Notes on Properties Needed in Software Safety Requirements

Peter Bernard Ladkin, Bernd Sieker
Working Draft 3, 2014-03-05

RVS White Paper 7

The terms “requirements” and “requirements specifications” in this note refer exclusively to functional requirements.

We are concerned in this note exclusively with safety requirements. Safety requirements partake of the characteristics of system requirements, but they also have a more specific character which is not necessarily to be found in general system requirements.

Introduction: Development of SW to Requirements and Contrast with Hardware Design

The physicist Richard Feynman commented during the Challenger-accident inquiry (ref) on top-down versus bottom-up design:

> *"The engine is a much more complicated structure than the Solid Rocket Booster, and a great deal more detailed engineering goes into it. Generally, the engineering seems to be of high quality and apparently considerable attention is paid to deficiencies and faults found in operation.*
>
> *"The usual way that such engines are designed (for military or civilian aircraft) may be called the component system, or bottom-up design. First it is necessary to thoroughly understand the properties and limitations of the materials to be used (for turbine blades, for example), and tests are begun in experimental rigs to determine those. With this knowledge larger component parts (such as bearings) are designed and tested individually. As deficiencies and design errors are noted they are corrected and verified with further testing. Since one tests only parts at a time these tests and modifications are not overly expensive. Finally one works up to the final design of the entire engine, to the necessary specifications. There is a good chance, by this time that the engine will generally succeed, or that any failures are easily isolated and analyzed because the failure modes, limitations of materials, etc., are so well understood. There is a very good chance that the modifications to the engine to get around the final difficulties are not very hard to make, for most of the serious problems have already been discovered and dealt with in the earlier, less expensive, stages of the process.*
>
> *"The Space Shuttle Main Engine was handled in a different manner, top down, we might say. The engine was designed and put together all at once with relatively little detailed preliminary study of the material and components. Then when troubles are found in the bearings, turbine blades, coolant pipes, etc., it is more expensive and difficult to discover the causes and make changes. For example, cracks have been found in the turbine blades of the high pressure oxygen turbopump. Are they caused by flaws in the material, the effect of the oxygen atmosphere on the properties of the material, the thermal stresses of startup or shutdown, the vibration and stresses of steady running, or mainly at some resonance at certain speeds, etc.? How long can we run from crack initiation to crack failure, and how does this depend on power level? Using the completed engine as a test bed to resolve such questions is extremely expensive. One does not wish to lose an entire engine in order to find*

out where and how failure occurs. Yet, an accurate knowledge of this information is essential to acquire a confidence in the engine reliability in use. Without detailed understanding, confidence can not be attained.

"A further disadvantage of the top-down method is that, if an understanding of a fault is obtained, a simple fix, such as a new shape for the
turbine housing, may be impossible to implement without a redesign of the entire engine."

[Richard P Feynman; Personal observations on the reliability of the Shuttle; Appendix F of the Rogers Report on the Challenger Enquiry, 1986.]

In contrast to this HW design, the safety requirements for complex SW according to IEC 61508 derive from overall system safety requirements and are thus derived quasi-top-down. The reason for this is that SW as an object in itself cannot logically be adequately or inadequately safe – SW just sits there. It is the actions of the HW on which it executes which are adequately or inadequately safe. Some of these actions are SW-controlled, and the logic of the SW may well lead directly to inadequately safe actions of the HW. It is in this sense one speaks of "software safety".

IEC 61508 follows this quasi-top-down logic: IEC 61508-3:2010 Figure 2 Box 9 shows the E/E/PE System Safety Requirements Specification (SRS) phase, which "feeds in" to IEC 61508-3:2010 Figure 4 Box 10.1, the SW SRS phase.

The Waterfall Model and the Intermingling of SW Requirements and Design Stages

One model of SW development still useful for conceptual purposes is the Waterfall model (which originated with Winston Royce of Lockheed Martin in the 1970's. It is often expressed in a diagram, see for example *Section 6.1.2, Figure 6.2, of Parametric Estimating Handbook, 4th edition, International Society of Parametric Estimators, April 2008.* For our purposes, a textual statement, below, suffices). Royce understood that this model was at best a post-hoc conceptualisation of SW development and that practical development would deviate from it.

The Waterfall model is roughly as follows. The task of developing SW is proposed to follow the linear task progression

Requirements Specification → Design → Coding → Testing → Deployment

The progression can be refined; for example, a distinction may be profitably made between unit testing and integration testing for embedded SW-based systems. There are also steps in SW development which this progression does not show. For example, when a SW-based system has many simultaneously active components, then the SW functions will be partitioned amongst the components, or be realised through joint action of several components; the partitioning task is not explicitly shown above.

In their paper "On the Inevitable Intertwining of Specification and Implementation", Comm. ACM 25(7):438-440, July 1982, William Swartout and Robert Balzer gave an example that demonstrated definitively that in the general case results of actions and decisions in "later" phases in a Waterfall model unavoidably influence "earlier" phases in the model. The example they gave was not SW-based but was algorithm-based, a package-sorting system. It was not necessarily practical, but that was not the point.

At the time of writing of this paper, 1982, many influential developers of SW believed that a Waterfall-type development cycle was the best way of comprehending and managing what they did.

A common justification for this belief was as follows: a requirements specification says "what" the system is to do and a design specification says "how" it does it, and the “what” must come before the “how”. Specifically, theoretically you could take all the detailed tasks involved in specifying a real system and assign them a ranking “R” (for requirements-step, or “what”-step) and “D” (for design-step or “how”-step) and (re)order the system development (or, as David Parnas would have it, construct the system-development documentation) so that all “R” steps are completed before all “D” steps. That plausibly cannot be accomplished with the example of Swartout and Balzer. They couldn't and didn't prove that rigorously, because proving a negative (“can't be done”) means you have to be very ingenious in showing how you have thought of all possible cases (this issue occurs as well in both requirements specification and in hazard analysis!). So the paper gives a set of plausible reasons for thinking it cannot be done, rather than a formal proof, and issues an implicit challenge, to show how, to anyone who might think such an ordering, putting the R's before the D's, can be accomplished in a principled manner. We believe their challenge succeeds. This vitiates, in particular, the "what" and "how" explanations of the difference between requirements and design.

This is not just a theoretical result of interest mainly to academics, but is an academic justification of a pervasive practical phenomenon that limits any use of the Waterfall model. That the intermingling of requirements and design is inevitable in the practice of general software development has also been emphasised by Michael Jackson. The following is taken from his book System Development, Prentice-Hall, 1983, describing his system development method Jackson System Development (JSD), Section 14.5:

> *JSD is not top-down. This is a proud claim, not an embarrassed confession. For many people, 'top-down' is a synonym of 'good': a development method, if it is to claim serious consideration, must be able to claim that it is 'top-down' and, preferably, 'structured' too. The esteem in which top-down methods are held is greatly strengthened by the good repute of their more academically flavored cousin, stepwise refinement.*
>
> *The fundamental idea of top-down development is that the object to be developed, whether it is a small program or a large system, can best be regarded as a hierarchy. The top-down developer begins by stating the highest level of the hierarchy, the decomposition into a small number of large objects: then, each of these is further decomposed into a small number of smaller objects, and so on, until the lowest level is reached. The name 'top-down' is appropriate because development begins at the top of the hierarchy and works down, level by level, until the bottom is reached. The hierarchy may be a hierarchy of subroutines, or it may be a hierarchy of data-flow diagrams; in either case there is a level-by-level decomposition from the top level downwards.*
>
> *Top-down is a reasonable way of describing things which are already fully understood. It is usually possible to impose a hierarchical structure for the purposes of description, and it seems reasonable to start a description with the larger, and work towards the smaller aspects of what is to be described. But top-down is not a reasonable way of developing, designing, or discovering anything. There is a close parallel with mathematics. A mathematical textbook describes a branch of mathematics in a logical order: each theorem stated and proved is used in the proofs of subsequent theorems. But the theorems were not developed or discovered in this way, or in this order; the order of description is not the order of development.*

Costs Mirror Waterfall

Although the reality of developing SW practically may well require the intermingling of requirements and design stages, and indeed specifications, the logic of safety requirements specifications in IEC 61508-3:2010 strictly follows the "Waterfall" logic, as noted earlier.

Furthermore, it has been argued by many that the cost of discovering error follows a similar logic. A "Pyramid Model" of the cost involved in rectifying errors says that discovering an error at a particular development stage costs some fraction 1/X of the cost of discovering it at one stage later. Rectifying an error discovered at requirements-specification/analysis costs 1/X of the cost of discovering it at the SW Design stage; discovering it at SW Design stage costs 1/X of the cost of discovering it at unit-testing stage; and so on. Obviously, one can only discover an error at the requirements stage if it is a requirements-based error. Another way of expressing the cost model is that discovering a requirements error during design costs X times as much as discovering it at requirements specification/analysis time; discovering it during coding costs X^2 as much; discovering it during testing costs X^3 as much, and during deployment of the system X^4 as much.

Obviously, this is only a rule of thumb. It is also very dependent upon the number of stages conceptualised. For example, mathematically, no parameter 1/X can be consistently derived for the detailed "Waterfall"

Requirements Specification → Design → Coding → Unit Test → Integration Test

which can be made to cohere with a parameter 1/Y for the simpler "Waterfall"

Requirements Specification → Design → Testing

no matter what values of X and Y are tried.

Whatever the status of this rule of thumb as a piece of organisational science, there is little practical doubt that some cost proportionality of this sort is involved: that discovering errors early saves resources over discovering them later. Some development methods such as Altran's Correct-By-Construction (CbC) aim to avoid, or otherwise detect and resolve errors at the stage at which they are made (see, for example, *Correct by Construction: Better Can Also Be Cheaper, Peter Amey, Crosstalk: The Journal of Defence Software Engineering, March 2002*, available at http://www.macs.hw.ac.uk/~air/rmse/c_by_c_better_cheaper.pdf).

A "folded" version of the Waterfall model, the so-called (standardised) V-Model of SW development (*IEC 61508-3:2010, Figure 5*, also *EN 50126*) relates testing phases to the forward development phases. Later testing phases test earlier forward-development phases; in the V-Model, the approximate truth of the "Pyramid Model" of costs of error-discovery are more intuitively clear.

The values of X associated with the Pyramid Model vary between 5 and 10 (*Ralf Schumacher, personal communication 2013-11-22; Morris Chudleigh, personal communication + literature ??*).

It follows that SW development has different costs associated with error discovery from those associated with HW development according to Feynman; and a crude cost model coheres with a top-down construal of SW development. This means that, whatever may be the reality of developing SW, detection of an error near the "top" of the right side of the V-model costs proportionately more than detection of an error "lower down".

Is a Construction of SW Safety Requirements "Top-Down" Appropriate?

IEC 61508 requires the addition of safety functions to an EUC and EUCCS when the operation of

the latter does not assure an appropriate level of safety in itself. The safety of the system is assessed in part by how reliable the safety functions must be – they are assigned Safety Integrity Levels (SILs) which specify how (in)frequently they may fail while assuring an appropriate level of safety. So the appropriate safety of a system is in part assured by appropriate design of the EUC and EUCCS, and in part through the reliability of any safety functions which have been added as required.

We have noted the argument of Feynman that a detailed understanding of the functioning of a system is needed to ensure the reliability of a complex engineered system, and that a system is most appropriately designed for reliability by starting with these details (and, presumably, composing subsystems as needed). In contrast, the Waterfall model posits a "top-down" SW system development. We have noted that it is best understood as an idealisation, more appropriate for documenting the development of a SW system than for guiding the SW development process. It is in practice unhelpful (Jackson et al.) and cannot be realised for some systems in theory either (Swartout and Balzer). However, it seems to help in assessing the costs of error detection at different stages in development.

It is inherent in IEC 61508 that an initial SW Safety Requirements Specification is derived from the activity in Lifecycle Box 3, the PHA, which must actually (and not notionally) precede SW development. This is not only notionally but also practically "top down". This goes against the arguments we have rehearsed that SW development as a whole does not proceed according to a Waterfall model. Can this top-down development of SW Safety Requirements Specification be justified? We believe so. The argument follows.

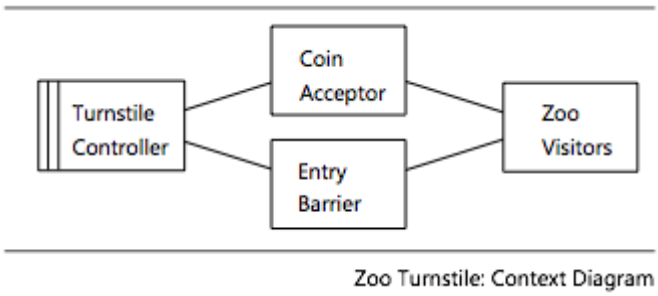
System Design is Dependent upon and Follows the Role of the Customer in the Wider Context

Michael Jackson argues that a SW developer must determine bounds on the problem-world (the context in which the system is to operate) and the problem (what problem should be solved?). These bounds can be derived from considering the context of the customer. It is well argued through an example in this extract from his book on problem frames (*Michael Jackson, Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley and ACM Press, 2001, Chapter 2 Section 3*):

> ***THE REAL PROBLEM***
>
> *Over the past fifty years, we software developers have built up a reputation for solving the wrong problem. We don't listen hard enough to what our customers say. We don't take the trouble to understand their world well enough. We don't look behind what they say, and interpret what they really mean, or what they ought to be saying. We don't solve the real problem, which lies behind the superficial problem that they tell us about.*
>
> *Well, there's some truth in all this, but it's important not to get carried away. In particular, it's important to resist two temptations. One is to generalise the problem context, and abstract away the specific details of the customer's world. The other is to generalise the requirement -- to keep asking 'Why?', to try to get at the 'real problem' behind the problem the customer is talking about. Here's a little example.*
>
> ***Zoo Turnstile Problem***
>
> *Our customer is the visitor admissions manager of a private zoo. The zoo has bought a small turnstile system consisting of a rotating barrier and a coin- accepting device, that can be connected to two ports of a small computer. Our job is to build the software to operate*

the turnstile system. The requirement is, essentially, in two parts. First, no visitor should be able to enter the zoo without having paid the entry price (which is two coins). Second, any visitor who has paid the two coins should be allowed to enter.

The obvious context diagram looks something like this:



Zoo Turnstile: Context Diagram

Generalising the Problem Context

The first danger is generalising the problem context and abstracting away the realities of the customer's world. The real problem is to collect entrance fees from visitors. The coin acceptor is just an implementation, which we should ignore. Would a credit-card device be better? What about pre-payment by post, or on the web? We might question the whole idea of using a turnstile. Perhaps an escalator would work better? There will be an attendant near the gate. Would it not be better for the attendant to collect the entrance fee from each visitor? This personal service would be more attractive than the harsh impersonality of the turnstile. If you think like this, you're treating the turnstile and the coin acceptor as obstacles to your understanding of the true, more abstract, problem. It's a bad mistake.

Generalising the Problem Requirement

The second danger is to look behind the problem that's given, and to keep asking: Why? Software development always serves a human purpose, and that purpose can always be subsumed in a larger purpose. Why is the zoo collecting entrance fees at all? Because the owners want to raise revenue? Perhaps it would be better to follow the lead of some successful web businesses: let the visitors enter free, and collect revenue from advertisement hoardings placed on the animals' cages. Perhaps a zoo with free admission could form the basis of a very profitable television series. But why do they want to raise revenue? Is it to achieve a return on their investment? Perhaps they could get a higher return by demolishing the zoo and using the site for an expensive residential development. But is return on investment really the goal? Are we not all really pursuing happiness? Would the owners not be happier living a less materialistic life altogether? Why not make reservations for them at a spiritual retreat? This way madness lies.

The Customer

There is no stopping place in this progression towards 'the real problem' unless you can anchor yourself in something reasonably firm. The firmest anchorage is with your customer.

Your customer is the visitor admissions manager of the zoo, and in that role has certain limited authority and responsibilities. Most of our imaginary questionings raise issues that lie far beyond those limits. It is no part of the admissions manager's job to reorganise the zoo's owners' lives on a more spiritual basis, or to move them into the real estate business, or even to abolish zoo entrance fees. And the reduced possibilities that then remain open don't include scrapping the expensive new turnstile equipment: the visitor admissions manager has no authority to write off such an investment. So the context diagram is probably just about right, after all. A safe anchorage.

The zoo example is playful, of course. And most software developments don't have a single easily identified person who is the customer for the problem solution. But the fundamental point remains valid. Even if your customer is purely notional -- for example, the management committee that is your project's sponsor, or that scattered group of people who are sometimes called the stakeholders -- you can still usefully ask yourself a vital question: Where are the limits of their combined authority and responsibility? And when you are in doubt about the location and scope of the problem, use those limits as a touchstone.

The problem requirements must not be too small in relation to the customer's responsibilities. That places a lower bound on the domains that must appear in its context diagram. The visitor admissions manager is not just responsible for the turnstile, but also for the way the Zoo Visitors are treated when they are trying to get into the zoo. How you solve the Turnstile problem will certainly affect that treatment. So the Zoo Visitors must appear in your context diagram. And the customer's authority limits the scope of what the machine may legitimately be designed to do and on what assumptions: it places an upper bound on the domains that may appear in the problem context and be affected by the machine. If you're thinking of making the machine unlock the lions' cages when there are no visitors in the zoo, you have stepped outside your proper bounds. The visitor admissions manager has no authority to affect the zoo animals at all. Lions' cages should definitely not appear in your context diagram.

Here's the conclusion. When you are thinking of generalising, abstracting or extending the problem context or requirements, you should ask yourself: Given who the customer is, must this requirement be in scope? Can it be in scope? And what are the consequences for the context diagram?

How Far Into the World?

Identifying the customer's authority and responsibility can help you to avoid broadening the problem too far. But it can also help to avoid narrowing it too much. We included the Zoo Visitors domain in the context diagram for the Zoo Turnstile problem, because the customer is responsible for visitor admissions, not just for turnstile operations. By introducing the Zoo Visitors into the problem, you are undertaking to study how they behave. That could save you from a serious error in your software development.

Here's how. At first sight you would expect each visitor to place two coins in the coin acceptor and then to pass through the turnstile. The sequence of events would then be:

coin, coin, pass, coin, coin, pass, coin, coin, pass, coin, coin, pass, ...

If the coin acceptor is very close to the barrier, this seems quite plausible. Each visitor steps up to the barrier, puts the coins in the acceptor, and passes through. You could develop your machine to enforce this sequence. But some visitors may be a little impatient, especially

when the zoo is crowded or closing time is approaching. An impatient visitor may try to hurry things along a little by putting her coins in while the visitor in front has not yet passed through the barrier. And, now that we come to think of it, it's quite likely that a schoolteacher bringing twenty children to the zoo would stand at the coin acceptor putting in forty coins as fast as possible while the children stream through the barrier. The sequence of events might then be:

> *coin, coin, coin, pass, coin, coin, coin, coin, pass, pass, coin, coin, coin, pass, ...*

The machine you need must allow this sequence too.

Without the Zoo Visitors domain you might perhaps think of these possibilities spontaneously; or the customer might tell you. But by putting the Zoo Visitors domain into the context diagram you make your development process more reliable: you impose on yourself a definite and explicit obligation to think about and describe the Visitors' possible behaviour, and to analyse its consequences.

Any Questions?

> *[Q] Do we really want to 'anchor ourselves' by the customer's responsibilities and authority? Perhaps as ethical software developers we have a responsibility to suggest that the spiritual life is preferable to the materialistic life.*

[A] If you think so, perhaps you have. But that would be in your capacity as the customer's friend and confidant, not in your capacity as a software developer.

> *[Q] I'm absolutely sure that I would think about the Zoo Visitors anyway, without putting them into the context diagram.*

[A] Of course you would. But then again, there are many software developers who might sometimes forget. This section is for them.

> *[Q] Will we make a model of the Zoo Visitors in our solution to the Zoo Turnstile problem?*

[A] No. Just a description. Not everything worth thinking about has to result in a piece of program text or a designed domain.

And conversely, the fact that there will be no model of the visitors doesn't mean that you don't have to describe them in your problem analysis.

Safety Requirements Specification from the PHA Logically Precedes SW Development

As noted above, IEC 61508:2010 takes a very Waterfall-y view of the derivation of the Safety Requirements Specification. How is this to be reconciled with the inevitable intermingling for which there are both theoretical and practical arguments? An answer lies, we suggest, in the observations of the customer role in the extract from Jackson's book above.

As we have noted, SW by itself does not take action, and can therefore not by itself take actions which may cause harm. Rather, it expresses logic which governs the actions taken by HW on which it is executed. In order to derive safety requirements for SW, it is therefore necessary to consider the range of actions available to the hardware on which it runs, and the results of those actions within

the overall systems and the system's operational environment. If any of these actions may cause harm, elimination and mitigation (by means of safety functions in IEC 61508) must be considered, and such elimination and mitigation measures will transitively impose requirements on the SW logic. These requirements are expressed in the SW Safety Requirements Specifications.

In general, it will be the case that the personnel who perform the Box 3 Hazard Analysis (HazAn) and Risk Analysis (RiskAn), and who devise elimination and mitigation measures, will be Haz/RiskAn specialists, system safety specialists, and not necessarily SW developers, SW specification experts or programmers. The “customer” of the SW developer in a IEC 61508-conformant development, in Jackson's sense, is here the system safety analyst who performs the Box 3 actions (PHA). The fact that these are two different roles entails an assignment of detailed steps in the SW development to “system safety analysis” and “SW development”. Can we argue that all “PHA” steps can logically be taken to precede all “SWD” steps (in contrast to the Swartout-Balzer example for system implementation)? We believe so.

Let us assume here that the PHA has resulted in a complete enumeration of hazards; that is, a complete list of all the ways in which actions or change to the HW may result in harm.

The physical functioning of the HW which may result in harm, and a prohibition on certain kinds of action, are most obviously expressed entirely in terms of the physical constitution of the HW and the actions and other change phenomena in which its physical components engage. Describing and constraining such actions and change may be and is undertaken in most cases in a vocabulary which does not refer at all to any of the concepts used in SW development; the vocabularies are disjoint (we elaborate).

Note that this is not to say that, for example, variable names in the SW may not mimic objects in the physical hardware – indeed, in most understandable SW they often do. The concept of a variable in the SW is bound necessarily by the semantics of the SW source language to the concept of a memory location in HW. That a memory location must or must not contain a certain bit pattern or undergo/not undergo a certain sequence of bit patterns is rarely if ever an initial safety requirement. Bit patterns in themselves rarely if ever result directly in harm, but do so in virtue of the causal relations they have to other system objects which may directly cause harm, through motion or some other change. (This claim is prone to contrived counterexamples, hence the caveat “rarely if ever”.) The PHA leads to the formulation of safety requirements concerning the objects which may directly cause harm, and these are not the bit patterns but those non-digital-HW objects to which those bit patterns are causally related.

This is also not to say that the causal connection between a bit pattern or sequence of bit patterns and potentially-harm-causing system object is not direct – for it may indeed be direct. But a specific bit pattern cannot be isomorphic to a harm-causing action; in particular it cannot *be* a harm-causing action, for a pattern may not be an action, and a change of pattern is not in itself harm-causing. The vocabulary of SW, even when it refers directly to items of digital-computing HW, is thus disjoint from that of harm-causing actions (QED).

Thus harm-causing actions, identified by the PHA, are disjoint from the possibly-digital-logical *causes* of those actions. Harm-causing actions may be labelled “HCA” in a causal graph, and their causes “CHCA” (there may be many causes of an HCA, and those causes may include events and states). By the metaphysics of causality, in any occurrence of an HCA a CHCA will precede it in a causal graph (called by us a Why-Because Graph or WBG). Thus a constraint upon a CHCA is derivative upon a need to eliminate or mitigate the HCA. The HCAs are identified by the PHA. The results of the PHA manifest themselves in the constraints on HCA represented by the Safety Requirements Specification derived from the PHA. This SRS may be taken logically to precede the

Software Safety Requirements Specification, because (amongst other things) its vocabulary is disjoint.

Instantiation of This Precedence in IEC 61508

The logical precedence of PHA over SW development activity is instantiated in IEC 61508-1:2010 as follows.

According to IEC 61508-1:2010 Figure 2, the first stage of system development is "concept"; then comes "overall scope definition"; followed by "hazard and risk analysis". What these are is specified in IEC 61508-1:2010 Table 1: Overall safety lifecycle – overview, and in more detail in clauses 7.2 (concept), 7.3 (overall scope definition) and 7.4 (hazard and risk analysis). The purpose of the hazard and risk analysis at this stage is, according to Table 1

> *To determine the hazards, hazardous events and hazardous situations relating to the EUC and the EUC control system (in all modes of operation), for all reasonably foreseeable circumstances, including fault conditions and reasonably foreseeable misuse (see 3.1.14 of IEC 61508-4); To determine the event sequences leading to the hazardous events; To determine the EUC risks associated with the hazardous events.*

IEC 61508-1:2010 clause 7.4.1 contains three subclauses which specify the activities just noted. Clause 7.4.2.1 specifies what the hazard and risk analysis shall address:

> *A hazard and risk analysis shall be undertaken which shall take into account information from the overall scope definition phase (see 7.3). If decisions are taken at later stages in the overall, E/E/PE system or software safety lifecycle phases that may change the basis on which the earlier decisions were taken, then a further hazard and risk analysis shall be undertaken.*

The hazard analysis to be undertaken at this stage is often known as the PHA (for example, see op.cit. Table 1, Row 3 "hazard and risk analysis", column "Scope", where these words are used). The analysis at this stage is concerned exclusively with

- The EUC
- The EUC control system (EUCCS)
- Any "human factors" (involving "use" and "reasonably foreseeable misuse")

and not, for example, with hazards which arise from "later" implementation of a required functionality, say use of a piece of kit from a specific manufacturer which has strict temperature limits on its specified functionality.

There is a reason why the safety requirements specification derived from the PHA is more rigid and less fluid than general requirements. The PHA identifies hazards with the operation of the system, with the EUC and EUCCS and its human operation, which pertain no matter how the resulting system is implemented, including what is implemented in HW and what in SW and how. The safety requirements specifications derived via IEC 61508-1:2010 clause 7.5.2.1 refer specifically to these hazards identified by the PHA (required in clause 7.4.1 as above):

> ***7.5.1 Objective***
> *The objective of the requirements of this subclause is to develop the specification for the overall safety requirements, in terms of the overall safety functions requirements and overall safety integrity requirements, for the E/E/PE safety-related systems and other risk reduction*

measures, in order to achieve the required functional safety.

.....

7.5.2 Requirements

7.5.2.1 A set of all necessary overall safety functions shall be developed based on the hazardous events derived from the hazard and risk analysis. This shall constitute the specification for the overall safety functions requirements.

These safety requirements specifications are thus impervious to details of implementation at the level of SW specifications. They must be assured throughout the development of this system, defined by the EUC and EUCCS. This explains why the overall safety requirements specification is prior to and "input" to the SW specification, as in Box 9 of IEC 61508-3:2010 Figure 2.

We note that the argument we have given for the PHA to have logical precedence over SW safety requirements specification depends upon the assumption that the PHA is in some sense complete, that is, that all HCAs of the EUC and EUCCS at this level of construal (given by the architectural-language constraint) are explicitly identified in the PHA. If there is not some kind of assurance of completeness at the time of PHA performance, then there cannot be an argument that a hazard analysis at any stage of development, including at SW safety requirements derivation time, would not identify previously-unidentified hazards, for it indeed may do so.

A recommendation for assessing safety requirements specifications for relative completeness is included at the end of this paper.

Sources of Error; What Must be Avoided, What Must be Assured

We have noted that SW may be regarded as appropriately or inappropriately safe derivatively from the actions of the HW which it directly commands. Michael Jackson quantifies three ways in which software may behave in an unintended fashion (error) in such a way that the SW commands can be suitably considered to be the sole cause of inappropriately safe HW actions:

By 'safety' ..... we mean some set of properties of
the behaviour of the problem world outside the computer. Within the envelope of the given characteristics of the relevant parts of the problem world, this behaviour will be governed by the behaviour of the computer executing the controlling software.

Complexity in software design suggests that someone may not have understood something well enough. Among the many ways to misunderstand are:

1. *Making a 'pure software error'. The programmer's intention was correct, but the code does not achieve it.*

2. *Misunderstanding the problem world behaviour required. The code achieves the programmer's intention, but the intention---possibly expressed in a specification--- was faulty: valve W should be opened, not valve V.*

3. *Misunderstanding the complexity of the intended behaviour in the problem world. Software may be complex because it tries to satisfy two functional requirements whose combination has not been adequately understood. This is the 'feature interaction' problem.*

[Michael Jackson, Contribution to System-Safety Mailing List, 2013-06-26

http://www.systemsafetylist.org/0258.htm]

When asked what properties of system-requirements should be required by a standard, Martyn Thomas refers to his US National Academy of Sciences study with Daniel Jackson and Lynette Millet [*Daniel Jackson, Martyn Thomas and Lynette Millett, eds., Software for Dependable Systems – Sufficient Evidence? NAP, 2007* http://www.nap.edu/openbook.php?record_id=11923] where it was concluded

> *that the properties that were being claimed for the system should be specified rigorously and unambiguously and that sufficient, independently verifiable evidence that the system has these properties under all the claimed operating conditions should be provided. That still seems a reasonable requirement.*
>
> *Whether those properties are sufficient to achieve and preserve the behaviour and properties that the owner, operator or regulator of the system requires in the real world is a different issue, owned by the owner, operator or regulator of the system. Again, sufficient evidence should be provided.*

(*Martyn Thomas, personal communication to Peter Bernard Ladkin, 2013-08-02*)

Martyn Thomas is founder of the company Praxis Ltd (which became Praxis High Integrity Systems, Altran Praxis, and now Altran UK). Praxis set about developing software using formal techniques to ensure freedom from as much error as possible, and is a pioneer in Correctness-by-Construction (CbC) methods. Praxis used a requirements-specification and -analysis technique called REVEAL, based on Michael Jackson's Problem Frames approach. REVEAL as used at Praxis used the formal language Z to specify requirements, having taught the domain experts responsible for formalising requirements to read Z in a few days' tutorial (with which Praxis/Altran has had considerable success *[Rod Chapman, personal communication with Peter Bernard Ladkin many dates]*). Martyn Thomas has emphasised how the formal approach using Z enabled Praxis to analyse requirements as given to them for consistency and for relative completeness, and how this approach was essential to Praxis's business model involving CbC development *[Martyn Thomas, personal communication with Peter Bernard Ladkin, many dates]*.

Consistency

By consistency is meant here freedom from contradiction. Requirements are contradictory when it is not logically possible for them jointly to be satisfied. Obviously, if it is not logically possible for requirements jointly to be satisfied, then any hope of building a system to satisfy them is futile, for it is impossible. Martyn Thomas's requirement that the system shall be shown analytically to satisfy the properties claimed cannot be fulfilled if the system requirements are inconsistent; his requirement entails that the system requirements are consistent.

However, ensuring consistency of requirements specifications is not trivial.

Michael Jackson, in his paper Topsy-Turvy Requirements *[Michael Jackson, Topsy-Turvy Requirements, Requirements Engineering, published online August 2013, http://link.springer.com/article/10.1007/s00766-013-0179-2]* relates two studies of requirements in engineered systems, one of a Flight Guidance System for a complex aircraft by Miller, Tribble, Whalen and Heimdahl.

> *In a paper [Steven Miller, Alan Tribble, Michael Whalen and Mats Heimdahl; Proving the Shalls; International Journal on Software Tools for Technology Transfer (STTT) 2006]*

presented at FME'03 in Pisa, Steve Miller, Alan Tribble and Mats Heimdahl describe how the requirements for the mode logic of a Flight Guidance System, expressed as "shall" statements in natural language, were formalised in CTL and checked against a large formal model of the intended system behaviour [Mats P E Heimdahl; Let's Not Forget Validation; In Proceedings of VSTTE Workshop, ETH Zurich, 2005, Bertrand Meyer and Jim Woodcock eds; Springer LNCS 4171, 2008, op. cit.]. The requirements were initially expected to be complete, unambiguous and consistent: inevitably, they were not. When errors were found, and corrections chosen, the affected informal requirements were modified. Virtually all the informal requirements had to be modified in this way.

(The references have been expanded from the original.) Jackson gives another example deriving from David Harel:

David Harel, in a paper [David Harel; Statecharts in the Making: A Personal Account; Communications of the ACM Volume 52 Number 3 pages 67-75, March 2009] describing the origins of statecharts, also criticises requirements expressed in the fragmented style. He recounts how the voluminous requirements of a chemical manufacturing plant included the following three specifications of a tiny piece of behaviour, buried in three totally different locations:

"If the system sends a signal HOT then send a message to the operator" "If the system sends a signal HOT with T>60deg then send a message to the operator" "When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when T<60deg."

Harel points out the redundancy, inconsistency and logical confusion in these three statements, and goes on to explain how, participating in the development of an avionics system, he was led to develop the language and semantics of statecharts.

The fundamental point arising from these examples, as well as Michael Jackson's view, is that consistency of requirements cannot be assumed and needs to be checked.

The [requirements] specification gives complete, exact and objective criteria for the acceptability of a proposed solution, and so provides a firewall between the distinct obligations of whoever poses the problem and whoever undertakes to solve it. The aspiration to bring as much as possible of a system requirements specification within this salutary firewall is clearly seen in IEEE Std 830-1998:

*"An SRS [here: **Software** Requirements Specification] should be: (a) Correct; (b) Unambiguous; (c) Complete; (d) Consistent; (e) Ranked for importance and/or stability; (f) Verifiable; (g) Modifiable; (h) Traceable."*

[Michael Jackson, personal communication to Peter Bernard Ladkin, September 2013].

Both Michael Jackson and Martyn Thomas thus consider that consistency of requirements is an essential property which must be assured. Since consistency is a logical property, it must be checked using tools of logic. Experience of logicians with consistency has shown that it is a property that one cannot reliably check by hand in many cases.

Completeness and Relative Completeness

By "completeness" is meant here that a requirements specification should specify system behavior

in a suitable fashion, for every eventuality which occurs during operation of a system. It is widely regarded as all-but-impossible so to do, being dependent on perfect foresight. However, it is possible to check specifications for completeness exhaustively against some set of explicitly-given criteria. I refer to this as relative completeness (relative to the explicitly-given criteria).

There is an issue of relative completeness of requirements specifications. The IEEE Std 830-1998 requires that requirements-specifications are complete (see above). Some notable studies of requirements regard inspection of completeness as well as consistency as essential *[Mats Heimdahl and Nancy Leveson, Completeness and Consistency of Hierarchical State-Based Requirements, IEEE Transactions on Software Engineering 22(6), June 1996 http://sunnyday.mit.edu/papers/mats-tse.pdf ; Nancy Leveson, Completeness in Formal Specification Language Design for Process-Control Systems, Proceedings of the Conference on Formal Methods in Software Practice, August 2000, http://sunnyday.mit.edu/papers/completeness.pdf]*. Heimdahl and Leveson verified and validated the requirements for the TCAS II aerial collision avoidance system. The first reference relates their use of a semi-formal state-machine-based language SpecTRM to do so.

(Incidentally, the TCAS example validates Jackson's point 3 above. It turns out that TCAS II has some properties which allow it to *contribute to* aerial collision. See *[Peter Bernard Ladkin, Causal Analysis of the ACAS/TCAS Sociotechnical System, in Safety Critical Systems and Software 2004, the Proceedings of the 9th Australian Workshop on Safety-Related Programmable Systems, volume 47 of Conferences in Research and Practice in Information Technology, ed. Tony Cant, Australian Computer Society, 2005.])*

A summary of Leveson's criteria for requirements-specification completeness may be found in lecture-presentation form (*Requirements, Design, HCI, Nancy Leveson, lecture slides 2006*, available at http://sesam.smart-lab.se/seminarier/Leveson08/Sweden-day2.pdf).

Other references also emphasise the need for completeness (op. cit. in Section **Consistency** above). In view of Michael Jackson's point 3, and the TCAS example just given, the best one may reasonably hope for is a form of relative consistency, relative to the problem frame in which the requirements have been developed, respectively the language and ontology in which the requirements are specified. Peter Bernard Ladkin has summarised a set of reasons for the importance of various notions of relative completeness of safety requirements *[Peter Bernard Ladkin, The Parable of the Exploding Apples, Abnormal Distribution Blog, 2010-11-09, http://www.abnormaldistribution.org/2010/11/09/the-parable-of-the-exploding-apples/]* as well as a formal definition of safety requirement which incorporates the property of relative completeness, relative to the language in which requirements are expressed *[Peter Bernard Ladkin, Formal Definition of the Notion of Safety Requirement, Abnormal Distribution Blog, 2010-11-09, http://www.abnormaldistribution.org/2010/11/09/formal-definition-of-the-notion-of-safety-requirement/]*.

In summary, completeness of requirements, and therefore of safety requirements, is regarded to be very desirable or essential. There are various theoretical and practical notions of completeness in the literature. We conclude that each of these may be best termed a form of relative completeness; that is, one chooses a set of criteria for a requirements specification to be complete, and evaluates the specification with regard to the criteria. The choice of criteria is just that: a choice. There are no proofs of equivalence of criteria of which we are aware. The completeness assessment is thus relative to the choice of criteria.

Formality

Checking requirements specifications by hand for consistency is not a reliable process. Thus,

checking safety requirements specifications by hand for consistency is unlikely to be a reliable process. Some reasons for this are

1. human beings are not reliable or complete inference machines;
2. functional requirements specifications may be quite intricate;
3. the informal terms, usually in natural language, in which functional requirements are often expressed for human readability, allow ambiguities in the use of informal terms to obscure inconsistencies, so that requirements can appear to a human to be consistent when they are in fact not consistent when the ambiguities have been removed.

Point 3 demonstrates the importance of *freedom from ambiguity* in the statement of requirements. Ambiguities are most easily identified through use of a formal specification language in which the informal natural-language requirements are translated. Ambiguities are thus identified "by hand". It should be noted that freedom from ambiguity is not a property of requirements specifications which contributes by itself to the effectiveness of the specifications, as do consistency and relative completeness; rather, it is a property of such specifications which allows the application of means to ensure consistency and relative completeness.

Formal specification languages are often supported by SW tools which help automatically to determine consistency (such SW tools are automated theorem provers, which is often a misnomer, or proof checkers, or model checkers). SW tools may also help to ascertain relative completeness of requirements specifications written in a formal language (through use of model checkers).

Requirements on Safety Requirements Specifications in IEC 61508

There are currently no requirements on safety requirements specifications in IEC 61508:2010 for

- Checking safety requirements specifications for consistency
- Checking safety requirements specifications as far as possible for relative completeness
- Checking safety requirements specifications for freedom from ambiguity

IEEE Std 830-1998 requires all these properties of *software* requirements specifications. The studies and citations above emphasise these properties as essential, and all-too-often violated (because easy to overlook). It is particularly important to check safety requirements for consistency and completeness, because inconsistent safety requirements specifications are not fit for purpose (because they cannot be realised), and incomplete safety requirements specifications may allow unspecified unsafe situations to develop during operation of the system.

Recommendation

Requirements for checking safety requirements specifications for

- Consistency
- Relative Completeness
- Freedom from Ambiguity

should be incorporated into IEC 61508:201x (Edition 3).

As noted above, effective methods for checking consistency and relative completeness of SRSs are dependent upon the use of automated analysis tools, in particular formal languages and their automated checkers. In this regard, formal methods of some sort are essential for these properties.

Freedom from ambiguity is a property necessary for practical consistency and relative-completeness checking, whether by hand or with the help of SW tools. It is most easily assured through the translation, usually by hand, of requirements specifications from natural-language specification into a formal language.

Acknowledgements

The first author wishes to acknowledge recent extensive consultation with Martyn Thomas and Michael Jackson on the matter of what properties are important for safety requirements specifications, as well as, gratefully, the permission to quote substantial portions of copyrighted work. He also would like to thank the many contributors to the York Safety Critical Systems Mailing List and the Bielefeld System Safety Mailing List for discussion over many years; and also recent discussion since 2010 with members and guests of DKE GK 914 and AK 914.0.3.

The second author wishes to acknowledge the support of the German Federal Ministry for Economics and Technology, Bundesministerium für Wirtschaft und Technologie (BMWi), for support under INS Project 1234, negotiated through and administered by the German electrotechnical standardisation organisation DKE.