tier' in Figure 1 is used to represent one level of decomposition in the software design (for example, levels of decomposition may be requirements to high-level design, or detailed-design to implementation). This will be replaced at instantiation by the level of design abstraction under consideration (e.g. detailed design). Specifically, the '*Goal: SSRidentify*' provides an argument that the SSRs at each tier are adequately allocated, decomposed, apportioned and interpreted. The term '*adequately*' means that the intent of the high-level SSRs is maintained. It should be noted that this is more than just a traceability argument. The argument must demonstrate that the behaviour is equivalent (cf. notions of 'rich traceability' [7] or 'intent specifications' [8]). The '*Goal: SSRnAddn*' makes a claim regarding each SSR at each software design tier. The '*Goal: SSRnAddn+1*' then shows that the SSR is addressed at the next level of decomposition as well (tier n+1). The same type of argument is created for each tier (as indicated by the loop going back up to '*Strat: sw contribution*').

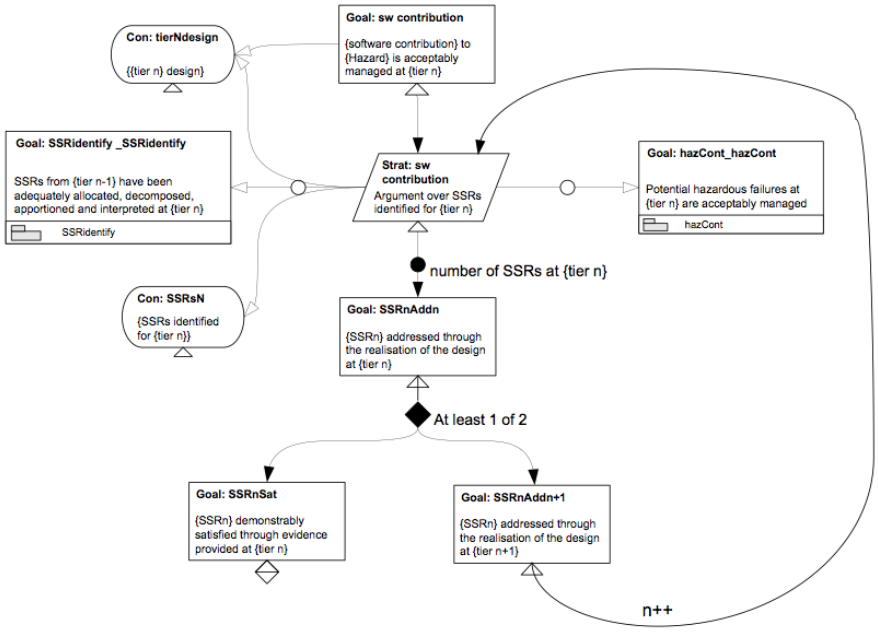

Fig. 1 A pattern for software safety arguments

Principle 3 **–** There is the potential to undertake verification, and provide evidence of satisfaction of the SSRs at any tier (e.g. integration testing for the software architecture, or unit testing for the detailed design). The '*Goal: SSRnSat*' provides an opportunity to do this in the safety argument. Note that it is not always necessary to provide satisfaction evidence for every tier. However, this judgement will affect the level of assurance achieved (this is discussed further under Principle 4+1).

Principle 4 – It is important to justify that that potential hazardous behaviour is managed at each level of design. This is dealt with under the '*Goal: hazCont*'. The argument developed here must demonstrate that (1) systematic errors have not been introduced whilst creating this tier of design and that (2) unanticipated behaviours and interactions arising from the software design decisions at this tier are eliminated or mitigated. The full details of how the '*Goal: hazCont*' is developed is provided in [5].

Principle 4+1 – It is important to demonstrate in the software safety case that the confidence with which the principles have been addressed is commensurate to the contribution of the software to system risk. This requires the provision of a confidence argument [9]. A confidence argument documents the reasons for having confidence, and assesses and where possible quantifies the sources of uncertainty [10], in the main (software) safety argument and evidence.

4 Conclusions

This paper has explained how the software safety assurance principles, observed from software assurance standards and industry best practice, can be addressed in software safety case construction (illustrated by means of a safety argument pattern). Software safety cases are often seen to be about a single issue such as process rigour, standards compliance or V&V. In this paper we have shown how a software safety case should include aspects of all these issues, and must necessarily span the software development process from requirements to verification, and integrate with the wider system safety assessment.

References

1. Hawkins, R., Habli, I., Kelly, T.: The Principles of Software Safety Assurance. 31st International System Safety Conference, Boston, Massachusetts USA (2013)
2. MoD, Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems. HMSO (2007)
3. Bloomfield, R., Bishop, P.: Safety and Assurance Cases: Past, Present and Possible Future – An Adelard Perspective. 18th Safety–Critical Systems Symposium, Bristol, UK (2010.
4. Goal Structuring Notation Working Group: GSN Community Standard Version 1 (2011)
5. Hawkins, R., Kelly, T.: A Software Safety Argument Pattern Catalogue, Technical Report, Department of Computer Science, University of York, YCS-2013-482 (2013)
6. Hawkins, R., Habli, I., Kelly, T., McDermid, J.: Assurance Cases and Prescriptive Software Safety Certification: A Comparative Study. Safety Science, Vol. 59 (2013)
7. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer-Verlag, (2002)
8. Leveson, N.: Intent Specifications: An Approach to Building Human-Centered Specifications. IEEE Transactions on Software Engineering, Vol. 26, No. 1 (2000)
9. Hawkins, R., Kelly, T., Knight, J., Graydon, P.: A New Approach to Creating Clear Safety Arguments. 19th Safety–Critical Systems Symposium, Southampton, UK (2011)
10. Denney, E., Pai, G., Habli, I.: Towards Measurement of Confidence in Safety Cases. Symposium on Empirical Software Engineering and Measurement, Banff, Canada (2011)