ISSN 1744-1986

The Open University

Technical Report N° 2006/14

A problem-oriented approach to normal design for safety-critical systems

Derek Mannering
Jon G Hall
Lucia Rapanotti

19th October 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

A problem-oriented approach to normal design for safety-critical systems

Derek Mannering[1], Jon G. Hall[2], and Lucia Rapanotti[2]

[1] General Dynamics United Kingdom Limited
[2] Department of Computing, The Open University, UK
{derek.mannering@generaldynamics.uk.com;
J.G.Hall@open.ac.uk ; l.rapanotti@open.ac.uk.}

Abstract Normal design is, essentially, when an engineer knows that the design they are working on will work. Routine 'traditional' engineering works through normal design. Software engineering has more often been assessed as being closer to radical design, i.e., repeated innovation. One of the aims of the *Problem Oriented Software Engineering framework* (POSE) is to provide a foundation for software engineering to be considered an application of normal design. To achieve this software engineering must mesh with traditional, normal forms of engineering, such as aeronautical engineering. The POSE approach for normalising software development, from early requirements through to code (and beyond), is to provide a structure within which the results of different development activities can be recorded, combined and reconciled. The approach elaborates, transforms and analyses the project requirements, reasons about the effect of (partially detailed) candidate architectures, and traceably audits design rationale through iterative development, to produce a justified (where warranted) fit-for-purpose solution. In this paper we show how POSE supports the development task of a safety-critical system. A normal 'pattern of development' for software safety under POSE is proposed and validated through its application to an industrial case study

Keywords: problem orientation, normal design, safety analysis, traceability.

1 Introduction

Vincenti, in [23], defines 'normal design' as that the engineer is engaged in when s/he knows from the outset "how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task". Much of the routine design encountered in traditional engineering disciplines works 'normally'. Many have observed that much of software engineering does not: Maibaum [14] states that "SE ignores the principles of engineering design and almost always adopts radical design methods, to its detriment;" Jackson [9] states that "Though less conspicuous than radical design, normal design makes up by far the bulk of day-to-day engineering enterprise. Unfortunately, this is not true of software engineering."

Through regulation, standardisation and its co-location with traditional engineering disciplines, safety-critical software intensive systems engineering may be

closer to being a normal design area of software engineering. There is much to recommend it: typically, industrial standards and practices require integration with other engineering processes. They require hazard identification and preliminary hazard analysis to occur in the early phases of the development process (e.g. [16]), which is consistent with studies that have shown that safety-related software errors arose most often from inadequate or misunderstood requirements [13]. Other work has also highlighted the need to conduct a safety analysis of the requirements [4, 5]. These factors all support the notion that safety must be built into the design, and that the evolving design representations are analysed to demonstrate that they have the desired safety properties [12].

Normal design processes are distinguished by Vincenti through three characteristics [23], they (a) rely on engineering judgement in the searching of past experience; (b) allow the conceptual incorporation of the novel features that come to mind in solving a problem; and (c) allow the mental winnowing of the conceived variations [23] to pick out those most likely to work.

In a previous case study, we captured a record of a current industrial safety-critical software intensive design process in the Problem Oriented Software Engineering (POSE) framework of [7]. Working from this record, the goal of this paper is to validate it, through its normal use in the design of a different, functionally unrelated avionics system component. We find that the process is a good fit for the needs of safety-critical development, has Vincenti's three characteristics, and consider this to be validation that to some limited extent POSE offers an approach to capturing normal design in software engineering.

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 defensive aids system. Section 5 contains a discussion and conclusions.

2 Background and Related

The case study 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 [19]. 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 (Preliminary Safety Analysis) in place of PSSA. In this paper requirements follow the fundamental clarification work of Jackson [24] and Parnas [3] 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 stake-holder(s) and the specification of the solution which is formulated in terms of objects manipulated by software [21]. 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. This has been used by Parnas et al. as part of a table driven approach [3], which 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 [11] methods, which form part of a human centred, safety-driven process which is supported by an artefact called an Intent specification [12]. 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 [4] 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 [20] and the DO-178B [18] software guidelines.

3 Problem Oriented Software Engineering

Problem Oriented Software Engineering (POSE, [7]) brings together many non-formal and formal aspects of software development, providing a structure within which the results of different development activities can be captured, 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 papers have illustrated the solution of safety- and mission-critical development problems under POSE [6, 8].

3.1 Problems, solutions and transformations

POSE is a Gentzen-style sequent calculus for problems, in which problems requiring solution, i.e., requirements in a real-world context, are transformed into other problems that are easier to solve, or that will lead to other problems that are easier to solve. Problem transformations capture (atomic) discrete steps in development [7]. The following classes of transformation are recognised in the framework:

Representation: The initial problem capture covering the identification of the major component parts of a problem: the problem context; the solution to be designed; and the requirement to be satisfied by the solution in its context.

Interpretation: by which increasing knowledge of the real-world and designed

artefacts is captured within the problem definition.

Expansion: when structures identified as useful are allowed to influence the structure of the problem. For instance, when solution interpretation introduces an architecture made of a number of distinct components, expansion allows the derivation of sub-problems that identifies the individual problems they solve.

Reduction: by which a problem is simplified through the removal of domains from the problem's context, simultaneously changing the requirement to preserve the solution.

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.

The definition of POSE in [7] is formal. In this paper, as in others, we use the graphical notation of the Problem Frames approach (PF, [10]) for illustration. POSE generalises PF, and they agree on the meaning of a problem. A thorough treatment of PF is beyond the scope of this paper, but can be found in [10]. Here we give an overview of problem diagrams - the notation used by PF to represent problems - through the example in Fig.2 (on page 6).

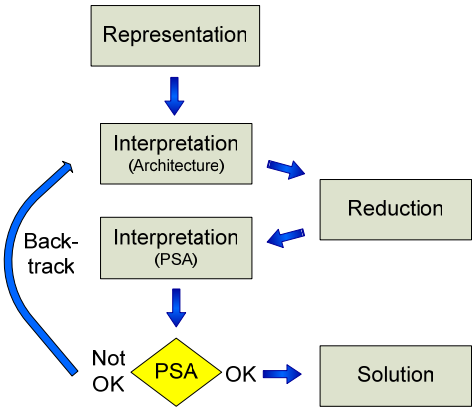

Fig. 1 POSE Safety Requirements Transformation "Pattern"

The problem there is to specify a solution machine called *DC* (represented as a double-barred box), which interacts with real-world domains *Dispenser Unit*, *Pilot*, 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, *DC* shares *ok* with *Pilot*; that such a phenomenon is

controlled by *Pilot* is indicated by the 'P!' on the arc annotation. The requirement is linked to domains whose properties and phenomena are referred to or constrained by the requirement. For instance, the annotated dotted line between R and *Pilot* indicates that R refers to phenomenon *ok*, while the annotated dotted arrow between R and *Dispenser Unit* indicates that R constrains phenomena *fire* and *sel*. 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.

3.2 Problem-oriented approach to safety analysis

POSE allows that previous developments can be captured for re-use. A previous paper [6] used POSE to design a safety-critical component of an aircraft. In this paper, we show how the captured development process provides an approach that can be successfully applied in another a safety-critical development. It is shown how the POSE transformations can be combined to form a re-usable template or "pattern" [2] for identifying, analysing and assimilating safety requirements into a functional requirements framework. The "pattern" emerged from [6], which used the POSE transformations in the sequence shown in Fig. 1. The pattern is iterative between problem and solution spaces with the engineer backtracking and refining the architecture interpretation until a possibly successful solution – as determined by the PSA – is established. The process we have captured supports the formulation of a requirements model for a safety-critical system that can undergo PSA (a combination of 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. As well as the actual process, POSE captures many important outcomes of the process, including a record of the choices that have been made, the rationale for the revision of requirements statements, and rich traceability links from problem to solution space and back.

4 Case Study

In this case study we develop a safety specification for the Decoy Controller (*DC*) component of a defensive aids system on an aircraft, using the POSE safety requirements transformation pattern of Fig. 1. The study is based on a system flying on a real aircraft. The role of the *DC* is to control the release of decoy flares to provide defence against an incoming missile attack. The *DC* interfaces the Defence System (*DS*) computer which is responsible for controlling and orchestrating all the defensive aids on the aircraft. The *DS* and other domains (see Fig. 2) are given – that is, their design and interfaces already exist.

Here we apply POSE within a traditional safety critical development process capable of satisfying the provisions of DS 00-56 [20]. As is common practice in the industry, 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, including the defensive aids system. It also assumes that a system safety analysis has

been completed on the defensive aids system and this has allocated requirements to its sub-systems, including the *DC*. Two safety hazards were allocated to the *DC*, these being concerned with the inadvertent firing of the decoy flares, as follows:

H1: Inadvertent firing of the decoy flare on ground – Safety Target: safety critical, 10^{-7} fpfh.
H2: Inadvertent firing of the decoy flare in air – Safety Target: safety critical, 10^{-7} fpfh.

These hazards have a systematic ("safety related") and a probabilistic ("10^{-7} fpfh", where fpfh means failures per flight hour) component. To counter these hazards, the architectural design of the overall defensive aids system introduces a number of safety interlocks as input to the *DC* to provide safety protection. These are: an input from the pilot indicating whether the release should be allowed; an input indicating whether the aircraft is in the air; and an input indicating whether the safety pin, which is present when the aircraft is on the ground, is in place. The expected behaviour is that flare dispensing should be inhibited if any of the following conditions hold: a) the pilot disallows flares, b) the aircraft is not in the air, or c) the safety pin has not been removed. The on ground/in the air interlock provides extra assurance for hazard H1 but not for H2. Further, the safety pin is applied on the ground, but removed as the aircraft is readied for take off. So it too can provide protection for H1, but not necessarily for H2. Therefore, the safety task is to demonstrate that H2 can be feasibly satisfied at this level, with the knowledge that if H2 can be satisfied, then so can H1. In fact, H1 can achieve an even higher level of assurance.

4.1 Overview of the System

The first phase of the POSE safety transformation pattern (POSE pattern from henceforth) is "Representation", involving the identification of the major components in the problem, including the given domains with their phenomena and behaviours (refer to Section 3). The initial problem representation for the Flare Dispenser part of the aircraft defensive aids system is problem $\boldsymbol{P_{Initial}}$ shown in Fig. 2, with designation of phenomena given in Table 1.

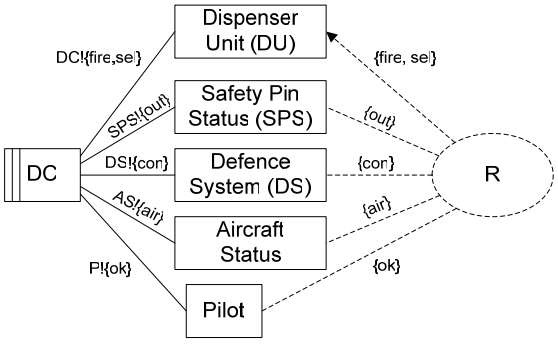

Fig. 2 The *DC* Problem ($P_{Initial}$)

The decoy flare *DU* (Dispenser Unit) has a number of different flare types which can be selected by control messages from the *DC* - the *sel* phenomenon in *DC!{fire,sel}*. The *DC* is told which flare type to select by control messages from the

Defence System (*DS!{con}*). Flare selection and timing are not safety related, it is only applying an inadvertent fire command to any flare that is regarded as a safety issue.

The selected flares are released by the *fire* command (in *DC!{fire,sel}*) from the *DC* to the *DU*. The *Pilot* domain inputs the allow release (*P!{ok}*) to the *DC*. The Aircraft Status domain inputs the in air status (*AS!{air}*) to the *DC*. The *SPS* provides the safety pin status (*out*) to the *DC*.

Table 1 *DC* Problem Phenomena Description

Phenomenon	Designation
fire	Command to release the selected flare
sel	Indicates which flare type should be selected
out	Pin status; “out = yes” indicates pin has been removed
con	Contains command to fire and selected flare type
air	Aircraft status; “air = yes” indicates aircraft is in the air
ok	Pilot intention; “ok = yes” indicates allow release

Inspection of the *DU* design documentation indicates that the *SPS* is co-located with but independent of the *DU* functionality. Therefore, these are represented as separate domains in Fig. 2. This partition introduces an assumption (A1) that independent common cause analysis will be performed to validate the separation claim. A1 is added to the requirement.

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

Ra: The *DS* shall command which flare is to be selected using a field in its *con* message issued to the *DC*. The *DC* shall obtain the selected flare information from this field in the *con* message, and use it in its *sel* message to the *DU* to control the flare selection in the *DU*.

Rb: The *DS* shall command the *DC* to issue a *fire* command in its *con* message. This shall be the only way in which a flare can be released.

Rc: The *DC* shall cause a flare to be released by issuing a *fire* command to the *DU*, which will fire the selected flare.

Rd: The *DC* shall only issue a *fire* command if its interlocks are satisfied – i.e. aircraft is in air (*air* = yes), *SPS* safety pin has been removed (*out* = yes) and pilot has issued an allow a release command (*ok* = yes).

As well as Ra to Rd, the *DC* must also satisfy its safety targets set by the aircraft system level safety analysis. Recognising this, we add safety requirement RS to R:

RS: The *DC* shall mitigate H1 & H2 (Target: safety critical & 10^{-7} fpfh).

Therefore, the overall requirement R = Ra & Rb & Rc & Rd & RS & A1, and is indicated in the dotted ellipse in Fig. 2. A complete statement of R should also include requirements that cover space, weight, environmental performance, interfaces and so on, but these are beyond the scope of this work.

4.2 A DC Candidate Architecture

The second POSE pattern phase is the problem "Interpretation" transformation which is used to introduce a candidate solution architecture for *DC.* The architecture of Fig. 3 is chosen based on the design strategy to minimise (where possible) the number and extent of the safety related functions and to localise them to simple, distinct blocks. This is a typical design strategy in the industry, hence the candidate architecture for *DC* allocates complex functionality to non-safety involved components. The *DC* architecture consists of three components, Safety Controller (*SC*), Decoy Microcontroller (*DM*) and Interlock Input (*II*), as shown in Fig. 3(a).

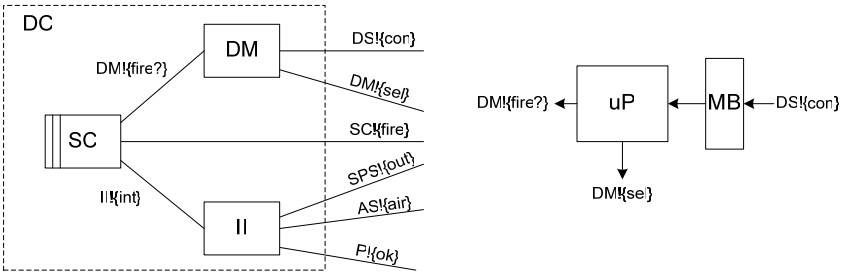

Fig. 3 (a) *DC* Candidate Architecture and (b) *DM* Internal Architecture

The given component *II* collects together the interlock inputs and passes their status to *SC* (via shared phenomenon *int).* It is a simple hardware block used for buffering the three discrete hardware interlock signals and conditioning them into a suitable input to the *SC*. The given component *DM* is a microcontroller used to decode messages from the Defence System (*con*), and when appropriate issue the fire command request to the *SC* (via *fire?*). The internal architecture of *DM* is shown in Fig. 3(b). The Message Buffer (*MB*) holds the received control message *con* from the *DS*. The microcontroller *uP* decodes this message to extract: a) the fire command request status (*fire?*) sent to the *SC*, and b) the selected flare type (*sel*) sent to the *DU*. The *SC*, the component to be designed, is intended as a simple block that handles the safety critical elements of the interlocking. *SC* is expected to relate an active *fire?* request to the *DU* (through phenomenon *fire*) if the interlocks are satisfied. Flare selection is not safety related, so it is not controlled by the *SC*.

The justification, J_1, for introducing this architecture is that it allows the safety critical functionality to be localised to a simple (by design intent) controller block, *SC*, and a simple hardware conditioning block, *II*. All the complexity is vested in the *DM*, which is not safety related. A simple block is easier to design and validate that its behaviour satisfies the safety critical systematic target (involving formal specification and proof of behaviour). In contrast, the complex message decoding, selection and timing is handled by the *DM* domain. The *SC* acts as a gate on the *DM* output, only allowing a fire command request to the *DC* if the safety interlocks are satisfied. This architecture potentially lowers the cost of the design solution significantly by minimising and localising the safety critical functionality. Moreover, the *DM* internal architecture satisfies the design goal of using a standard existing processor board architecture with processor (*uP*) and on-board message handling buffer (*MB*), as shown in Fig. 3(b).

Introducing the *DC* architecture transforms the requirements in R (section 3.1 Expansion transformation). Basically terms in *DC* are replaced by terms in *DM*, *SC* and *II* as appropriate. Inspection of R indicates that only RS remains unchanged by the *DC* expansion. R'a is the same as Ra, except *DC* is replaced by *DM*. R'c and R'd are the same as Rc and Rd respectively, except this time *DC* is replaced by *SC*. The significant change occurs for R'b, which is shown below with the changes in bold:

R'b: The *DS* shall command the ***DM*** to issue a *fire?* command in its *con* message. **The *DM* will request the SC to send the *fire* command**. This shall be the only way in which a flare can be released.

Therefore, R' = R'a & R'b & R'c & R'd & R'S & A1. The overall result of the transformation step is problem $\boldsymbol{P_{Interpreted}}$ shown in Fig. 4.

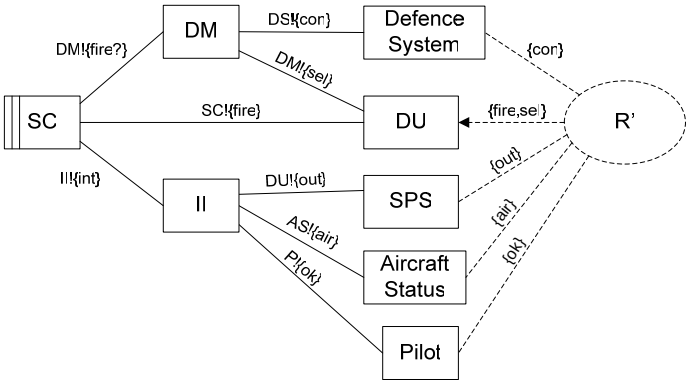

Fig. 4 Solution Interpretation of *DC* (problem $P_{Interpreted}$)

4.3 Problem Simplification

With the introduction of the candidate architecture, a PSA is required to see whether the *DC* architecture can safely be the basis of the *DC*. 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. In addition, we would like to transform the requirement R', that relates to system level entities, into a requirement that applies directly to the safety controller, *SC*, thus facilitating the derivation of a suitable specification for *SC*. This task corresponds to the third phase of the POSE pattern, "Reduction" (refer to Fig. 1). 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 [7, 8].

Generally, domains furthest from the machine to be designed (*SC* in the example) are removed first: in this case the first domain we remove is the *Pilot*. The reduction consists of two transformation steps. Firstly, the requirements relating to the *Pilot* are transformed appropriately to relate to the *II* domain. For example, R'd contains the term "the pilot has issued an allow release command (*ok* = yes)". Under the

transformation this will be re-phrased to refer to *II* by making transformations from *Pilot*-oriented phenomena into *II*-oriented phenomena. In this case the transformed term is "*II* observes the pilot input *ok* = yes". Secondly, the *Pilot* domain is removed from the problem context[1]. To complete this removal, we record assumptions that need to be in place for consistency. In this case the assumptions (A2) will include the fact that the pilot expresses an intention to allow flare release if the input *ok* = yes. That we rely on these assumptions is the justification, for this reduction. Transforming R' in this way yields a new requirement statement, that we will call R1, in which R'a becomes R1a, R'b becomes R1b, R'c becomes R1c and R'S becomes R1S without change. However, R'd is changed in the transformation to R1d as indicated below (changes are shown in bold):

R1d: The *SC* shall only issue a *fire* command if its interlocks are satisfied – i.e. aircraft is in air (*air* = yes), the *SPS* safety pin has been removed (*out* = yes) and ***II* observes pilot input *ok* = yes.**

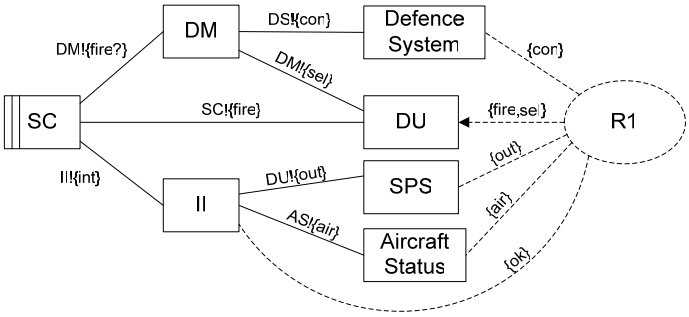

Fig. 5 Pilot Domain Removal (problem P_{Red1})

This removal transforms the problem from $P_{Interpreted}$ to P_{Red1} as shown in Fig. 5. At this point the requirement is R1 = R1a & R1b & R1c & R1d & R1S & A(1-2).

There are a number of domain removals that follow which, for brevity, we do not describe fully. These have the effect of removing, in turn, the *Aircraft Status* (*AS*) domain, the *SPS* domain and the *Defence System* (*DS*) domain. In the following, only the changed requirements are described, the other requirements translate unchanged and the justification assumptions not made explicit.

AS removal: in this removal, R1 is transformed first into R2 with assumption A3 as justification. Only R1d is modified to become,

R2d: The *SC* shall only issue a *fire* command if its interlocks are satisfied – i.e. ***II* observes input *air* = yes**, *SPS* safety pin has been removed (*out* = yes) and *II* observes pilot input *ok* = yes.

SPS removal: in this removal R2 is transformed into R3 with assumption A4 as justification. Only R2d is modified to become,

R3d: The *SC* shall only issue a *fire* command if its interlocks are satisfied – i.e. *II* observes input *air* = yes, ***II* observes input *out* = yes** and *II* observes pilot

[1] In the general case of this transformation, any relevant constraints on phenomena due to the presence of the to-be-removed domain should also be recorded as assumptions in a re-stated requirement. However, no such assumptions arise in this example.

input *ok* = yes.

DS removal: in this removal R3 is transformed into R4 with assumption A5 as justification. In this case, both R3a and R3b are modified to become respectively,

R4a: **The *DM* shall decide which flare to select by observing a field in the *con* message it receives**. The *DM* shall obtain the selected flare information from this field in the *con* message, and use it in its *sel* message to the *DU* to control the flare selection in the *DU*.

R4b: **The *DM* shall observe the *con* message it receives and issue a *fire?* command if commanded**. The *DM* will request the SC to send the *fire* command. This shall be the only way in which a flare can be released.

At this point, the problem $\boldsymbol{P_{Red1}}$ of Fig. 5 has been transformed into the problem $\boldsymbol{P_{Reduced}}$ of Fig. 6. Inspection of Fig. 6 shows it is much simpler than Fig. 5, and that the transformed requirement, R4, now relates directly to *SC*. For example, R4b, talks about "The *DM* will request the SC to send the *fire* command", and this corresponds directly to the phenomena shared between the *DM* and the *SC*, namely *fire?*. Therefore, Fig. 6 is a good model as the basis for the PSA work.

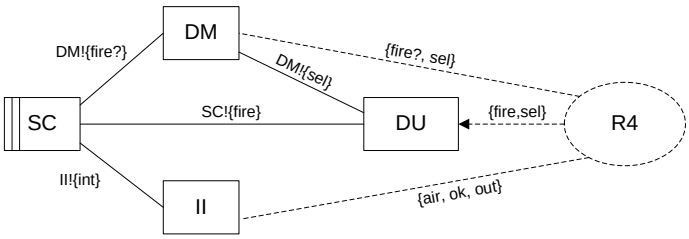

Fig. 6 The Reduced *SC* problem (problem $\boldsymbol{P_{Reduced}}$)

4.4 Formalising the Requirements

It is a simple step to formalise the requirements for input into the PSA. The corresponding POSE transformation is part of the Requirements Interpretation which, with justification, allows a requirement to be rewritten in an equivalent form (Interpretation (PSA) in Fig. 1). The non-safety aspects of the requirement can be formalised into a Parnas Table-like form, as shown in Table 2, with the safety targets and assumption appended as shown.

Table 2 Formalised Requirement R4, prior to PSA

Monitor Condition	Constraint
air = yes ∧ out = yes ∧ ok = yes ∧ fire? ∧ sel	fire ∧ sel
¬(air = yes ∧ out = yes ∧ ok = yes) ∧ fire? ∧ sel	¬fire ∧ sel
¬(air = yes ∧ out = yes ∧ ok = yes) ∧ ¬fire? ∧ sel	¬fire ∧ sel
R4S: H1 & H2 safety targets satisfied ∧ Assumptions A1 to A5 are valid	

4.5 Preliminary Safety Analysis (PSA)

The solution preserving nature of problem transformation under POSE leaves us at a point at which the solution of the reduced *SC* 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 *SC* problem – this will depend on the results of the PSA which will define if the reduced *SC* can satisfy its safety targets as defined by R4S.

In general, the goal of a PSA is to: (a) confirm any relevant hazards allocated by the system level hazard analysis; (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 *SC* problem exists. A number of techniques can be applied to perform a PSA. This work uses a combination of mathematical proof [15], Functional Failure Analysis (FFA) [17] and functional Fault Tree Analysis (FTA) [22].

The structuring provided by the POSE framework and the phased development means that it is relatively straightforward to develop formal Parnas table-like requirements (as in Table 1) that apply directly to the solution machine. Simple logic proofs demonstrate that R4 (Table 1) has the required functional properties. Therefore, the remaining feasibility check at this level is to demonstrate that the behaviour of the design blocks (*SC*, *DM*, and *II* in Fig. 6) can satisfy R4S.

Table 3 FFA Summary for *SC*

Id	Failure Mode	Effect	Hazard
F1	No *fire?* signal	Flare release inhibited	No
F2	*fire?* signal at wrong time	Inadvertent flare release	Yes
F3	*fire?* signal when not required	Inadvertent flare release	Yes
F4	Intermittent *fire?* signal	Could inhibit flare release	No
F5	Continuous *fire?* signal	Inadvertent flare release	Yes

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 F4S.

There is insufficient space to present the full PSA, hence we summarise only the main elements of the *DM* analysis to demonstrate the process followed. The significant results from applying FFA to the *DM* are shown in Table 3.

The functional FTA requires a suitable model and the architecture of Fig. 3(b) has an appropriate form. A functional FTA can be applied to this block diagram, using the three FFA problem cases (i.e. those with “Yes” in the Hazard column) F2, F3 and F5 as top events. The FTA indicates that a failure in uP (systematic or probabilistic) could result in the *fire?* failing on. The Pilot allow input provides some mitigation, but as soon as the allow input is set active (*ok* = yes) then a flare will be released, and this is an undesirable behaviour. Making the *fire?* signal integrity safety related (not

safety critical) would provide sufficient integrity, but this is contrary to the design aim of making the *DM* non-safety involved. The conclusion of the PSA is that the selected *DM* architecture is not a suitable basis for the design. The choice is either to: a) design the *DM* to be safety related, or b) re-structure the *DM* architecture with the goal of partitioning the safety and non-safety elements. The first option is undesirable due to the expense and long term impact – timing and selection are not safety functions and are expected to be fine-tuned to support different flare types. Making this safety related would have a detrimental impact on the affordability of the solution. Therefore the second option is more appealing, and a candidate architecture is shown in Fig. 7, where the simple safety functions (those associated with the *fire?* request) are routed separately through the MB and FPGA (Field-Programmable Gate Array), while the other complex functionality is routed through the MB and uP. This means that the MB and FPGA, which have simple functionality, have to be designed to a safety related standard, but this is still economic compared with the alternative of making the uP safety related.

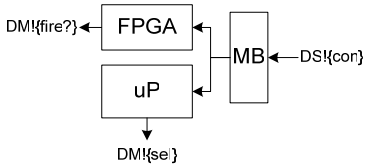

Fig. 7 Revised candidate Architecture for *DM*

The selection of the revised *DM* architecture (Fig. 7) means that the POSE pattern has to backtrack to "(Solution) Interpretation", as shown in Fig. 1. This is because the PSA has demonstrated that the original architecture choice for the *DM* being non-safety cannot form a feasible basis for the overall solution, hence it has to be revised. Only the information associated with the revised architecture is new, the rest of the information can be carried across from the earlier pass through the POSE pattern. Therefore the second application of the POSE pattern will be similar to the previous one, presented in Sections 4.2 to 4.4. The revised *DM* of Fig. 7 has the same interfaces as that of Fig. 3(b), hence the requirement resulting from the second set of reductions is the same as that obtained from the first - that of Table 2. The PSA applied to this revised architecture shows that the modified architecture satisfied R4S. Therefore, the revised architecture model obtained from the second run through of the POSE pattern (of Fig. 1) does form a suitable basis for the rest of the development.

5 Conclusions

The analysis in this paper demonstrates how the artefacts produced by following POSE can be used to (a) evolve the system level requirements such that they are directly related to the solution machine, (b) form the basis of the PSA work and (c) combine the functional and safety requirements into a coherent, safe requirements model that forms a good basis for the remainder of the development. Thus the POSE pattern is suitable as the front-end of an integrated safety critical development

approach suitable for embedded applications and supports the goal of extending the normal design concept to software.

In terms of Vincenti's three characteristics, the process we capture can be described thus:

- The safety analysis of section 4.5 demonstrated that failures in the *DM* domain could result in the safety targets not being satisfied. This is a form of "winnowing of the conceived variations" in that choices are restricted by the need to satisfy extra constraints, in this the safety analysis.
- A revised (but not new) architecture was developed, by which "engineering judgement was used to search past experience";
- And this was used to mitigate failure; novel features " that come to mind" were incorporated.

This work demonstrates that a POSE pattern introduced in [6] has a more general application than that it was recorded from. Although further case study work is required to demonstrate the use of this "pattern" on other application domains, the results from this case study are encouraging that POSE is able, to some extent, to record normal design practice.

References

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

[2] P. Coad, "Object Oriented Patterns," *Communications of the ACM*, vol. 35, pp. 152(8), 1992.

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

[4] 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.

[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. Hall, L. Rapanotti and D. Mannering, "Relating Safety Requirements and System Design through Problem Oriented Software Engineering," Open University, Dept. of Computing, Milton Keynes 2006/11, 2006.

[7] J. G. Hall and L. Rapanotti, "A framework for software problem analysis," Open University, Technical Report 2005/05 2005.

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

[9] M. Jackson, "Problem Frames and Software Engineering," *Information and Software Technology*, vol. 47, pp. 903-912, 2005.

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

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

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

[13] R. R. Lutz, "Analysing Software Requirements Errors in Safety-Critical Embedded Systems," IEEE International Symposium Requirements Engineering, San Diego, California, 1993.
[14] T. Maibaum, "Mathematical Foundations of Software Engineering: a roadmap," ICSE 2000, King's College, London, 2000.
[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, Denver, Colorado, 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] RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," December 1 1992.
[19] SAE, "ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment," December 1996.
[20] UK-MoD, "Safety Management Requirements for Defence Systems Part 1 Requirements," MoD, Interim Defence Standard 00-56 Issue 3, 17 December 2004.
[21] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective," ICSE'00, 22nd International Conference on Software Engineering, Limerick, 2000.
[22] W. Vesely, F. Goldberg, N. Roberts and D. Haasl, *Fault Tree Handbook*, vol. NUREG-0492: U.S. Nuclear Regulatory Commission, 1981.
[23] W. G. Vincenti, *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*: The Johns Hopkins University Press, 1990.
[24] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Transactions on Software Engineering and Methodology*, vol. VI, pp. 1-30, 1997.