The Principles of Software Safety Assurance

R.D. Hawkins, Ph.D.; Department of Computer Science; The University of York, York, UK

I. Habli, Ph.D.; Department of Computer Science; The University of York, York, UK

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

Keywords: Software Safety, Standards

Abstract

We present common principles of software safety assurance that can be observed from software safety standards and best practice. These principles are constant across domains and across projects, and can be regarded as the immutable core of any software safety justification. The principles also help maintain understanding of the 'big picture' of software safety issues whilst examining and negotiating the detail of individual standards, and provide a reference model for cross-sector certification.

Introduction

As the use of software in safety-critical applications has grown, so have the number of software assurance standards. There are now many software standards, such as DO-178B/C for avionics [1], CENELEC-50128 for railway applications [2], ISO26262 for automotive applications [3], and the cross-domain standard IEC61508 [4]. Unfortunately, amongst these standards there are many differences in terminology, concepts, requirements and recommendations. This may seem like a hopeless situation. However, there are a small (and manageable) number of common software safety assurance principles that can be observed both from these standards and best practice. This paper presents these principles, the rationale for these principles, and explains how they relate to existing standards. The principles presented are constant across domains and across projects, and can be regarded as the immutable core of any software safety justification. Recognising these principles, of course, doesn't remove the obligation to comply with domain specific standards. However, they help maintain understanding of the 'big picture' of software safety issues whilst examining and negotiating the detail of individual standards, and provide a reference model for cross-sector certification.

Software Safety Principles

1. Requirements Validity

The assessment and mitigation of hazards is central to the engineering of safety-critical systems. Hazards such as unintended release of braking in cars or absence of stall warnings in aircraft are conditions that occur at the system level and can lead, under certain environment conditions, to accidents. 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). Typically, the way in which software, as well as other components such as sensors, actuators or power sources, can contribute to hazards is identified in the system safety assessment process using safety analysis techniques such as Fault Tree Analysis or Hazard and Operability Studies (HAZOP). The outcome of these techniques should drive the development of safety requirements and the allocation of these requirements to software components.

It is important to note that, at this stage of the development, software is treated as a black box, used for enabling certain functions, and with little visibility into the way in which these functions are implemented. Failure to identify hazardous software failures, and define and allocate appropriate safety requirements, can increase of the risk of certain system hazards to unacceptable levels.

For example, software implements safety-critical functions in medical devices such as pacemakers and infusion pumps. In particular, infusion pumps, according to the Food and Drug Administration (FDA) in the US, have been associated with safety problems [5]. Some of these problems relate to the embedded software. "Key bounce" is one of the software problems reported to the FDA. This problem occurs when the software in an infusion pump

interprets a single keystroke (e.g. rate of 10 mL/hour) as multiple keystrokes (e.g. infusion rate of 100 mL/hour). Obviously, this behaviour, implemented by the software, can lead to events with serious safety consequences.

In 2010, the FDA ordered Baxter Healthcare Corp to recall Colleague Volumetric Infusion Pumps used in the US [6]. This recall was the result of the FDA identifying safety problems associated with these infusions pumps (amongst other models of infusion pumps). The analysis carried out by the FDA traced the sources of some of these safety problems to software defects. The recall of the Colleague pumps was not an isolated instance. 87 recall cases were reported between 2005 and 2009 by the FDA as a result of infusion pump safety concerns. More worryingly, between the same period, the FDA received over 56,000 reports of issues related to the use of infusion pumps: "approximately 1% were reported as deaths, 34% were reported as serious injuries, and 62% were reported as malfunctions" [7]. Since 2010, the FDA has placed increased attention to embedded software, particularly within its new guidance document that can be used for preparing premarket notification submissions for infusion pumps [7].

In short, software is a core enabling technology used in safety-critical systems and as such the ways in which software can contribute to system hazards should be an integral part of the overall system safety process. Hazardous software contributions, identified in a safety process, are 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 in danger of defining generic software safety requirements, or simply correctness requirements, that can fail to address the specific hazardous failure modes that affect the safety of the system.

This gives us our first software safety assurance principle:

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

2. Requirement Decomposition

As the software development lifecycle progresses, the requirements and design are progressively decomposed and a more detailed software design is created. The requirements derived for the more detailed software design are often referred to as derived software requirements. Having established complete and correct software safety requirements at the highest (most abstract) level of design, the intent of those requirements must be maintained as the software safety requirements are decomposed.
On 14th September 1993, Lufthansa Flight 2904 from Frankfurt was cleared to land in heavy rain at Warsaw-Okecie Airport. On landing, the aircraft failed to achieve the necessary deceleration. Seeing the approaching end of the runway and an earth embankment behind it, the pilot steered the aircraft off the runway. The aircraft travelled a further 90 m before hitting the embankment. Two of 70 occupants of the Airbus A320-200 died in the accident, including the co-pilot [8].

There were three means of ground deceleration on the aircraft: spoilers, brakes, and thrust reversers. For safety reasons, both the thrust reversers and spoilers should only be deployed when the aircraft is on the ground (deploying them in the air could be catastrophic). This requirement was achieved for the A320-200 through software which (simplified) required that the spoilers are only activated if there is either: weight of over 12 tons on each of the two main landing gear, or the wheels of the aircraft are turning faster than 72 knots. The thrust reversers are only activated if the first condition is true. There was no way on this aircraft for the pilot to override the software and activate either braking system manually.

It should be noted at this point that the high-level safety requirement (not to deploy air braking until the aircraft is on the ground) was valid. The causes of the accident only start to emerge when we consider how that high-level safety requirement was interpreted for the detail of the software design. Under normal circumstances, the detailed requirements implemented by the software will meet the intent of the high-level safety requirement (the software will deploy the spoilers and thrust reversers once the aircraft is on the ground). On this day, this did not happen. The intent of the high-level safety requirement was not successfully met by the detailed software safety requirements. Why not?

On landing, the aircraft was deliberately banked to the right and travelling slightly faster than usual in order to compensate for expected cross-winds. Instead of a cross-wind, there was in fact a tail-wind. As a result, the aircraft remained banked to the right as it touched the ground. This meant that there was only weight on one of the landing gear. In addition the very wet runway led to hydroplaning, meaning that the one wheel that was on the ground was not spinning at the required 72 knots. Neither of the required conditions was therefore met, and braking was not activated. Note that the software functioned correctly according to its specification (defined by the software safety requirements), it was the software safety requirements themselves which had been interpreted incorrectly.

What has been described above is essentially a problem of on-going requirements validation. There is always a problem as you go through a software development process of how to demonstrate that the requirements at one level of design abstraction are equivalent to the requirements defined at a more abstract level. Simply looking at requirements satisfaction is insufficient. In the accident above, the software safety requirements had been satisfied, however we have seen how those requirements were not equivalent under all real-world environmental conditions to the intent of the high-level safety requirement. This is a very challenging problem. As well as environmental issues, other factors which may complicate the suitability of the decomposition include human factors issues (a warning may be displayed to a pilot as required, but that warning may not be noticeable on the busy cockpit displays).

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.

Clearly if the software is to be considered safe to operate, the decomposition of safety requirements is an area that must always be addressed. This gives us our second software safety principle:

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

3. Requirements Satisfaction

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. A key prerequisite for requirements satisfaction is that these requirements are clear, defined in sufficient detail and are indeed verifiable. The types of verification technique used to demonstrate the satisfaction of software safety requirements will depend on the safety criticality, development stage and implementation technology. As such, it is neither feasible nor prudent to try to prescribe specific verification techniques that should be used for the generation of verification results.

The loss of the Mars Polar Lander (MPL) in January 1999 [9], as part of NASA's Mars Surveyor Program, represents an event in which inadequate software verification, specifically inadequate software testing, was a contributory factor (another contributory factor was inadequate requirements specification). A software error leading to the premature shutdown of the decent engines was considered a probable cause of the loss of the lander. In particular, the software fault-injection testing regime was considered inadequate to stress test the flight software, especially testing for transient surface touchdown sensor signals. Further, the test environment was deemed insufficient to detect flaws in the touchdown sensing software.

Given the complexity and the safety criticality of many software-based systems, it is clear that the use of one type of software verification is insufficient to satisfy the software safety requirements and hence a combination of verification techniques are often needed to generate the verification evidence. Although testing and expert review are commonly used to generate primary or secondary verification evidence, there is an increased emphasis on the suitability and effectiveness of formal verification in satisfying the software safety requirements with a high degree of certainty [1].

The principal challenge for demonstrating that the software safety requirements have been satisfied resides in the fundamental limitations of the evidence obtained from the techniques outlined above. 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, particularly those used in implementing autonomous capabilities. Formal methods do have some advantages for verification of the core logic of the software, though challenges remain here, too: assurance of model validity is difficult to provide, and formal methods do not address the fundamental issue of hardware integration.

Assuring the safety of software systems clearly rests in the ability to satisfy the defined software safety requirements. This gives us our third software safety assurance principle:

Principle 3: Software safety requirements shall be satisfied.

4. Hazardous Software Behaviour

Although the software safety requirements placed on 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 from the way in which the software has been designed and developed, which 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
- systematic errors introduced during the software development process

On 1st August 2005, a Boeing 777-200 flight from Perth to Kuala Lumpur experienced a number of serious alerts, spurious indications and dangerous auto-pilot activity [10]. The pilot managed to return the aircraft safely to Perth only by disengagement of the auto-pilot, manually overriding warnings and automatic commands, and reliance on information provided by air traffic control. The initiating event for this incident occurred some four years previously when one of the aircraft accelerometers failed such that it provided erroneously high output. The software in the aircraft's Air Data Inertial Reference Unit (ADIRU), in accordance with its specification, disregarded the erroneous accelerometer and instead relied upon data from a back-up accelerometer. The incident occurred when the back-up accelerometer also failed. At this point the ADIRU software reverted to taking input from the accelerometer that had initially failed. The reason for this was that the ADIRU software had been designed such that when it was shut down and restarted the accelerometer was no longer recognised as faulty, and therefore taken by the software to be available for use should it be required.

This incident highlights the problems that can arise from the unanticipated side-effects of a software design. It would only be through a systematic and thorough consideration of potential software failure modes and their effects (both on the software and other systems) that such incidents might be anticipated. If potential hazardous software behaviour has been identified, then it is possible to put in place measures to address it. However this requires that analysis of the potential impact of software design commitments is performed.

Not all hazardous software behaviour will arise due to unanticipated effects of the software design. Hazardous behaviour may also be observed as a direct result of errors introduced during the software design and implementation processes. During the first Gulf war a Patriot missile battery failed to track and intercept a Scud missile, which hit a US Army barracks killing 28 soldiers and injuring 98 [11]. This incident was a direct result of, in part, a lack of precision in a conversion performed by the software from a floating point number to an integer, which reduced the accuracy of the integer over time. It is not uncommon for such seemingly trivial development errors to have large consequences.

The key thing to note here is that this is not a general software quality issue. For the purposes of software safety assurance we are only concerned with those errors that could lead to hazardous behaviour. This allows effort to be focussed on reducing systematic errors in those areas where there may be a safety impact. In practice it may not be realistic to systematically establish direct hazard causality for every error, it is probably desirable therefore for a period of time to adopt activities that are seen to be best practice. However the rationale for doing so should at least be based upon some experience within the software safety community of how the specific issue being addressed has given rise to safety related incidents. It is also important that the most critical aspects of the software design are identified, to ensure that sufficient rigour is applied to their development.

If there is to be confidence that the software will always behave safely, any software behaviour that may be hazardous must be identified and prevented. This gives us our fourth software safety principle.

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

5. Confidence

For any safety related system containing software, the four principles described above apply. It is necessary to provide evidence to demonstrate that each of the principles has been established for the software. The evidence may take numerous forms based upon the nature of the software system itself, the hazards that are present, and which principle is being demonstrated. The evidence provided will determine the confidence or assurance with which the principle is established, and may vary hugely in quantity and rigour. It is therefore important to ensure that the confidence established is appropriate in all cases. This is commonly done by ensuring that the confidence achieved is commensurate to the contribution that the software makes to system risk. This approach ensures that most effort (in generating evidence) is focussed on those areas that reduce safety risk the most. 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.

The Boeing 777 aircraft has a Fly-By-Wire Flight Control System (FCS). The central element of the FCS is the Primary Flight Computer (PIFC) [12]. The 777 FCS provides the single source of control of the aircraft in the pitch, roll and yaw axes. Should there be a failure of the PIFC, then control of the aircraft could be lost, with no further mitigations available. The PIFC function clearly makes a very large contribution to risk for the aircraft, and as such was determined to require the highest level of assurance. This in turn necessitated a robust software architecture and design approach (including triple-redundancy) and the generation of extensive evidence including rigorous verification and validation activities (such as mathematical proofs for satisfying requirements and design specifications).

Sizewell B is a pressurised water reactor nuclear power station in the UK. The reactor protections system entails a computerised Primary Protection System (PPS) and a hard-wired Secondary Protection System (SPS) [13]. The reactor protection system as a whole was determined to require the highest safety integrity level due to the high risk associated with the system. In this case the computerised PPS does not take sole responsibility for this risk. Since the PPS and the SPS were determined to be independent (they have sufficiently independent failure modes), the risk contribution can be split between the two systems. The SPS offers failure protection for the PPS, and thus the integrity required from the PPS is reduced. This means that the confidence that must be achieved for the PPS software function is also reduced (the quantity and rigour of the software safety evidence generated may be lower).
These two simple examples illustrate how projects make decisions on the required confidence based on a judgement of the software's contribution to system risk. Such a judgement is not always straightforward. For example there has been much debate about the overall risk posed by unmanned aerial vehicles and the contribution that software can make to this risk.

Once the required confidence has been determined, it is necessary to be able to assess whether this has been achieved. In order to judge the level of confidence with which each principle is demonstrated, there are a number of considerations. Firstly the appropriateness of the evidence should be considered. This must take account of the limitations of the type of evidence being used. These limitations will influence the confidence that can be gained from each type of evidence with respect to a particular principle. These limitations will include for example the achievable coverage of different types of testing, the accuracy of the models used in formal analysis techniques, or the subjectivity of review and inspection. The limitations of different types of evidence mean that in order to achieve the required level of confidence against any of the principles, it may be necessary to use multiple forms of evidence.
Secondly the trustworthiness of each item of evidence must be considered. This considers the confidence there is that the item of evidence actually delivers its expected capability. This is often also referred to as evidence integrity or evidence rigour. The trustworthiness of the evidence item is determined by the rigour of the process used to generate that item. The main parameters that will affect trustworthiness are [14]:

- Independence

- Personnel
- Methodology
- Level of audit and review
- Tools

So although the four software safety principles will never vary, the confidence with which those principles are established will vary considerably. We have seen that it is necessary to make a judgement as to the necessary confidence with which the principles must be established for any given system. This gives us our final principle.

Principle 4+1: The confidence established in addressing the software safety principles shall be commensurate to the contribution of the software to system risk.

This principle is referred to as Principle 4+1 since it *cross-cuts* the implementation of the other four principles.

Relationship to Existing Software Safety Standards

The software safety assurance principles described in this paper are inherent, though often are implicit, in most software safety standards. It is however easy for software engineers using these standards to lose sight of the core objectives by focusing solely on compliance with the letter of these standards (e.g. through box-ticking). Below we describe how each of the principles is exhibited in some of the most commonly used software safety standards: DO 178C [1], IEC 61508 [4], ISO 26262 [3].

Principle 1: The establishment of a link between hazard analysis at the system level and software safety requirements is observable in IEC 61508 and ISO 26262. DO-178C requires that "high-level requirements that address system requirements allocated to software to preclude system hazards should be defined". This addresses Principle 1, particularly when applied in the context of companion standard ARP 4754 [15].

Principle 2: The need for traceability in software requirements is universal. The standards also emphasise step-wise validation of the software requirements. DO-178C and ISO26262 provide specific models of requirements decomposition. Where the standards tend to be less strong is in capturing the rationale for the requirements traceability (a crucial aspect of Principle 2). In particular what is lacking is a particular emphasis on maintaining the intent of the software safety requirements. This requires richer forms of traceability, which consider the intent of the requirements, than merely syntactic traceability mechanisms between the requirements at different development stages.

Principle 3: Guidance on requirements satisfaction forms the core of the software safety standards. This principle is universally well addressed although there are clearly differences in the recommended means of satisfaction (for example DO-178 traditionally placed strong emphasis on testing).

Principle 4: Some aspects of this principle can be observed in the standards – it is required to show the absence of errors introduced during the software lifecycle. The software hazard analysis aspect however is the least discussed in any of the standards. The standards imply that safety analysis is a system-level process. As such, the role of software development is to demonstrate correctness against requirements, including safety requirements allocated to software, as generated from the system-level processes. The process of refining and implementing these requirements at subsequent stages in the development process does not involve any explicit application of software hazard analysis. DO 178C allows for identified safety issues to be fed back to the system level, however there is no explicit requirement to identify 'emerging' safety issues during software development.

Principle 4+1: The concept of moderating the software assurance approach according to 'risk' is common across the standards. There are crucial differences however in how the criticality of the software is determined. IEC61508 sets a Safety Integrity Level according to the probability delta in risk reduction; DO-178B places more focus on severity; ISO 26262 incorporates the concept of controllability of the vehicle. In addition, there are many differences in the recommended techniques and processes at different levels of criticality.

Conclusions

This paper has presented the 4+1 model of fundamental principles for software safety assurance. As we have discussed, the principles can be seen to relate strongly to features of existing software safety assurance standards and can provide a common reference model through which standards can be compared. Through the examples we have presented it can also be observed that, though it is possible to state these principles simply, they have not always been observed in practice, and challenges can remain in the implementation of the principles. In particular, the management of confidence in relation to software safety assurance (Principle 4+1) remains an area of significant investigation and debate.

References

[1] RTCA/EUROCAE, Software Considerations in Airborne Systems and Equipment Certification, DO-178C/ED-12C, 2011.

[2] CENELEC, EN-50128:2011 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, 2011.

[3] ISO-26262 Road vehicles – Functional safety, FDIS, International Organization for Standardization (ISO), 2011

[4] IEC-61508 – Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems. International Electrotechnical Commission (IEC), 1998

[5] FDA, Examples of Reported Infusion Pump Problems, Accessed on 27 September 2012, http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202496.htm

[6] FDA, FDA Issues Statement on Baxter's Recall of Colleague Infusion Pumps, Accessed on 27 September 2012, http://www.fda.gov/NewsEvents/Newsroom/PressAnnouncements/ucm210664.htm

[7] FDA, Total Product Life Cycle: Infusion Pump - Premarket Notification 510(k) Submissions, Draft Guidance, April 23, 2010.

[8] "Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993", Main Commission Aircraft Accident Investigation Warsaw, March 1994, http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html, Accessed on 1st October 2012.

[9] JPL Special Review Board, "Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions". Jet Propulsion Laboratory", March 2000.

[10] Australian Transport Safety Bureau. In-Flight Upset Event 240Km North-West of Perth, WA, Boeing Company 777-2000, 9M-MRG. Aviation Occurrence Report 200503722, 2007.

[11] H. Wolpe, Geneal Acounting Office Report on Patriot Missile Software Problem, February 4 1992, Accessed on 1st October 2012, Available at: http://www.fas.org/spp/starwars/gao/im92026.htm

[12] Y.C. Yeh, Triple-Triple Redundant 777 Primary Flight Computer, IEEE Aerospace Applications Conference pg 293-307, 1996.

[13] D.M. Hunns and N. Wainwright, Software-based protection for Sizewell B: the regulator's perspective. Nuclear Engineering International, September, 1991.

[14] R.D. Hawkins, T.P. Kelly, A Framework for Determining the Sufficiency of Software Safety Assurance. IET System Safety Conference, 2012.

[15] SAE. ARP 4754 - Guidelines for Development of Civil Aircraft and Systems. 1996.

Biography

R.D. Hawkins, Ph.D., Department of Computer Science, The University of York, Deramore Lane, York, YO10 5GH, UK, telephone +44 (0) 1904 325463, e-mail – richard.hawkins@york.ac.uk.

Richard Hawkins is a Research Associate within the Department of Computer Science at the University of York. His main research interests are software assurance and safety case development. He has worked on a number of research projects including the UK MoD Software Systems Engineering Initiative (SSEI), the Industrial Avionics Working Group (IAWG) and the BAE Systems Dependable Computing Systems Centre (DCSC). His work involves review

and assessment of software in a number of projects, in the aerospace and naval domains. He previously worked as a software safety engineer for BAE Systems and as a safety adviser for British Nuclear Fuels.

I. Habli, Ph.D., Department of Computer Science, The University of York, Deramore Lane, York, YO10 5GH, UK, telephone +44 (0) 1904 325566, e-mail – richard.hawkins@cs.york.ac.uk.

Ibrahim Habli is a Research and Teaching Fellow at the High Integrity Systems Engineering (HISE) Research Group at the University of York. He has previously worked as a Research Associate at the Rolls-Royce University Technology Centre at the University of York. His main research interests include safety case development, software safety assurance, software architecture design and safety-critical product-lines. He has undertaken reviews of safety critical software in a number of domains, including aerospace and automotive.

T.P. Kelly, PhD., Department of Computer Science, The University of York, Deramore Lane, York, YO10 5GH, UK, telephone +44(0) 01904 325477, facsimile +44 (0)1904 325599, e-mail – tim.kelly@york.ac.uk.

Tim Kelly is a Professor in Software and Safety Engineering within the Department of Computer Science at the University of York. His expertise lies predominantly in the areas of safety case development and management. He has published over 70 papers on safety case development in international journals and conferences and has been an Invited Panel Speaker on software safety issues. He has extensive experience as a consultant, giving advice on the development and assurance of safety critical software in a number of domains, including aerospace naval and medical.