ISSN 1744-1986

Technical Report N° 2006/11

Relating Safety Requirements and System Design through Problem Oriented Software Engineering

Derek Mannering
Jon Hall
Lucia Rapanotti

14th September 2006

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Relating Safety Requirements and System Design through Problem Oriented Software Engineering

Derek Mannering
General Dynamics UK Limited
UK
derek.mannering@generaldynamics.uk.com

Jon G. Hall Lucia Rapanotti
Centre for Research in Computing
The Open University, UK
{J.G.Hall, L.Rapanotti}@open.ac.uk

Abstract

Standards mandate the demonstration of safety properties for industrial software, starting at the initial requirements phase. The processes involved are iterative, with the choice of potential solution architecture being a driver for the discovery of system failure modes. Managing the resulting development is a complex task.

Problem Oriented Software Engineering brings together many non-formal and formal aspects of software development, providing a structure within which the results of different development activities can be combined and reconciled.

This paper illustrates how problem orientation can support the development task of a safety-critical system through its ability to elaborate, transform and analyse the project requirements, reason about the effect of partially detailed candidate architectures, and traceably audit design rationale through iterative development. The approach is validated through its application to an industrial case study.

1. Introduction

Ensuring adequate safety is a crucial factor in the deployment of many embedded systems, including those used in avionics applications. This concern has been captured in safety standards such as the UK Defence Standard 00-56 [21] and the international IEC 61508 [8]. These standards require hazard identification and preliminary hazard analysis to occur in the early phases of the development process (e.g. [16]). This is consistent with studies that have shown that a large proportion of anomalies occur at the requirements and specification stages of a system development [4], [11]. Further, the anomalies of interest are not restricted to just component or functional failure, but include significant contributions from factors that are emergent properties of the interactions of complex systems [11], [18]. A study by Lutz concluded that safety-related software errors arose most often from inadequate or misunderstood requirements [14]. Other work has highlighted the need to conduct a safety analysis of the requirements [3], [5]. These factors all support the notion that safety must be built into the design, and that the evolving design representations analysed to demonstrate that they have the desired safety properties [13].

The goal of this paper is to demonstrate how the Problem Oriented Software Engineering (POSE) framework [6] can be used to support directly the process of formulating a requirements model that can undergo hazard identification and preliminary hazard analysis – called Preliminary Safety Analysis (PSA) in this paper – as required by the safety standards (e.g. [21]). The result is a revised requirements model that is known to be able to satisfy its identified safety requirements and thus forms a good basis for the remainder of the development process. The approach is validated through its application to a real avionics system. In this work, PSA consists of simple logic proofs to demonstrate systematic correctness [15], Functional Failure Analysis (FFA) [17] to identify safety hazards and issues, and functional Fault Tree Analysis (FTA) [23] to resolve them. These techniques are well defined in their respective references and only the results of applying them are considered in this text.

The paper is organised as follows: background and related work are presented in Section 2. The basics of the POSE framework are described in Section 3. Section 4 demonstrates the use of POSE on a case study involving the development of requirements and high level architecture for a component of an aircraft warning system. Section 5 contains a discussion and conclusions.

2. Background and related work

The work presented in this paper is based on a multi-level safety analysis process typical of many industries. For example, commercial airborne systems are governed by ARP4761 [20]. ARP4761 defines a process incorporating Aircraft FHA (Functional Hazard Analysis), followed by System FHA, followed by PSSA (Preliminary System Safety Assessment, which analyses the proposed architecture). This paper is concerned with the latter, PSSA, but uses PSA in place of PSSA.

The view of requirements in this paper follows the fundamental clarification work of Jackson [24] and Parnas [2] which distinguishes between the given domain properties of the environment and the desired behaviour covered by the requirements. This work also distinguishes between requirements that are presented in terms of the stakeholder(s) and the specification of the solution which is formulated in terms of objects manipulated by software [22]. Therefore there is a large semantic gulf between the system level requirements and the specification of the machine solution. One of the goals of applying POSE is to bridge this gulf by transforming the system level requirements into requirements that apply more directly to the solution.

The POSE notion of problem used in this work fits well with the Parnas 4-Variable model, which has been used by Parnas *et al.* as part of a table driven approach [2]. This model and table-based approach is particularly well suited to defining embedded critical applications. This is demonstrated by the fact that they form the basis for the SCR [1], and the RSML methods. The RSML work led to the SpecTRM [12] methods, which form part of a human centred, safety-driven process which is supported by an artefact called an Intent specification [13]. The work in this document covers much of the second level System Design Principles of the Intent specification, and thus is complementary to the third level Blackbox level provided by SpecTRM.

The work of Anderson, de Lemos, and Saeed [3] share many of the principles and concepts that have driven the development of this work. Particularly the notions that safety is a system attribute and the need to apply a detailed safety analysis to the requirements specifications. The main advantages of the POSE approach over that work are: (a) it provides a framework for transforming requirements; (b) it is rich in traceability; and (c) the models it uses are suitable for the safety analysis. The latter means it is efficient because there is no need to develop "new" models (with all its attendant validation problems) just to perform the PSA. Further, the traceability makes it particularly suited for use with standards such as DS 00-56 [21] and the DO-178B [19] software guidelines.

3. Problem Oriented Software Engineering

Problem Oriented Software Engineering (POSE, [6]) brings together many non-formal and formal aspects of software development, providing a structure within which the results of different development activities can be combined and reconciled. Essentially, the structure is the structure of the progressive solution of a system development problem; it is also the structure of the adequacy argument that must eventually justify the developed system. POSE does not prescribe any particular development process; rather it identifies steps of development which may be accommodated within the development process chosen. Other work has illustrated the solution of mission-critical development problems under POSE [7].

In this paper, we show how POSE fits within a safety-critical development context, such as that defined by DS 00-56 [21]. The process we support is that of formulating a requirements model for a safety-critical system that can undergo hazard identification and preliminary hazard analysis as required by the safety standards. This process is complex and iterative in that design choices affect requirements, and *vice versa*. For this complex process, we show how POSE could be used to: a) provide revised requirements statements together with a design that is known to satisfy them; and b) provide rich traceability and record design rationale throughout the iterative development.

Under POSE, *problems requiring solution*, i.e., requirements in context, are transformed into other problems that are easier to solve, or that will lead to yet other problems that are easier to solve. Problem transformations capture discrete steps in the solution process. The following classes of transformation are recognised in the framework:

Representation: The initial transformation covering the identification of the major component parts of a problem: the given domains (problem context), with their phenomena and behaviours; the machine to be designed and its shared phenomena with the context; the requirement to be satisfied by the machine in its context.

Interpretation: As analysis proceeds, knowledge of the real-world and designed artefacts increases and this will be captured by changes in the respective domain and requirement descriptions. Example of the use of interpretation include: capturing further detail of a domain's description; and using experience and engineering knowledge to select a solution architecture. Requirements may also be interpreted when, for instance, better understanding is reached of a customer's requirements, or in order to separate and address standard concerns, such as safety or security.

Reduction: This allows one to simplify a problem by removing domains from the context, simultaneously changing the requirement to preserve the solution. It is an essential transformation in the framing of subproblems and the derivation of specification statements from requirement statements.

Solution: This provides justification that a solution description is adequate in solving, that is, in its context, it satisfies the problem's requirement.

Each defined problem transformation transforms problems in a way that respects solution adequacy. What this means is given by their general form:

For problems P, P1,...,Pn, with solutions S, S1,..., Sn, respectively, that a problem transformation transforms problem P to the problems Pi, i =1,..., n, with justification J, means that under the transformation, S is a solution of P with adequacy argument

(A1 & A2 & ... & An) & J whenever S1,..., Sn are solutions of P1,..., Pn, with adequacy arguments A1,..., An, respectively.

The justification *J* for the transformation will not, in general, be formal and so the transformations of the framework need not be sound in any formal sense: the informality of the subject matter precludes fully formal treatment of some transformations. Illustrations of the forms of justification we admit are given later in the paper, however, as an example, one justification we use is that of an engineer deeming an extant component to be appropriate as part of the solution; the ramifications of this being incorrect are discussed in Section 4.

Problems are transformed from more complex to less complex and, in general, it cannot be known during transformation application whether a future solution will satisfies a requirement until that solution is actually built. All the transformation says is that, if solutions to the simpler problems can be found then so can a solution to the more complex problem. During development, then, often the best we can do is to ensure that future design choices, such as that of a high-level architecture, do not make it impossible to satisfy the requirements, and that this be recorded in the justification. As a simple example of what we mean, consider that one should not choose a processor that is known to be very unreliable as the basis of a safety-critical system as such a processor is unlikely to meet safety-critical requirements; the best that can be done is to choose an ultra-reliable processor noting, however, that even this choice does not guarantee that the requirements will be met, as there are many other factors that could impact the satisfaction of the safety requirements.

For definiteness and precision, the original POSE framework [6] is presented as a Gentzen-style sequent calculus [10]. In this paper, as in others, we use the graphical notation of the closely related Problem Frames approach (PF, [9]) for illustration.

Both POSE and PF regard a problem as requirements in a real-world context. A context is a set of (possibly) interacting domains described in terms of their indicative properties; each domain description captures a part of the real-world which is of interest in the problem; a requirement is a statement – written in the optative mood – of what should be true (or what should turn out to be true) of the context given an operating solution to the problem. A solution is simply the description of a domain, representing a machine whose behaviour is constrained by a developed program, that solves the problem. Thus, a software problem challenges us to find the solution that, in the given context, brings about the requirement.

A thorough treatment of PF is beyond the scope of this paper, but can be found in [9]. Here we give a short overview of problem diagrams - the notation used by PF to represent problems - through the example in Figure 1. The problem is to specify a solution machine called *FAS* (represented as a double-barred box), which interacts with real-world domains *Speaker*, *Catastrophic System*, *System 1*, etc. (represented as boxes) in such a way that the requirement R (represented as a dotted oval) is satisfied. Links between solution machine and real-world domains capture relevant shared phenomena (e.g., entities, values, events, commands or operations). For instance, in Figure 1, *FAS* shares phenomenon `message' with *Speaker*; that such a phenomenon is controlled by *FAS* is indicated by the '!'. The requirement is linked to domains whose properties and phenomena are referred to or constrained by the requirement. For instance, the dotted line between R and *Catastrophic System* indicates that R refers to phenomenon *cat*, while the dotted arrow between R and *Pilot* indicates that R constrains phenomenon *audio*. Appropriate descriptions of its solution, domains, requirement and phenomena are associated with a problem diagram in the course of analysis, as we will see in the case study of the next section.

4. Case study

The case study concerns the design of a Failure Annunciation System (*FAS*) that is part of a military aircraft. The *FAS* is a component of an actual operational System Failure Warning System (*SFWS*) flying today. The *SFWS* contains a number of other communication functions but these do not impact the function being considered, and hence have been omitted from the study for reasons of brevity. In this sense, the case study is simplified but much of the interesting complexity of the system remains.

Here we have fitted POSE within a traditional safety-critical system development process, that of DS 00-56 [21]. The case study assumes that an aircraft level safety analysis has been completed and that this has allocated safety requirements to the main aircraft systems, of which the *SFWS* is one. It also assumes that a system safety analysis of the *SFWS* has been completed and that this has allocated requirements to its sub-systems, including the *FAS*. In this way the safety requirements for the *FAS* (H1 & H2, see Section 4.1) are defined and allocated to it. Below we give a POSE characterisation of the problem and transform it, using POSE, to a design suitable for carrying out a PSA. As the PSA highlights problems with the design, we then show how to rethink design, with POSE providing backtracking and traceability.

4.1 Overview of the system

In a real aircraft, the *SFWS* monitors a number of the aircraft's sub-systems, warning the pilot if one of the monitored systems has failed. The *SFWS* comprises three diverse warning systems: (a) the Failure Annunciation System (*FAS*) which provides audio warnings through a speaker; (b) a visual warning system that drives a dedicated warning display panel in the cockpit; and (c) a visual warning system that sends warning messages to the pilot's main display.

Typically, the monitored systems include: inertial navigation; GPS navigation; aircraft data; environment data; and health monitoring. Failure of these systems will, typically, not prove insurmountable by the pilot. In contrast, a failure in the *Catastrophic System* (part of the flight control system) could result in the loss of the aircraft and/or pilot.

The SFWS safety requirement allocated by the aircraft level system safety analysis identifies the following *SFWS* failures as hazards to be considered:

H1:	Inadvertent indication of the Catastrophic message.
H2:	Failure to indicate the Catastrophic message.

These two hazards do not have the same nature: hazard H1 is particularly problematic because of the action the pilot has to take if the *Cat fail* message is played. The inadvertent indication of *Cat fail* is regarded as an event to be avoided. As a result, it is classified as *safety critical*, and assigned a target failure probability of 10^{-7} fpfh (failures per flight hour).

In contrast, hazard H2, the failure of one of the *SFWS* systems to indicate a *Catastrophic System* failure, is mitigated by the remaining two *SFWS* systems, which are intentionally diverse in operation. Hence H2 is classified only as *safety related*, and assigned a target failure probability of 10^{-5} fpfh.

In the following, we focus on the development of the *FAS*. In particular, we show how POSE applies to formulate the specification of the embedded computer control system that controls the selection and generation of audio warning messages. Moreover, we demonstrate how POSE assists in the development of an architectural model that is capable of satisfying its H1 and H2 safety requirements.

A first representation of the *FAS* problem is given by problem $P_{Initial}$ in Figure 1, expressed as a problem diagram. In the figure, the *FAS* can be seen to monitor directly the status of the *Catastrophic System* using a discrete input (*cat*). It also monitors the status of other three systems, *Systems 1*, *2* and *3*, representing the various sub-systems of the aircraft, by interrogating their health status messages (*sys1*, *sys2* and *sys3*, respectively). The *FAS* issues warning audio messages to the pilot via a speaker.

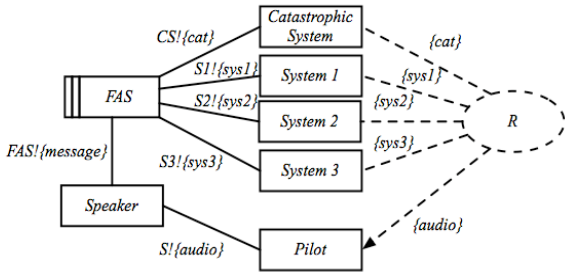

Figure 1 The FAS problem (problem $P_{Initial}$)

The functional aspects of the requirement R for the *FAS* are:

Ra:	When health monitoring indicates that a monitored system has failed, the system should play the correct audio message to pilot.
Rb:	The message levels should be comfortably heard by the pilot.
Rc:	If more than one system has failed, then messages should be selected for play in the order: *Cat fail*, *Sys1 fail*, *Sys2 fail* and *Sys3 fail*.
Rd:	If no system failures are detected, then no message should be played.

As well as Ra, Rb, Rc and Rd, *FAS* should also satisfy the safety targets set by the aircraft system level safety analysis. Recognising this we add safety requirement RS to R:

RS:	For hazards H1 and H2, their respective safety targets (10^{-7}fpfh and 10^{-5}fpfh) must be satisfied.

The overall requirement R is Ra & Rb & Rc & Rd & RS is indicated in the dotted ellipse in Figure 1. A complete statement of R should also include requirements that cover space, weight, interfaces, maintenance and so on, but these are beyond the scope of this work.

4.2 A FAS candidate architecture

Safety-critical developments are subject to many of the same constraints as other developments, with system cost being an important consideration. Because of this, we will make use of off-the-shelf components for Failure Detection (the *FD* component), for Audio Output Selection (the *AO Selector* component) and for Audio Output Decoding (the *AO Decoding* component), combining them together with a (still to be designed) Failure Annunciator Controller (*FA*).

The *FD* is shown in Figure 2. It receives health status information about *Systems 1*, *2* and *3,* which it decodes and sends to the *FA* via its *Status* signal. In addition, it monitors the *Catastrophic System*'s discrete input *cat* which it also sends to the *FA* via its *Status* signal. The *Control Decoder* prioritises the failure data to send to the *FA*.

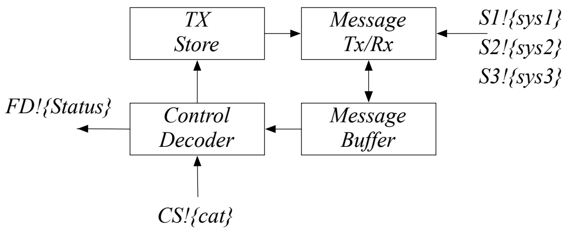

Figure 2 The Failure Detection component *FD*

The combination of the *AO Selector* and *AO Decoder* is shown in Figure 3. Their combined role is to output the audio signal of the message selected by the

FA. The *AO Selector* is an FPGA-based device, containing a library of digital audio messages stored in PROM. The *AO Decoder* decodes the selected digital message and turns it into an audio wave for the speaker.

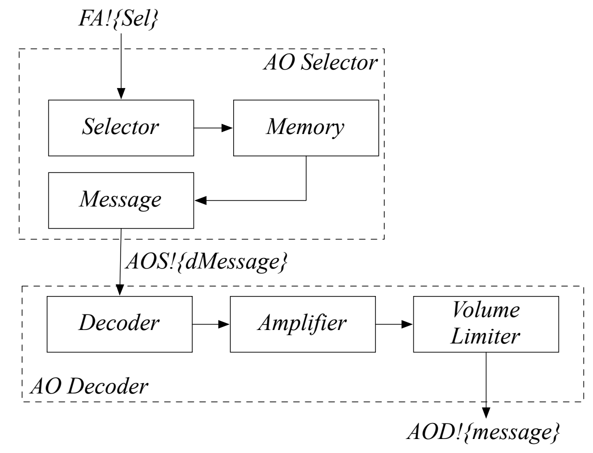

Figure 3 The Audio Output components

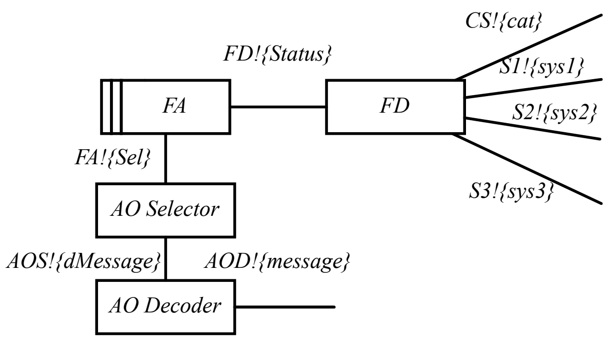

Figure 4 The candidate architecture for *FAS*

Comprising four components, a candidate architecture for *FAS* is shown in Figure 4 and, given the initial representation of the problem ***P**Initial*, we may introduce this candidate architecture (using the POSE transformation *Solution Interpretation*) the result being problem ***P**Interpreted* shown in Figure 5.

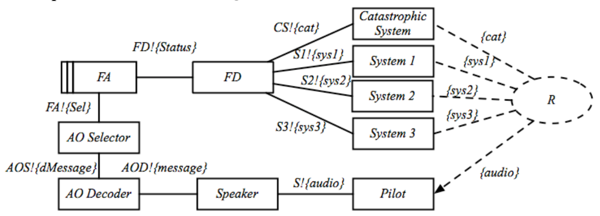

Figure 5 Solution Interpretation of *FAS* (problem *PInterpreted*)**

From the figure, we see that the to be designed component *FA* should take in the failure status information from the *FD*, process it, then send out the appropriate audio message request to the *AO Selector*. If no failures are reported by the *FD*, then the *FA* sends no message request to the *AO Selector*, otherwise the *FA* sends out the highest priority message request.

The justification J_1 that this step is adequate is simply engineering judgement: that the candidate architecture is a suitable starting point for the design. (Both problem and justifications are named so that we can easily refer to them later.)

4.3 Simplifying the problem

With the introduction of the candidate architecture, a PSA is required to see whether the architecture can safely be the basis of the *FAS*. However, the problem has become quite complex, and we want to simplify it as much as possible to remove any factors that will unnecessarily complicate the PSA.

Under POSE, problem simplification is achieved through *Problem Reduction* which allows domains to be removed from the context whilst simultaneously rewriting the requirement to compensate for their removal. For reasons of brevity, only an outline of the transformation can be given here; a thorough presentation can be found in [6], [7].

Generally, domains furthest from the machine are removed first: in this case the first domain to be removed will be the *Pilot*. This reduction consists of two transformation steps. Firstly, the requirements relating to the *Pilot* are transformed to relate to the *Speaker* domain. For example, Ra contains the term "Play the correct audio message to the pilot". Under the transformation, this will be rephrased to refer to the *Speaker* by making (and justifying) transformations from *Pilot*-oriented phenomena into *Speaker*-oriented phenomena; similarly for Rb (see below). Secondly, to complete the domain removal of the *Pilot*, we record assumptions that need to be in place for consistency. In this case, this will include the assumption that the *Pilot* can actually hear the message as it issues from the *Speaker*. That we rely on this assumption holding is the justification J_2 for this reduction. Transforming R in this way yields a new requirement statement, that we will call R1, in which Ra becomes R1a and Rb becomes R1b (shown below). Rc and Rd are not changed by the transformation (they do not mention *Pilot* phenomena) and become R1c and R1d unchanged, respectively. RS remains unchanged too. The resulting problem, with requirement R1 = R1a & R1b & R1c & R1d & RS, is shown in Figure 6.

The revised R1a and R1b are:

R1a:	When health monitoring indicates that a monitored system has failed, play the correct audio message through the *Speaker*.
R1b:	The message levels should be at 60dB above the ambient cockpit level.

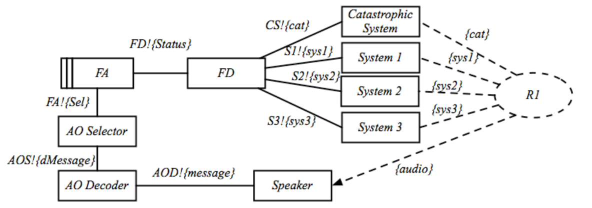

Figure 6 Pilot domain removal (problem P_{Red1})

There are a number of domain removals that follow, which, for brevity, we do not illustrate with figures. The next domain removed is the *Speaker*. As before this involves two steps. In this case R1 is transformed into R2, and further assumptions are added to cover the removal of the *Speaker* domain. This time R1a and R1b that refer to the *Speaker* have to be re-phrased in terms of the *AO Decoder* audio output stream that drives the *Speaker*, to become R2a and R2b:

R2a: When health monitoring indicates that a monitored system has failed, the *AO Decoder* should generates a sequence of audio signals that corresponds to the correct system fail message.

R2b: The audio sequence amplitude levels are within the defined range.

As before R1c becomes R2c, and R1d becomes R2d without change, and RS is unchanged too. The domain removal assumption is that the audio output stream provided by the *AO Decoder* correctly drives the *Speaker* within the specified comfortable audio range, to produce the desired warning message (justification J_{30}).

There is still a wide gulf between the phenomena that *FA* understands (to do with message selection) and the requirement R2a, which talks about audio sequences. Further reduction is required.

Our next goal is to remove the *AO Decoder* domain. To do so, we need to consider that it consists of a volume limiter, an amplifier and the digital to analogue decoding block (*Decoder*), as shown in Figure 3.

Removing the *Volume Limiter* does not affect R2a, R2c or R2d, these becoming R3a, R3c and R3d directly. However, it requires R2b to be modified to remove the upper limit. The latter becomes one of the assumptions of the domain removal, i.e., removing *Volume Limiter* adds the assumption that the analogue drive to the *Speaker* will be limited to be within a maximum sound output, with the result that the audio will not be uncomfortably loud (justification J_{31}).

R3b: The audio sequence amplitude levels are within comfortable levels.

Removing *Amplifier* does not affect R3a, R3c or R3d which become R4a, R4c and R4d, respectively. However, R3b is, effectively, removed, becoming an assumption that there will be a defined audio signal gain from input to output, i.e., the audio will be loud enough to be comfortably heard, (justification J_{32}). Other assumptions concerned with signal quality (e.g., distortion removal) might also be added at this point.

Removing *Decoder* does not affect the requirement R4c, which becomes R5c. However, the *Decoder* block transfers byte streams into analogue audio signals; therefore the requirements R4a and R4d and the transfer function through *Decoder* are key elements of the transformation. The removal of *Decoder* requires R4a to become R5a and R4d to become R5d as follows:

R5a: When health monitoring indicates that a monitored system has failed, then *AO Selector* should generate a sequence of bytes that corresponds to the selected system fail message.

R5d: If no system failure is detected then the *AO Selector* block should generate no message.

The assumption is that this sequence of bytes, when appropriately decoded, will produce the desired audio signal, i.e., the correctness of the domain transfer function for *Decoder* becomes an assumption that is added to the requirements when the *Decoder* domain is removed (justification J_{33}).

The requirement R5 is R5a & R5c & R5d & RS together with the set of assumptions associated with the domains removal. At this point, R5 talks about "selected system fail message" which corresponds directly to the phenomena shared between *FA* and *AO Selector*. Also, inspection of the *FD* phenomena and their comparison with the R5 monitoring terminology indicates that they are equivalent. That is, R5 talks about "when health monitoring indicates that that monitored system has failed" and this corresponds directly to the *Cat* and message status data that is sent via *Status*. Hence, removing *Catastrophic System,* and *Systems 1*, *2* and *3* leaves R5 unchanged, with the resulting problem, $P_{Reduced}$, being that shown in Figure 7. (This rationale becomes J_{34}, the justification for the removal of the monitored systems: there is no need to rewrite R5.)

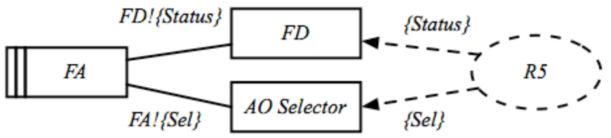

Figure 7 The reduced FA problem (problem $P_{Reduced}$)

4.5 Formalising the requirements

It is a simple step to formalise the requirements for input to the PSA. The corresponding POSE transformation is *Requirements Interpretation* which, with adequate justification, allows a requirement to be rewritten in an equivalent form. The non-safety aspects of the requirement can be formalised into Parnas Table-like form, with the resulting requirement shown in Table 1.

Table 1 Formalised R5, prior to PSA

Monitor Condition	Constraint
CAT = on	*Sel = Cat fail*
CAT ≠ on ∧ Sys1fail	*Sel = Sys1 fail*
CAT ≠ on ∧ Sys2fail	*Sel = Sys2 fail*
CAT ≠ on ∧ Sys3fail	*Sel = Sys3 fail*
CAT ≠ on ∧ Sys4fail	*Sel = null*
RS: H1 (10^{-7}), H2 (10^{-5}) safety targets must be satisfied	

4.6 Preliminary Safety Analysis

The solution preserving nature of problem transformation under POSE leaves us at a point at which the solution of the reduced *FA* problem will also be a solution of the initial problem. Note that this does not mean that there is, necessarily, a solution to the reduced *FA* problem. In fact it turns out from a PSA that the chosen candidate architecture is *not* a suitable basis for a solution, and it will have to be reworked: the process is necessarily iterative.

In general, the goal of a PSA is to: (a) confirm any relevant hazards from the system level hazard list; (b) identify if further hazards need to be added to the list; and (c) then analyse an architecture to validate that it can satisfy the safety targets associated with the identified relevant hazards. In this case, within the POSE, the PSA goal is also to determine whether a solution to the reduced *FA* problem exists. A number of techniques can be applied to perform a PSA. This work uses a combination of mathematical proof, Functional Failure Analysis (FFA) [17] and functional Fault Tree Analysis (FTA) [23].

The POSE framework structuring and the phased development means that relatively simple formal requirements (as in Table 1) can be developed that apply directly to the solution machine. Simple logic proofs demonstrate that R5 (Table 1) has the required functional properties. Therefore, the remaining feasibility check at this level is to demonstrate that the hardware reliability of the design blocks, i.e., those of Figures 2 and 3, can satisfy the safety requirement RS.

The FFA can be used to identify any additional relevant hazards or, more likely, it will identify credible failure modes that result in an existing hazard. The FFA should be applied to each architectural component in turn. Functional FTA can then be used to analyse if the events identified by the FFA satisfy the targets contained in RS.

There is insufficient space to present the full PSA, hence we summarise only the main elements of the *AO Selector* analysis to demonstrate the process followed. The significant results from applying FFA to the *AO Selector* are shown in Table 2.

Table 2 AO Selector FFA Results

Id	Failure Mode	Haz
F1	Plays no messages – no *Cat fail* when required.	H2
F2	Plays *Cat fail* message too late.	H2
F3	Plays wrong message – inadvertent *Cat fail*.	H1
F4	Plays wrong message – no *Cat fail* when required.	H2
F5	Plays too loud – *Pilot* switches system off.	H2
F6	Plays too softly – *Cat fail* not heard.	H2

FTA, applied to the *AO Selector* architecture (refer back to Figure 3), with F1 to F6 as the top events, is used to establish if the architecture can satisfy its targets. F3 is dominated by the known failure rate of the FPGA that would implement the *AO Selector* functionality. This indicates that the failure rate for F3 is $3x10^{-7}$ fpfh which, therefore, does not satisfy the target for H1. The result is that the FPGA *AO Selector* component will need to be rethought in order to meet all safety requirements.

The POSE development so far is captured in Figure 8 as a tree (without branching). The tree is notable in that, from the PSA, ***P**$_{Reduced}$* has no solution, and so there is no further development to be done. Therefore, we must backtrack the development to the point at which the architecture was introduced, and continue afresh. How this is done is shown next.

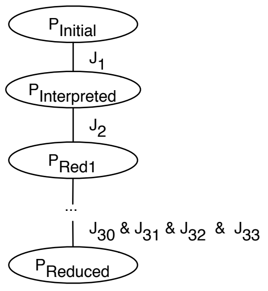

Figure 8. The development prior to the PSA

4.7 Rethinking the architecture

To address its shortcoming, a modified FPGA *AO Selector* component is developed to: (a) include sending back the message identifier of the currently playing message to the *FA* in a status message; and (b) add a mute input to the *FA* control that allows the *FA* to mute audio output if the message identifier does not tally with the required message to be played. (Although the design of the modified *AO Selector* component could be done within POSE, for reasons of brevity, we have not done this.)

A derived requirement DR is also introduced to the effect that, if the message identifier is not the same as the currently playing message, then audio must be inhibited, audio otherwise being allowed.

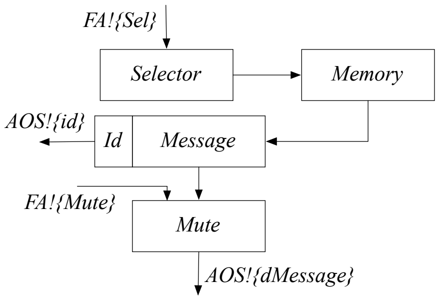

Figure 9. Modified AO Selector

The changes to the *AO Selector* are shown in Figure 9. The POSE development must also backtrack to allow the reintroduction of the new architecture. To do this the tree of Figure 8 is pruned back to the node labelled $P_{Initial}$ at which point the new architecture is introduced resulting in Figure 10.

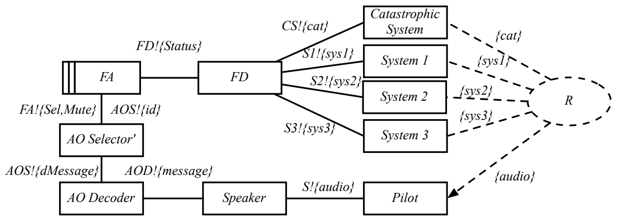

Figure 10. Introducing the second candidate architecture for AO Selector' (problem $P'_{Interpreted}$)

The justification (J'_1) for the introduction of the new architecture is notable in that it should justify the choice of the new architecture over the old. One way to do this is to include the pruned part of the tree in the justification, adding a note to the effect that this was the reason for the choice of new architecture.

Note that DR also introduces additional failure modes. For example, mute failing on is a form of F6 (see Table 2). It should also be noted that F2 can be considered to be a form of F1 in the limit. Table 3 shows the results of repeating the FFA, collating results and then performing the FTA with the new architecture. The Target – shown in brackets after the FTA calculation result – is the most severe probability applicable (e.g., the H1 target, 10^{-7} fpfh, used for first two terms).

Table 3. Collated PSA results for AO

Collated Failure Mode	FTA	Haz.
Wrong message played, correct id	10^{-12} (10^{-7})	H1&H2
Wrong messaged played, expected id	10^{-13} (10^{-7})	H1&H2
Correct message, wrong or no id.	10^{-7} (10^{-5})	H2
No message played: mute fails/too soft	10^{-6} (10^{-5})	H2
Message played too loud	10^{-6} (10^{-5})	H2

The worst effect of "Message played too loud" is considered to be that the pilot switches the *FAS* off to avoid the distraction – this equates to hazard H2, so the H2 target is used. Mitigation for this failure mode is provided by the *Volume Limiter* circuitry which is part of the *AO Decoder* (see Figure 3). Inspection of the analysis results in Table 3 indicates that the modified *AO Selector'* architecture is adequate for this aspect of the safety targets, and therefore it is valid to use it to continue with the development.

A similar analysis is applied to the *FD* domain and satisfactory results are obtained. Therefore, the modified *FAS* architecture can be argued not to prevent satisfaction of both the functional and safety target (RS) requirements, and hence is a suitable basis for the remainder of the development process, that is, designing *FA*.

The problem development forward from this point is similar to that previously: problem reduction is applied to arrive at the reduced *FA* problem of Figure 11.

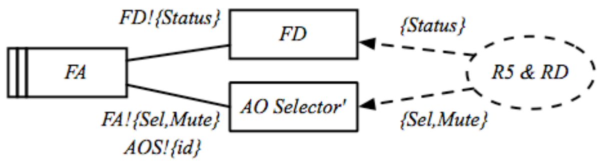

Figure 11. New reduced *FA* problem (problem $P'_{Reduced}$)

4.8 Towards a specification for *FA*

In this section we show how R5 and DR are combined leading to a requirement R6 which is the basis for specifying *FA*.

Effectively there are two machines to define and then combine. There is the *FA* machine that drives a correctly operating *AO Selector*, which is defined by R5 (Table 1), and there is the *FA* machine that has to safely drive a failed *AO Selector'*, whose behaviour is defined by DR. That is, under no-failure conditions the Table 1 behaviour should be followed. However, when the potentially catastrophic failure is detected, then the *FA*'s behaviour has to be modified in line with DR, and audio has to be muted. That is, the behaviour in the presence of failure dominates the normal operating behaviour so as to ensure safety.

With the interpretation of the requirements that DR has priority over R5, a new statement of the requirement for this problem is shown in Table 4, in which the behaviour required by DR effectively overrides aspects of the behaviour required by R5 in order to ensure safe operation.

Table 4. FA Requirement R6, after PSA

Monitor Condition	Constraint
CAT = on ∧ id = cat	Sel = Cat ∧ mute = off
CAT = on ∧ id ≠ cat	Sel = null ∧ mute = on

CAT ≠ on ∧ Sys(1) ∧ id = S1f	Sel = S1f ∧ mute = off
CAT ≠ on ∧ Sys(1) ∧ id ≠ S1f	Sel = null ∧ mute = on
CAT ≠ on ∧ Sys(2) ∧ id = S2f	Sel = S2f ∧ mute = off
CAT ≠ on ∧ Sys(2) ∧ id ≠ S2f	Sel = null ∧ mute = on
CAT ≠ on ∧ Sys(3) ∧ id = S3f	Sel = S3f ∧ mute = off
CAT ≠ on ∧ Sys(3) ∧ id ≠ S3f	Sel = null ∧ mute = on
CAT ≠ on ∧ Sys(0) ∧ id = nul	Sel = null ∧ mute = off
CAT ≠ on ∧ Sys(0) ∧ id ≠ null	Sel = null ∧ mute = on
RS: H1 (10^{-7}), H2 (10^{-5}) safety targets must be satisfied	

Applying simple logic proofs to Table 4 indicates that it does correctly capture the required safer behaviour. Therefore, the given choice of architecture and the above statement of R6 form a suitable basis for further development. In particular, they can be used to derive a specification for *FA, (*justification *J_4*), leaving problem ***P'**$_{Spec}$* (which is ***P'**$_{Reduced}$*, with R6 replacing R5 & RS).

5. Discussion and conclusions

In our view, software engineering includes the identification and clarification of system requirements, the understanding and structuring of the problem world, the structuring and specification of a hardware/software machine that can ensure satisfaction of the requirements in the problem world, and the construction of arguments, convincing both to developers, customers, users and other stake-holders that the system will provide the functionality and qualities that are needed.

Software development is, thus, a complex, iterative process made more difficult by the need to relate human and physical domains to the formal world of the machine. In some areas, such as safety-critical systems, paramount importance is placed on the quality of the software, with this driving the development process forwards, and backwards, as needed. An effective approach to system development must therefore deal adequately with the informal, the formal, and the relationships between them.

In this paper, we have illustrated how a problem oriented approach to software engineering (POSE) can mesh within and support more traditional processes, to provide structuring for the development process, and rich traceability and auditability during iteration. The analysis of Section 4 demonstrates how the artefacts produced under POSE can be used to: (a) evolve the system level requirements such that they are directly related to a software intensive embedded machine; (b) form the basis of safety analysis work; and (c) combine the functional and derived safety requirements into a coherent, safe requirements model that forms a good basis for the remainder of the development.

Moreover, the work of Section 4 illustrates how POSE can be used to record the structure of the development and of the argument of the adequacy of the proposed solution, including the rationale for important design choices (why the simple initial candidate architecture was not suitable as the basis of the design) properly situated within the documentation of the development. Indeed, under POSE, each step in the requirements transformation process can be audited, the rationale for making a particular decision validated, and the traceability of the process demonstrated. These features are extremely important in safety critical system applications where the case for safety has to be justified, and independent auditing is the accepted means of validating this justification [21], [8]. This suggests that POSE may be a suitable front-end of an integrated safety-critical development approach suitable for embedded avionics applications.

The completed development structure is shown in Figure 12, note that *J'_1* contains a portion of the candidate architecture development to justify the change of architecture.

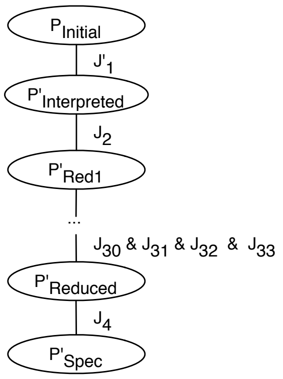

Figure 12. The complete POSE development structure and associated adequacy argument steps for *FA*

Currently, within the safety-critical development industry, there is no systematic procedure for arriving at quality requirements, i.e., requirements suitable to begin a formalised development process. Because the process is currently *ad hoc,* too many poor requirements get through (i.e., those that are ambiguous and/or difficult to verify and validate). Being able to provide some consistency and traceability, as we have shown possible under POSE, especially if we can also link to other known successful notations will greatly improve matters.

6. Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Grants, and the EPSRC, Grant number EP/C007719/1. Thanks also go to our colleagues in the Centre for Research in Computing at The Open University, particularly Michael Jackson and Bashar Nuseibeh.

7. References

[1] R. Bharadwaj and C. Heitmeyer, "Developing high assurance avionics systems with the SCR requirements method," 19th Digital Avionics Systems Conferences, 2000.

[2] P.-J. Courtois and D. L. Parnas, "Documentation for Safety Critical Software," 15th International Conference on Software Engineering, Baltimore, USA, 1997.

[3] R. de Lemos, A. Saeed and T. Anderson, "On the Integration of Requirements Analysis and Safety Analysis for Safety-Critical Systems," University of Newcastle upon Tyne, UK http://citeseer.ist.psu.edu/536230.html, 1998.

[4] A. Ellis, "Achieving Safety in Complex Control Systems," Safety Critical Systems Symposium, Brighton, United Kingdom, 1995.

[5] A. Gerstinger, G. Schedl and W. Winkelbauer, "Safety versus Reliability: Different or Equal," 20th International System Safety Conference, Denver, Colorado, USA, 2002.

[6] J. G. Hall and L. Rapanotti, "A framework for software problem analysis," Open University TR2006/10, 2006.

[7] L. Rapanotti, J. G. Hall and M. Jackson, "Problem transformations in solving the Package Router control problem," Open University, TR2006/07, 2006.

[8] IEC, "61508 Functional safety of electrical/electronic/ programmable electronic safety-related systems," International Electrotechnical Commission.

[9] M. A. Jackson, *Problem frames : analysing and structuring software development problems*. Harlow: Addison-Wesley, 2001.

[10] S. C. Kleene, *Introduction to Metamathematics*: Van Nostrand, Princeton, NJ., 1964.

[11] N. Leveson, *Safeware: system safety and computers*. Addison-Wesley, 1995.

[12] N. G. Leveson, "Completeness in formal specification language design for process-control systems," Proceedings of the third workshop on Formal methods in software practice ACM Press, 2000.

[13] N. G. Leveson, "Intent Specifications: An Approach to Building Human-Centered Specifications.," IEEE Transactions on Software Engineering, vol. 26, pp. 15-35, 2000.

[14] R. R. Lutz, "Analysing Software Requirements Errors in Safety-Critical Embedded Systems," IEEE International Symposium Requirements Engineering, 1993.

[15] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*: Addison Wesley, 1985.

[16] P. A. Martino and C. Muniak, "The Role of System Safety Engineering in Product Safety," 20th International System Safety Conference, USA, 2002.

[17] J. McDermid and T. Kelly, "Safety and Hazard Analysis Course," in High Integrity Systems Group, Department of Computer Science: York, 1999.

[18] E. N. Overton, "System Safety Analysis of Safety Critical Software and the Human User," 19th International System Safety Conference, Huntsville, Alabama, USA, 2001.

[19] RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," December 1 1992.

[20] SAE, "ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment," December 1996.

[21] UK-MoD, "Safety Management Requirements for Defence Systems Part 1 Requirements," MoD, Interim Defence Standard 00-56 Issue 3, 17 December 2004.

[22] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective," ICSE'00, 22nd International Conference on Software Engineering, Limerick, 2000.

[23] W. Vesely, F. Goldberg, N. Roberts and D. Haasl, *Fault Tree Handbook*, vol. NUREG-0492: U.S. Nuclear Regulatory Commission, 1981.

[24] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," ACM Transactions on Software Engineering and Methodology, vol. 6(1), pp. 1-30, 1997.