Using a Software Safety Argument Pattern Catalogue: Two Case Studies

Richard Hawkins, Kester Clegg, Rob Alexander, and Tim Kelly

The University of York, York, U.K.

{richard.hawkins, kester.clegg, rob.alexander, tim.kelly}@cs.york.ac.uk

Abstract. Software safety cases encourage developers to carry out only those safety activities that actually reduce risk. In practice this is not always achieved. To help remedy this, the SSEI at the University of York has developed a set of software safety argument patterns. This paper reports on using the patterns in two real-world case studies, evaluating the patterns' use against criteria that includes flexibility, ability to reveal assurance decits and ability to focus the case on software contributions to hazards. The case studies demonstrated that the safety patterns can be applied to a range of system types regardless of the stage or type of development process, that they help limit safety case activities to those that are significant for achieving safety, and that they help developers nd assurance deficits in their safety case arguments. The case study reports discuss the difficulties of applying the patterns, particularly in the case of users who are unfamiliar with the approach, and the authors recognise in response the need for better instructional material. But the results show that as part of the development of best practice in safety, the patterns promise signicant benets to industrial safety case creators.

1 Introduction

Providing a compelling software safety argument is a fundamental but challenging part of demonstrating that a system is safe. Part of the problem is providing evidence for low-level argument claims, but there are also difficulties in structuring the argument in an intelligible and maintainable way. To help with this latter problem, we have developed a catalogue of software safety argument patterns which guide engineers in structuring safety arguments.

The pattern catalogue is summarized in [1] and documented fully in Appendix B of [2]. The philosophy underpinning these patterns is that developers must demonstrate assurance in the same fundamental safety claims for all software used in a safety related role; the difference between arguments for different systems is in the way in which these claims are ultimately supported. The patterns we have created define the expected structure of a software safety argument which supports all of the fundamental safety claims.

We intend for the patterns to provide benefits to several different stakeholders. When a developer uses them during the earlier stages of a systems lifecycle, they should find it easier to identify areas where the assurance of the system may be weak. They can then make changes (to the system or its operating restrictions) to address these areas of concern. The patterns can also help reviewers of a software system to identify where assurance deficiencies may exist, and provide a common baseline for agreeing acceptability. In essence, the patterns attempt to encourage best practice in creating and reviewing software safety arguments.

In order to check the effectiveness of the patterns in achieving these aims we applied the patterns to a number of industrial case studies to determine their effectiveness. In this paper we describe some of our experiences of applying the patterns on two of these safety-critical software projects.

In particular, we wanted to assess the patterns in the software safety argument pattern catalogue against the following desirable criteria:

– The patterns should be easy to understand and apply by software development teams.
– The patterns should be flexible enough to apply to any safety-critical software system.
– The patterns should ensure that the resulting software safety argument is explicitly focused on controlling the software contribution to system hazards.
– It should be easy to judge the sufficiency of an argument created using the patterns.

In the next section we give an overview of the pattern catalogue. Sections 3 and 4 then describe our experiences in two case studies: a prototype autonomous vehicle controller, and an aircraft avionics software system. Finally, Section 5 draws some conclusions from these experiences and outlines the future for the pattern catalogue.

2 Software Safety Argument Pattern Catalogue

Prior to the development of our pattern catalogue, the main extant work in the area was that of Weaver [3]. Weaver's catalogue was unique in its time in that, unlike that of Kelly [4], it was specifically aimed at software systems, and specifically designed to connect its patterns together in order to form a single coherent argument. However, Weaver's catalogue has a number of weaknesses. First, the patterns take a fairly narrow view of assuring software safety, in that they focus on the mitigation of known failure modes in the design. Mitigation of failure modes is important, but there are other aspects of software assurance which should be given similar prominence. Second, issues such as safety requirement traceability and mitigation were considered at a single point in Weaver's patterns. This is not a good approach; it is clearer for the argument to reflect the building up of assurance relating to traceability and mitigation over the decomposition of the software design (see later discussion on the tiered approach). Finally, Weaver's patterns have a rigid structure that leaves little scope for any alternative strategies that might be needed for novel technologies or design techniques.

The other relevant existing patterns are those developed by Fan Ye [5] specifically to consider arguments about the safety of systems including COTS software products. Ye's patterns provide some interesting developments from 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 arguing over both the trustworthiness of the evidence and the extent to which that evidence supports the truth of the claim.

The patterns we created were deliberately constructed such that they make no assumptions about project, application or domain specific details. For example they are designed to be applicable for any software development process, any software design methodology, diverse types of system-level hazards and diverse software requirements.

The key organizing assumption for the patterns was that as the software system moves through the development lifecycle there are numerous assurance considerations against which evidence must be provided. Jaffe et al [8] proposed an extensible model of development which captures the relationship between components at different "tiers" (a set of tiers for one project might be for example the software architecture, the software high-level and low-level designs, and the source code). For our purposes we can note that at each tier, different assurance considerations arise. Our patterns are explicitly based on a view of software development as this process of refinement through tiers, and they consider the relationship between the design information at various tiers and the resulting assurance considerations. The number and type of tiers used in the specific design process being used is irrelevant, so long as the assurance considerations are sufficiently addressed at each tier.

Figure 1 summarises the assurance considerations that are repeated at each tier of a software development lifecycle. At each tier, the pattern instantiator must provide evidence which is sufficient to address each of these considerations, and they must provide a compelling argument which explains how the evidence addresses each assurance consideration. It follows that as software development progressed through more detailed design tiers, more assurance evidence is generated.

From Figure 1, the assurance considerations defined for each tier can be seen to be:

1. The safety requirements placed upon the software have been met.
2. Those safety requirements are appropriate for the design at this tier and are traceable to higher tiers.
3. Hazardous errors have not been introduced into design at this tier.
4. Hazardous failure behaviour has been assessed — it has been determined what could go wrong at this tier and how it is mitigated.

If a safety argument is to be compelling, then it is crucial that the high-level structure of the argument is correct. This requires that the argument focuses ex-

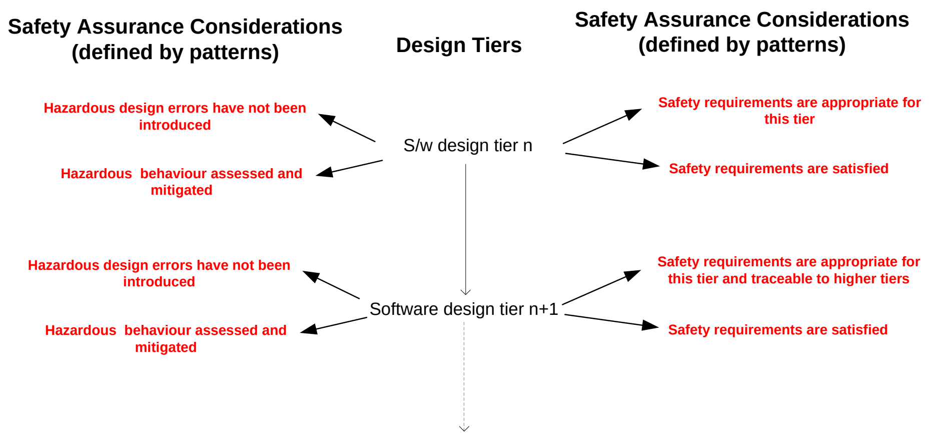
Safety Assurance Considerations (defined by patterns)
Design Tiers
Safety Assurance Considerations (defined by patterns)
Hazardous design errors have not been introduced
Hazardous behaviour assessed and mitigated
S/w design tier n
Safety requirements are appropriate for this tier
Safety requirements are satisfied
Hazardous design errors have not been introduced
Hazardous behaviour assessed and mitigated
Software design tier n+1
Safety requirements are appropriate for this tier and traceable to higher tiers
Safety requirements are satisfied

Fig. 1. System safety requirements

plicitly on how the software can contribute to system level hazards. Our patterns provide this structure by forcing users to consider each system hazard in which software may play a role and to identify the specific software behaviour which may contribute to the hazard. This might involve, for example, working systematically over the base events in fault tree. If an argument creator understands the specific functions or properties of the software in which assurance is required, then they can focus the argument and evidence on those things. This structured approach should help to discourage spurious information being included in the safety argument "just in case".

2.1 Identifying Assurance Deficits

There will be aspects of all safety-related software systems for which assurance is not demonstrated with complete certainty; there will always be things relating to the behaviour of the software system which remain unknown or unclear. We refer to such uncertainties as assurance deficits since they can undermine the assurance that can be demonstrated. It is important to note that assurance deficits do not necessarily correspond to faults or defects in the system, but instead to an inability to demonstrate complete certainty in each safety claim in the argument.

For example, it may be known that the compiler being used has some undefined behaviour. It may be known that a design model used may not accurately represent certain real environmental features or that a component has not been exhaustively tested. Or it may be known that an assumption that has been made about partitioning of some software modules may not actually hold in all cases.

The argument patterns can help identify where such assurance deficits exist. These assurance deficits may relate to the safety evidence generated or to the safety argument itself. It is through managing assurance deficits that the required assurance can be achieved.

3 Case Studies

The next two sections describe case studies where the software safety case patterns were used on real products. The first is the control software for a prototype autonomous vehicle and the second for an aircraft avionics system. Each of the case studies highlights different ways that using the patterns can benefit the production of safety case arguments by indicating where those arguments are either missing evidence to back up safety claims or failing to identify clearly the software contribution to the hazards being considered. The case studies differ in the stage of technology readiness, the type of software deployment and the derivation of safety properties. They also differ in terms of the level of experience of the engineers creating the safety case. Despite these differences, the patterns were sufficiently flexible to be used and provide benefits to both safety cases.

3.1 Prototype Autonomous Vehicle Case Study

As part of a SEAS DTC [9] project on safety of autonomous systems we craeted a safety argument for an autonomous system drawn from the SEAS DTC Exemplar 2 scenario [10]. The system chosen was a prototype Unmanned Ground Vehicle (UGV) that formed part of a larger System of Systems (SoS), including an Unmanned Aerial Vehicle (UAV) and ground control units. As a high-level hazard and safety analysis for the SoS had already been completed using previously developed techniques [11], we decided to construct a partial safety argument that would start at a system level hazard for the UGV and end with arguments justifying the safety of software that could contribute to that hazard. The UGV in question is an adapted all-terrain vehicle (the Wildcat) produced by the Advanced Technology Centre of BAE Systems. The current prototype is able to operate without a driver, to follow an off-road GPS waymarked route by calculating the best path within the waymarked corridor and is able to avoid static objects. As with many UGVs it is heavily reliant on GPS signals for their autonomous operation. However, in cases where the GPS signal is lost or jammed, the vehicle is able to continue to plan its path by taking measurements from the Inertial Measurement Unit (IMU) in conjunction with other on-board sensors (such as LIDAR). Unfortunately the IMU measurements (and therefore estimates of the vehicle's position) are subject to drift over time, giving an ellipse of uncertainty with regard to the vehicle's true position that can grow in an unbounded fashion in some scenarios (e.g. after entering a long tunnel). This could result in the vehicle colliding with objects or the side of the road as it miscalculates its position.

While there are many potential collisions that could be described, the approach adopted was first to identify potential accidents and the hazards that could lead to those accidents occurring. Based on the hazards, we next identified a set of top level system safety requirements. These requirements were then further decomposed using techniques derived from the Goal-Oriented Requirements Engineering (GORE) approach of Lamsweerede and others [12]; this proved fairly straightforward and intuitive.

The act of decomposing the system safety requirements gave rise to safety requirements over the software that form the starting point for instantiating the software safety case patterns discussed in this paper. Our intention was not to conduct an exhaustive safety review of a particular hazard and its mitigation; instead we chose to restrict ourselves to a particular aspect of one hazard (object collision after loss of GPS signal, see Table 1, safety requirement SR2-4) and to follow the instantiation of a software safety case pattern to the point at which evidence would normally be presented to meet the software safety requirements defined at the lowest levels. This naturally excluded much peripheral work that would have been essential to a full safety case, as either the technical information was not to hand or we felt it was not directly related to the part of the safety case we were covering.

Table 1. System safety requirements

Safety requirement	**Description**	**Notes**	**System or software components**
SR2 – top level	While moving, the UGV avoids collisions with objects.	Top level SR2 depends on availability of GPS, sensor ranges and quality of their data, path planning and vehicle driver functions.	All of UGV
SR2-1	Vehicle restricts speed such that stopping is possible before collision occurs with objects.	Stopping can be affected by traction, gradient, steering, hardware or vehicle damage.	Sensors. Actuators. SW: Driver.
SR2-2	Vehicle restricts speed such that it can manoeuvre to avoid objects in its path before collision occurs.	Steering can be affected by speed, camber, traction, rate of turn or hardware and vehicle damage.	Sensors. Actuators. SW: Pilot, Driver.
SR2-3	Vehicle restricts speed such that it can plan a new path to avoid objects.	Sufficient time is allowed for vehicle to plan new path even in complex environments.	Sensors. SW: Planner.
SR2-4	In cases where GPS is lost or blocked, vehicle to maintain last good path until GPS signal is re-established.	New plans can be formed relying on IMU data but this carries significant risk.	Sensors. Actuators. SW: Planner, Platform Manager.
SR2-5	On re-establishing a GPS, the vehicle converges path differences between estimated position and true position in a safe manner.	Convergence carries significant risk, as the degree of positional error is impossible to predict and object avoidance may be impossible if vehicle needs to "teleport" to true position.	Sensors. SW: Planner.
SR2-6	If vehicle is unable to maintain a planned path, vehicle is brought to emergency stop	Ability to plan a new path is limited by CPU processing and sampling speeds, current speed of vehicle and complexity of environment.	Sensors. SW: Pilot, Driver, Platform Manager.

The software safety case patterns require that the software contributions to the hazard in question have been identified. There are various ways this top level contribution can be obtained; for example, it might occupy one node in a causal model, such as a fault tree analysis. For the instantiations of the software safety case pattern to be as easily as possible, it is important that the high level software contribution to the hazard is clearly understood and defined, as this forms the starting point of the software safety case and defines the context in which the case is made.

The first thing that the use of the patterns helped to do was to highlight that the top level software contribution to the hazards had not been clearly identified for the prototype UGV. The reason for this is that safety requirements are not typically expressed by describing how the system or software can contribute to a hazard. Instead, safety requirements tend to be framed in language that states the requirement as “necessary to mitigate” the hazard. Thus from our system safety requirements we have SR2 decomposed to SR2-4 (see Table reftab2, note have selected just that which is related to the loss of GPS signal referred to above):

System Safety Requirements
SR2
While moving, the UGV avoids collisions with objects.
SR2-4
In cases where GPS is lost or blocked, vehicle maintains last good path until GPS signal is re-established.

Neither of these explicitly defines the hazard or mentions the software contribution to it. In fact SR2-4 does not specify how the vehicle should maintain a good path; it could be through hardware, software or a combination of both. The use of the patterns highlights the importance of making the software contributions to hazards explicit. Note that for our purposes, we are only concerned with the software contribution with regards to loss of the GPS signal (there are other software contributions we do not define here). We were able to transform the requirement above into the following expression of a contribution to a hazard.

Hazard described in SR2
UGV collides with static object.
Software contribution to Hazard in SR2
Software fails to plan safe path for vehicle when GPS signal is lost or blocked.

This rewriting of SR2-4 gives a clear starting point from which to construct a safety argument using the patterns. From this point downward in the decomposition we can refer to safety requirements over the software, as shown by the software safety requirements extract in Table 2 that decomposes SR2-4 by apportioning software safety responsibilities (note that the full decomposition is much more detailed and goes down to the level of individual program functions and variable declarations).

Table 2. Transistion from system to software safety requirements

Software Safety requirement	Description	Notes / mitigation in design / other risks.	Risks / hazards introduced as a function of design decision
SR2-4	In cases where GPS is lost or blocked, vehicle to maintain last good path until GPS signal is re-established.	New paths can be formed using IMU data but this carries unspecified levels of risk.	
SR2-4-1	Where GPS signal is momentarily lost, software takes positional input from IMU to continue planning new paths until GPS signal is re-established.	Paths planned without GPS data become increasingly inaccurate. If vehicle starts to skid or loses traction, IMU data becomes unreliable.	Positional "drift" can grow in an unbounded fashion during loss of GPS signal, therefore vehicle could collide with an object it knew about and had planned to avoid.
SR2-4-2	If vehicle experiences loss of GPS signal for longer than 30 seconds, software brings vehicle to halt.	Bringing vehicle to halt within a GPS "tunnel" may result in vehicle being unable to continue mission.	Vehicle falls into enemy hands or becomes "lost" to accompanying UAV.

Case Study Findings Before the patterns were used to guide the development of the safety case argument an explicit distinction had not been made between the system level safety requirements and the software contribution to the hazard. This led to some confusion when first instantiating the patterns, as the patterns require that the software contributions to hazards are identified. This is a strength of the software safety argument patterns, as they encourage the derivation of the software contributions and this represents good practice for software safety. The diagram in Figure 2 illustrates how the patterns force this distinction to be made - at the different design tiers, it must be demonstrated that the safety requirements are appropriate for that tier and are traceable to higher tiers. It can be seen that as the design tiers become more detailed and more software-specific, so the safety requirements for that tier must do also. Figure 2 shows how clear traceability can thus be established up to the system hazards.

We should note, here, that the system in this case study was a prototype product whose design is yet to be finalised. Perhaps we should not expect such a product to have something as specific as a fault tree (which would isolate the software contribution to the hazard in language that is more suited to the pattern);this type of safety analysis is more common on products with a higher

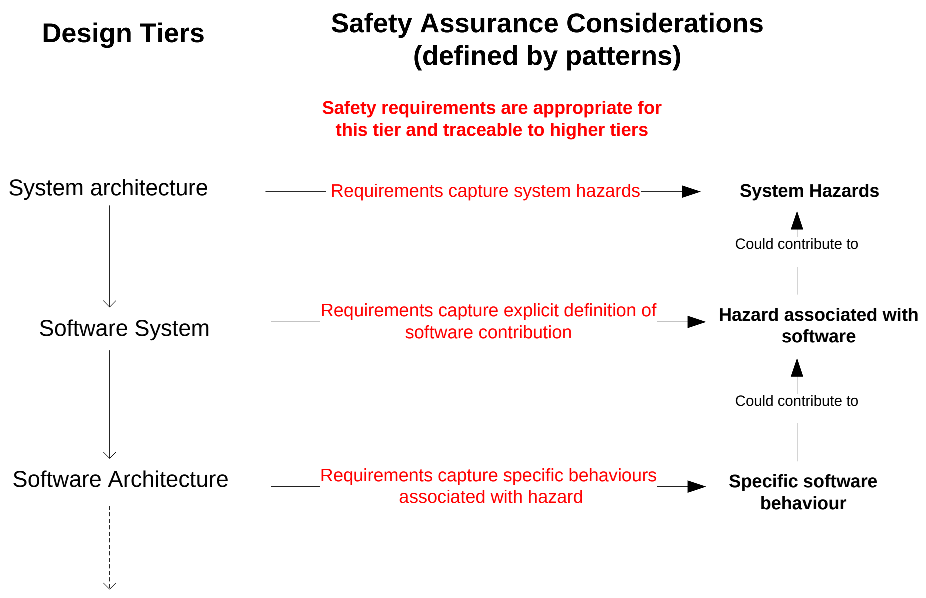
Design Tiers
Safety Assurance Considerations (defined by patterns)
Safety requirements are appropriate for this tier and traceable to higher tiers
System architecture
Requirements capture system hazards
System Hazards
Could contribute to
Software System
Requirements capture explicit definition of software contribution
Hazard associated with software
Could contribute to
Software Architecture
Requirements capture specific behaviours associated with hazard
Specific software behaviour

Fig. 2. Transistion from system to software safety requirements

level of technology readiness. Here, we carried out a fairly intuitive system to software safety requirements decomposition - given our prototype's operational scenario, and the fact it is undergoing further development, this is probably not inconsistent with real industrial practice. If this is the case, then care needs to be taken to explicitly define both the hazard and the top level software contribution to it, perhaps outside the main decomposition tables. Indeed, whether using the patterns or not, it is beneficial to carry this exercise out so as to have a clear understanding of the software contribution across the system. As everything below this point in the decomposition will be a safety requirement on the software rather than the system, the software safety requirements can be inserted into the corresponding tiers of the pattern alongside the evidence selected to meet those requirements.

It should be noted that this Wildcat UGV case study was carried out by researchers who had no prior experience of using the patterns and limited experience with safety cases in general. Despite this, implementing the patterns was relatively easy, and helped ensure we had adequately covered the necessary assurance considerations. The patterns themselves provide a well structured framework within which to document design rationale regarding mitigation of hazards and justification of evidence. By tying this to tiers within the software architecture, the patterns make it obvious where to locate arguments about design decisions at a particular level. This, in turn, ties argument claims closely to specific elements in specific design or implementation artefacts, which helps argument assessors judge the sufficiency of the resulting argument.

3.2 Aircraft Safety Critical Software System Case Study

The system considered in this case study was a safety critical aircraft avionics system. The system comprised of a single line replacement item; the software for this was the subject of the case study.

The potential safety hazards associated with the system are partially mitigated by means of hardware safety interlocks independent of the system software. This approach minimises the contribution to safety from the software. Software involved hazards can also be addressed by ensuring that the integrity of the CPU commands to the hardware is sufficient to mitigate these hazards. This was achieved by the use of a high integrity Safety Monitor component within the main application software. The application software is split into two components, the Controller and the Safety Monitor. The Controller implements all of the actual system functionality, but all critical outputs are routed as requests to the Safety Monitor. The Safety Monitor sees the same set of real world inputs as the Controller and continuously calculates the safety state of the system. All critical outputs which are passed from the Controller to the Safety Monitor are checked against defined System Level Safety Properties, and the Safety Monitor vetos any outputs which would infringe any of the safety properties. Any safe' outputs (those which are determined not to infringe any safety properties) are routed by the Safety Monitor onto the software device drivers.

The system level safety properties are the necessary conditions under which the behaviour of the system is considered to be safe. The properties were identified from the system hazards during system Preliminary Hazard Identification and Analysis. A formal definition (using the Z specification language) of the safety properties was provided in order to state the necessary conditions precisely and unambiguously.

This case study was undertaken at a fairly early point in the software development lifecycle. However, even at this early stage enough information was available about the design and development of the software, and plans in place for the later stages of the development, that a detailed software safety argument could be formed using the software safety argument patterns as guidance, particularly for the aspects of the argument relating to the Safety Monitor

Case Study Findings The use of the software safety argument patterns highlighted a number of potential assurance deficits associated with the software. Thus identified, the significant assurance deficits could be dealt with. If the patterns had not been used then the assurance deficits may well not have been discovered until later in the development process, increasing the cost and possibly causing schedule slips of the system. Here, we will focus our discussion on one particular deficit, which relates the provision of sufficient evidence for certain safety properties.

The issue is illustrated in Figure 3. The left-hand side of the figure shows the tiers of design for the Safety Monitor software, while the right-hand side shows one of the safety assurance considerations at that tier (as determined from the patterns) and how that consideration is met. The argument patterns demand that direct evidence of satisfaction of safety properties is provided. The diagram shows the way in which this evidence was provided for the software at each design tier.

It can be seen that at the software system level, evidence is generated by performing system tests that check the behaviour of the software is as defined by the safety properties specification. At the level of the software architecture, a separate specification is defined for each architectural element (module). Evidence can be generated at this tier through module testing against the specification for each module. Note that this evidence is not directly checking the behaviour of the module against that defined by the safety properties specification. This is acceptable as long as the safety properties required of the safety monitor module have been correctly captured in the safety monitor specification. The patterns highlight the importance of demonstrating that the safety properties are adequately interpreted for the Safety Monitor module.

For the class design of the safety monitor module there can be seen to be no evidence which can directly show that the classes behave in accordance with the safety properties. Although unit testing is performed, this is evidence only that the Safety Monitor behaves according to the design specification. In order for unit testing to meet the safety assurance consideration, we also need assurance that the class design correctly captures the required high-level safety properties. Again, the patterns highlighted the importance of adequately interpreting the safety properties for each of the classes in the safety monitor module design.

Finally in Figure 3, it can be seen that static analysis is provided as evidence at the level of the source code. The analysis was conducted using SPARK proof annotations [6] included in the safety monitor code. Again, for this evidence to be effective from an assurance perspective, it must be demonstrated that the proof annotations completely and correctly capture the required safety properties.

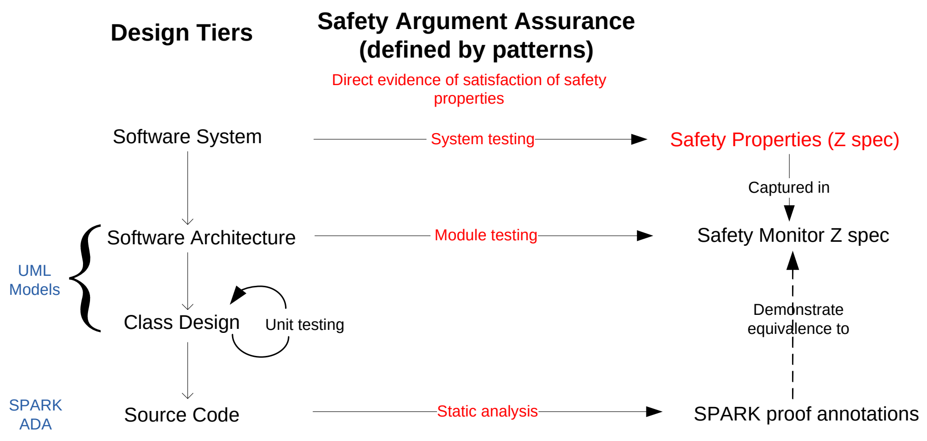

Fig. 3. Transistion from system to software safety requirements

This example illustrates how, by encouraging the developer to consider explicit assurance claims relating to the required safety properties at every tier of design decomposition, the software safety patterns highlighted the potential pitfalls of any “gap” between the safety properties themselves and the tier against which evidence is provided. The patterns identified that the most effective strategy from an assurance perspective would be to explicitly interpret the required safety properties at each level of design decomposition.

Other potential assurance deficits were highlighted by the use of the patterns in this case study. A lack of assurance regarding potential hazardous failures at each design tier was identified. Every time there is a decomposition in the design, there is the potential to introduce erroneous behaviour into the design which could manifest itself as a hazardous software failure. There had been analysis conducted to identify new or additional failure modes of the software at the architecture level (although this was fairly unstructured), but application of the patterns highlighted a need to perform similar analysis at other levels of design.

4 Conclusions

The case studies we have undertaken have given us confidence that they have the desirable criteria defined at the start of this paper:

– The patterns should be easy to understand and apply by software development teams.
– The patterns should be flexible enough to apply to any safety-critical software system.
– The patterns should ensure that the resulting software safety argument is explicitly focused on controlling the software contribution to system hazards.
– It should be easy to judge the sufficiency of an argument created using the patterns.

It is clear that the patterns are fairly easy to understand. This has been demonstrated through the relative ease with which they were applied to the autonomous system by people completely unfamiliar with the patterns.

It has been shown that the patterns are very flexible. The two case studies reported here were on very different types of software system, but the patterns proved to be equally applicable. This was particularly reassuring in the case of the prototype autonomous vehicle, which is a novel system at an early stage of development.

Both case studies made it clear that the resulting safety assurance argument is very focused on demonstrating how the software contributions to system hazards are controlled. This is an advantage over the unfocussed safety arguments that are often produced. It was seen in the case of the aircraft software system that the development team noted how the structure of the generated argument helped to clearly highlight to them which of their software safety and development activities were most important from a safety assurance perspective. In particular, they commented that the case study revealed that many of the things that they focus their attention on are general assurance activities, rather than activities that explicitly help to address specific software contributions. This could help to focus attention on the activities which are most important to software safety assurance. In addition, it makes it easier to judge the sufficiency of the resulting argument, since the relationship between the generated evidence and the safety of the system was clear and explicit.

Most importantly, the case studies have shown that applying the patterns can identify potential assurance issues, which can then be addressed as early as possible. If left unidentified, such issues could lead to safety problems during operation.

The case studies have also identified areas where more work on the patterns would be beneficial. In particular we think that clearer guidance is required on the process of instantiating the patterns for a particular application. This would seem to be particularly required when creating large, complex safety cases. For such large complex software systems, it would also be beneficial to provide guidance on how to group arguments with respect to the corresponding design elements in order to keep a clear relationship between the software design structures, and the structure of the argument.

The argument structures in the software safety argument patterns discussed in this paper are broadly in line with the new "assured safety case" structure presented by Hawkins et al in [7]. The patterns will be reviewed to ensure they are completely consistent with that structure. The constraints in this new format for safety cases have the potential to further focus software safety arguments on those claims and evidence that matter the most.

5 Acknowledgements

The authors would like to thank the Systems Engineering for Autonomous Systems Defence Technology Centre (SEAS DTC) and the Software Systems Engineering Initiative (SSEI), both funded by the UK Ministry of Defence, which supported the two case studies described in this paper.

References

1. Hawkins R., Kelly T.: A Systematic Approach for Developing Software Safety Arguments. In Proceedings of the 27th International System Safety Conference, Huntsville, AL (2009).
2. Menon C., Hawkins R., McDermid J.: Interim standard of best practice on software in the context of DS 00-56 Issue 4. Technical Report SSEI-BP-000001. Software Systems Engineering Initiative, York. https://ssei.org.uk/documents/ (2009).
3. Weaver R. A.: The safety of Software - Constructing and Assuring Arguments. PhD thesis, Department of Computer Science, The University of York (2003).
4. Kelly T.: Arguing Safety - A Systematic Approach to Managing Safety Cases. PhD thesis, Department of Computer Science, The University of York (1998).
5. Ye F.: Justifying the Use of COTS Components within Safety Critical Applications. PhD thesis, Department of Computer Science, The University of York (2005).

6. Barnes J.: High Integrity Ada - The SPARK Approach. Addison Wesley (1997).
7. Hawkins R., Kelly T., Knight J., Graydon P.: A New Approach to Creating Clear Safety Arguments. In Proceedings of the Nineteenth Safety-Critical Systems Symposium (SSS '11), Southampton (2011)
8. Jaffe M., Busser R., Daniels D., Delseny H., Romanski G.: Progress Report on Some Proposed Upgrades to the Conceptual Underpinnings of DO178B/ED-12B. In Proceedings of the 3rd IET International Conference on System Safety (2008).
9. Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre (DTC) `http://www.seasdtc.com/`
10. Bardo B.: Autonomous Systems — A New Partnership Between Man and Machine. Presentation to SEAS DTC (2010). `http://www.innovate10.co.uk/uploads/BillBardo-theSEASDTC.pdf`.
11. Alexander R., Herbert N., et al.: Deriving Safety Requirements for Autonomous Systems. Proceedings of the 4th SEAS DTC Technical Conference, Edinburgh (2009).
12. Lamsweerde A.: Goal-Oriented Requirements Enginering: A Roundtrip from Research to Practice. Proceedings of the Requirements Engineering Conference, 12th IEEE International (2004).