Towards Normal Design for Safety-Critical Systems

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

[1] General Dynamics UK Limited
[2] Centre for Research in Computing, The Open University

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 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.

1 Introduction

Vincenti ([1]) defines 'normal design' as what 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'. Some have recently observed that software engineering does not: Maibaum [2] states that

> "SE ignores the principles of engineering design and almost always adopts radical design methods, to its detriment."

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 398–411, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Jackson [3] 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 tending to normality. This view has something to recommend it: typically, industrial standards and practices require integration with other normal engineering processes.

Vincenti [1] characterises normal design processes as: (a) relying on engineering judgement in the searching of past experience; (b) allowing the conceptual incorporation of the novel features that come to mind in solving a problem; and (c) allowing the "mental winnowing of the conceived variations" to pick out those most likely to work.

In a previous case study [4], we made a record of a current industrial safety-critical software intensive design process, using the Problem Oriented Software Engineering (POSE) framework [5]. Working from this record, the goal of this paper is to show how it might reflect Vincenti's three characteristics, and so be called 'normal'. To do this, we apply the record in the design of a different, functionally unrelated avionics system component and find that the record is a good fit to the needs of safety-critical development and consider this to be evidence that—to some limited extent—POSE offers an approach to normal design for 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 Work

POSE is an extension and generalisation of Jackson's Problem Frame approach [6]. Problem Frames attempt to keep the focus of the software engineer on developing their understanding of the problem to be solved, rather than on a (premature) move to solution of a poorly understood problem. Problem Frames make certain fundamental assumptions: primary is the separation of descriptions of what is given—the *indicative* parts of a problem—from what is required—the *optative* parts of a problem. Originally confined to Requirements Engineering, the influence of Problem Frames has spread to the fields of domain modelling, business process modelling, software architectures and early design—see [7,8,9] for collections of recent work.

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 [10]. 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, a combination of hazard identification and preliminary hazard analysis as required by the safety standards) in place of PSSA. In this paper requirements follow the fundamental clarification work of Jackson [20] and Parnas [12] 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 [13]. Therefore there is a large semantic divide between the system level requirements and the specification of the machine solution. One of the reasons for applying POSE is that it bridges the divide by transforming system level requirements into requirements that apply 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 [12], which is particularly well suited to defining embedded critical applications as shown by its use in SCR [14] and the RSML methods. The RSML work led to the SpecTRM [15] methods, which form part of a human centred, safety-driven process which is supported by an artefact called an Intent specification [16]. The work in this paper is located in the area of the second-level System Design Principles of the Intent specification, and thus may be complementary to the third, Blackbox level provided by SpecTRM.

The work of Anderson, de Lemos, and Saeed [17] 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 safety analysis. The latter means it is efficient as there is no need to develop 'new' models (with all their attendant validation problems) to be able to perform a PSA. Further, its support for traceability makes it particularly suited for use with standards such as DS 00-56 [18] and the DO-178B [19] software guidelines.

3 Problem Oriented Software Engineering

POSE (see [5] for the formal definition) recognises that software engineering processes by necessity include 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 developed system will provide the functionality and qualities that are needed.

Briefly, POSE generalises and extends fundamental ideas expressing the completeness of requirements engineering [20,21] and those of problem orientation (see, for instance, [6]) to apply to software engineering.

In POSE, software development is viewed as solving a *problem*, the *solution* (S, a labelled double-barred box in the figures that follow) being a *machine*—that is, a program running in a computer—that will ensure satisfaction of the *requirement* (R, the dotted oval in the figures) in the given *problem world* (W) consisting of real-world domains (the labelled but otherwise undecorated boxes in the figures). Typically the requirement concerns properties and behaviours that are located in the problem world at some distance from its interface with the machine. Like Problem Frames, POSE views the problem world W as a collections of *domains* described in terms of their known, or indicative, properties, which interact through their sharing of *phenomena*, i.e, events, commands, states, *etc* (that decorate the arcs in the figures).

POSE is defined as a Gentzen-style sequent calculus that allows problems to be transformed into problems that are easier to solve, or that will lead to other problems that are easier to solve. A set of transformation rule schema defined in the calculus capture (atomic) discrete steps in development. Each requires a justification of application in order for the transformation to be solution preserving, although justifications need not be formal. The combination of the justifications is an argument that the solution is adequate as a solution to the original problem. The interested reader should consult [5] for a fuller presentation of POSE.

POSE problem transformations transform problems in ways that respects solution adequacy: simplifying only slightly, this means that a solution to a transformed problem is also a solution to the original problem.

In the following section we will give highlights of a POSE development of an avionics case study (more detail of the case study is given in [4]). For brevity, we present the development in graphical form, using a Problem Frame-like notation rather than the Gentzen-style presentation in [5]. Moreover, we present only the relevant details of Problem Frames in this paper as and when they are needed; a thorough presentation is beyond the scope of this paper, and can be found in [6].

3.1 A Problem-Oriented Approach to Safety Analysis

From previous work [4], we observe that POSE transformations can be combined to form re-usable process templates or "patterns" [22] for safety-critical development. One such process is shown in Fig. 1 as a UML activity diagram. The activities in the figure include the following POSE transformations[1]:

Domain and Requirement Interpretation used to capture increasing knowledge and detail in the context and requirement of the problem (used in activities 1 and 4);

Solution Expansion used to structure the solution according to a candidate architecture (used in activity 2);

Problem Progression by which the problem is simplified by removing domains (used in activity 3).

The choice point (labelled 5) in the figure is the PSA by which the candidate architecture is assessed for suitability. The outcome of the PSA determines whether

[1] Defined and described in [5,23,24].

the current architecture is viable as the basis of a solution or whether backtracking in needed so that another candidate architecture can be chosen.

The pattern is iterative, ending when an architecture suitable for solution development is found. This process is iterative in that design choices, through the choice of candidate architecture, influence requirements, and vice versa.

As we shall see, POSE allows the capture of many important other artefacts of the process, including a record of the choices that have been made, the rationale for the revision of requirements statements.

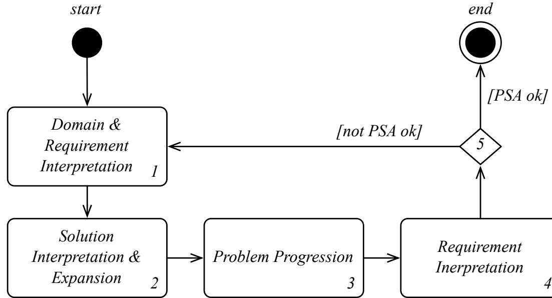

Fig. 1. POSE Safety Pattern

4 The Case Study

The POSE pattern emerged from the realisation that PSA feasibility checks can identify the inadequacy of the architecture early in a development, and avoid the cost associated with rework. The case study from which it was derived and that of this paper are real developments, underdone by the first author, based on systems flying in real aircraft. The case studies are cut-down only in the sense that some detail has been removed for brevity, and they retain all essential complexity. The POSE pattern was applied (retrospectively) in the context of the first case study to confirm that architecture inadequacy could be identified earlier in development (and, perhaps, therefore result in cost savings). In this paper, we use the POSE pattern to guide a safety critical development process capable of satisfying the provisions of DS 00-56 [18].

The case study concerns the development of the *Decoy Controller* (*DC*) component of a defensive aids system on an aircraft, as shown in Fig. 2. The *DC*'s role is to control the release of decoy flares providing defence against incoming missile attack. The *DC* interfaces with 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) already exist (and so appear as undecorated boxes in the figure).

As is common practice in the industry, we will assumes that an aircraft level safety analysis has been completed with safety requirements being allocated to the main aircraft systems, including the defensive aids system. This analysis

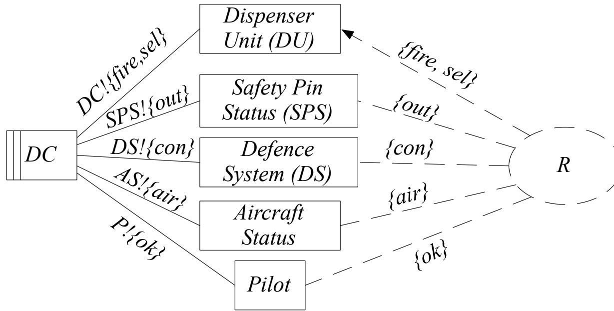

Fig. 2. The *DC* Problem ($P_{Initial}$)

has allocated requirements to the defensive aids sub-system, including the *DC*: in fact, there are two safety hazards allocated to the *DC*, concerned with the inadvertent firing of the decoy flares, as follows:

H1. Inadvertent firing of decoy flare on ground. Safety Target: safety critical, 10^{-7} fpfh[2]; and
H2. Inadvertent firing of decoy flare in air. Safety Target: safety critical, 10^{-7} fpfh.

These hazards have both systematic (safety related) and probabilistic components. 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, 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. It transpires that these interlocks provide extra assurance for hazard **H1** but not for **H2**. Therefore, the safety task is to demonstrate that **H2** can be satisfied, with the knowledge that if **H2** can be satisfied, then so can **H1**.

4.1 The process

The first activity in the POSE pattern—"Domain and Requirement Interpretation and Expansion," (labelled 1 in Fig. 1)—details the problem context and requirement. This works by identification of the major system components, and the description of their phenomena and their behaviour. The initial problem representation is problem $P_{Initial}$ shown in Fig. 2, with interface phenomena given in Table 1. Here we summarise the problems components.

The decoy flare *DU* (Dispenser Unit) has a number of different flare types which can be selected by control messages from the *DC*—the chosen type being

[2] *fpfh* is 'failures per flight hour.'

communicated in the *sel* phenomena ($DC!\{fire, sel\}$[3], in the figure). The DC is told which flare type to select by phenomena controlled by the Defence System ($DS!\{con\}$)[4].

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. Phenomena of *DC* Problem

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

The customer requirement for the DC can be expressed as follows:

Ra. The *DS* shall command which flare is 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 is **R** = **Ra** & **Rb** & **Rc** & **Rd** & **RS**, 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 initial work.

[3] That *fire* and *sel* are controlled by *DC* is indicated by the ! on the arc.

[4] Note, 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.

4.2 A DC Candidate Architecture

The next POSE pattern activity—"Solution Interpretation and Expansion", labelled 2 in Fig. 1—introduces the candidate architecture for *DC* shown in Fig. 3. The *DC* architecture consists of three components, Safety Controller (*SC*), Decoy Micro-controller (*DM*, shown in Fig. 3(b)) and Interlock Input (*II*), as shown in Fig. 3(a). This choice of architecture is typical of industrial safety design strategies that attempt to minimise the number and extent of the safety related functions, localising them to simple, distinct blocks. These strategies justify the candidature of the architecture, and are recorded, under POSE, as part of the justification for the transformations involved.

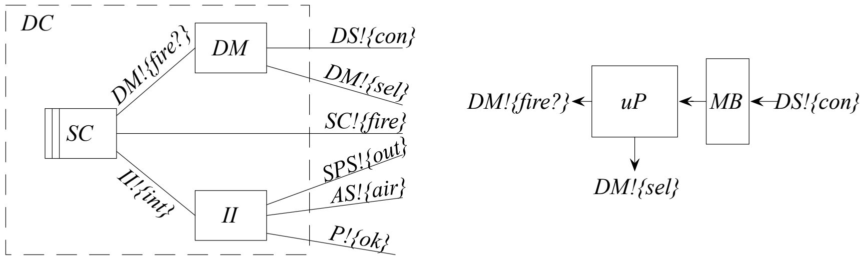

Fig. 3. (a) The *DC* Candidate Architecture and (b) *DM* Internal Architecture

Briefly, component *II* collects together the interlock inputs and passes their status to *SC* (*int*). Component *DM* is a microcontroller used to decode messages from the Defence System (*con*), and when appropriate to issue the fire command request to the *SC* (via *fire*?). The Message Buffer (*MB*, in Fig. 3(b)) holds the received control message *con* from the *DS*. The micro-controller *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, therefore, expected to relate an active *fire*? request to the *DU* (through phenomenon *fire*) if the interlocks are satisfied.

The introduction of the *DC* architecture affects the requirement **R** as terms in *DC* are replaced by terms in *DM*, *SC* and *II* as appropriate. The result is a new requirement statement

$$\mathbf{R'} = \mathbf{R'a} \,\&\, \mathbf{R'b} \,\&\, \mathbf{R'c} \,\&\, \mathbf{R'd} \,\&\, \mathbf{R'S}$$

in which **R'a** is **Ra** with *DC* replaced by *DM*; similarly for **R'c** and **R'd**, *mutandis mutatis*. The most significant change occurs for **R'b** (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.

The result of the transformation step is problem $P_{Interpreted}$ shown in Fig. 4.

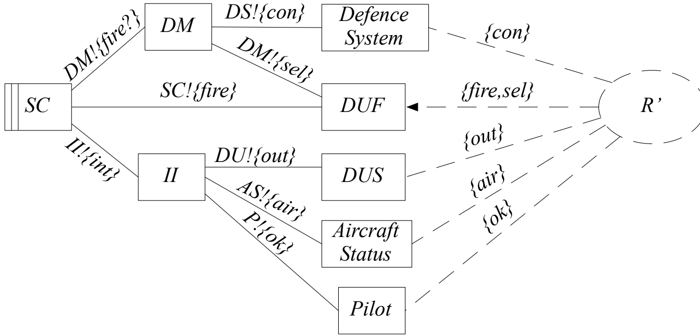

Fig. 4. Solution Interpretation of *DC* (problem $P_{Interpreted}$)

4.3 Problem Simplification

Of course, this candidate architecture is not guaranteed to lead to a solution; and we will use a PSA to determine whether the *DC* architecture can safely be the basis of the *DC*. Performing a PSA on the problem shown in Fig. 4 is an unnecessarily complex task that can be simplified by removing some of the contextual domains. Domain removal is achieved in POSE through "Problem Progression" (labelled 3 in Fig. 1), which simultaneously allows us to transform the requirement **R'** to apply directly to the safety controller, *SC*, used in the simplified PSA.

For brevity, we show only the removal of the *Pilot* and the associated requirements transformation. To remove the *Pilot* we must alter the requirement so that shared phenomena constrained by the *Pilot*'s actions are recorded: in this case, only that the release command ($ok = yes$) can occur[5]. In this case the requirement is rewritten to include the assumption $A1$ "the input $ok{=}yes$" as well as including "*II* observes the pilot input $ok = yes$". Given this rewriting of the requirement, the *Pilot* domain can be removed.

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. **R'd** becomes (changes 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*.**

There are a number of domain removals (and assumptions to the requirement) that follow which, for brevity, we do not describe fully[6]; they result in the problem shown in Fig. 5 which is a better basis for the PSA.

[5] Because the *Pilot* is an autonomous agent. If we fail to make this assumption, the problem becomes trivial.

[6] See [25] for more details.

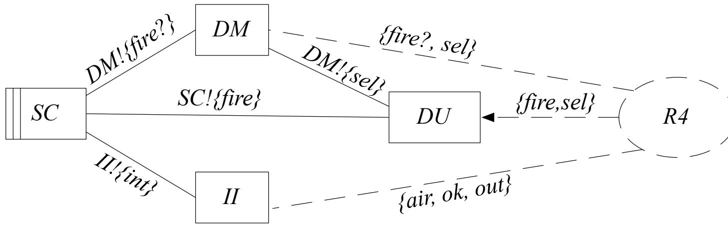

Fig. 5. The Reduced *SC* problem (problem $P_{Reduced}$)

4.4 Formalising the Requirements

The next activity—"Requirements Interpretation" in Fig. 1—is the formalisation of **R4** for input into the PSA. One must ensure that the justification of the transformation properly relates informal and form requirements, in this case a simple task. The non-safety aspects of the requirement can be formalised into a Parnas Table-like form, shown in Table 2, with the safety targets and assumption appended as shown.

Table 2. Formalised Requirement R4, prior to PSA

Monitor Condition	Output Constraint
$air = yes \wedge out = yes \wedge ok = yes \wedge fire? \wedge sel$	$fire \wedge sel$
$\neg(air = yes \wedge out = yes \wedge ok = yes) \wedge fire? \wedge sel$	$\neg fire \wedge sel$
$\neg(air = yes \wedge out = yes \wedge ok = yes) \wedge \neg fire? \wedge sel$	$\neg fire \wedge sel$
R4S: **H1** & **H2** safety targets satisfied and Assumptions A1 to A4 are valid	

4.5 Preliminary Safety Analysis (PSA)

Many techniques can be applied to perform a PSA. The work of this case study uses a combination of mathematical proof, Functional Failure Analysis (FFA) [10] and functional Fault Tree Analysis (FTA) [26].

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) analyse an architecture to validate that it can satisfy the safety targets associated with the identified relevant hazards. The solution preserving nature of problem transformation under POSE means that any solution of the progressed $P_{Reduced}$ problem will be a solution to the $P_{Initial}$ problem[7]. Simply, if the PSA fails, there is no feasible solution to $P_{Reduced}$.

The structuring provided by the POSE framework and the phased development means that it is relatively straightforward to develop a formal Parnas table-like requirement (as in Table 2) that applies directly to the solution machine.

[7] We do not, of course, yet know that either has a solution; it is also worth noting that $P_{Initial}$ may have a solution without $P_{Reduced}$ having one.

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

Simple logic proofs demonstrate that **R4** (Table 2) has the required functional properties. The remaining feasibility check at this level is to demonstrate that the behaviour of the design blocks (*SC*, *DM*, and *II* in Fig. 5) can satisfy **R4S**.

The FFA can be used to identify any additional relevant hazards and, 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 **R4S**. There is insufficient space to present the full PSA, and so we summarise only its main elements 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 this is set ($ok = yes$) a flare will be released, which is 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, then, that the selected *DM* architecture is not a suitable basis for the design—no adequate solution can be derived from its parametrisation. Choices at this point include: a) designing the *DM* to be safety related, or b) re-structuring the *DM* architecture to partition the safety and non-safety elements. The first option is undesirable due to the expense and long term impact, i.e. 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. The second option is more appealing, and a second candidate architecture is shown in Fig. 6 in which the simple safety functions (those associated with the *fire*? request) are routed separately through *MB* and *FPGA* (a Field-Programmable Gate Array, [27,28]), while the other complex functionality is routed through *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.

The failed PSA leads to iteration of the process. We note, only the information associated with the revised architecture is new, the remainder of the performed transformations can be carried across from the first iteration, simplifying the second (and any subsequent) iteration. The second iteration of the POSE pattern will be similar to the first. Indeed, the revised *DM* of Fig. 6 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 shown in Table 2. Although we do not show it, the PSA applied to this revised architecture shows that the modified architecture satisfies **R4S**; the revised architecture model obtained from the second run through of the POSE pattern can form a suitable basis for the remainder of the development.

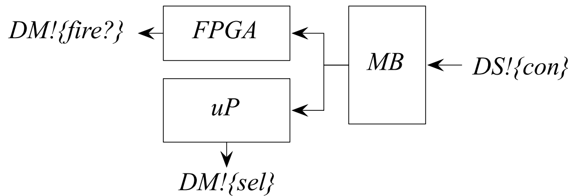

Fig. 6. Revised Candidate Architecture for *DM*

There is still work to do—we do not have *SC* as yet; a fresh application of the process would be possible at this point. This concludes our description of the controller synthesis.

5 Discussion and Conclusions

We have illustrated the synthesis of a controller for a safety-critical system under POSE. We have had, from necessity, to omit many details of the process, but we hope that some of the complexity of the development has remained, in particular, those showing how POSE structures and guides development of the product whilst recording the related justifications and, hence, the adequacy argument for it.

The process by which the synthesis was achieved was captured and re-used from a previous unrelated safety-critical development. The process appears to exhibit Vincenti's three characteristics, namely:

– 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 variation" 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.

We might therefore conclude that the study provides early validation that the pattern may be suitable as the front-end of an integrated safety critical development approach for embedded applications, hence supporting the goal of extending the normal design concept to critical software. Of course, further and more conclusive validation is still required and we are working closely with industrial partners to exercise the framework on further real-world safety critical problems. A wider application of the pattern to other software engineering domains is also under investigation.

The case-study as presented was cut-down to its most important, and complex, aspects to fit within the page limit. Most of the routine detail was omitted, but in its original form it was based directly on an industrial avionics example, and from a conceptual viewpoint the POSE pattern worked well. However, as size and complexity grow the desirability for tool support to handle the detail, the more mundane tasks and keeping track of progress is greatly increased. Tool support for POSE is ready to start development, based on an existing, successful commercial tool for safety-critical analysis and assurance.

Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Grants. Thanks also go to our colleagues in the Centre for Research in Computing and the Computing Department at The Open University, especially Michael Jackson. The comments of the three anonymous reviewers have helped in improving the paper greatly.

References

1. Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. The Johns Hopkins University Press (1990)
2. Maibaum, T.: Mathematical foundations of software engineering: a roadmap. In: ICSE 2000, King's College, London (2000)
3. Jackson, M.: Problem frames and software engineering. Information and Software Technology **47**(14) (2005) 903–912
4. Mannering, D., Hall, J.G., Rapanotti, L.: Relating safety requirements and system design through problem oriented software engineering. Technical Report 2006/11, Open University, Dept. of Computing (2006)
5. Hall, J.G., Rapanotti, L., Jackson, M.A.: Problem oriented software engineering. Technical Report 2006/10, Open University, Dept. of Computing (2006)
6. Jackson, M.A.: Problem Frames: Analyzing and Structuring Software Development Problem. 1st edn. Addison-Wesley Publishing Company (2001)
7. Cox, K., Hall, J.G., Rapanotti, L., eds.: Proceedings of ICSE 1st International Workshop on Applications and Advances of Problem Frames, IEEE CS Press (2004)
8. Cox, K., Hall, J.G., Rapanotti, L., eds.: Journal of Information and Software Technology: Special issue on Problem Frames. Volume 47. Elsevier (November 2005)

9. Hall, J.G., Rapanotti, L., Cox, K., Jin, Z.: Proceedings of the 2nd International Workshop on Advances and Applications of Problem Frames, ACM SIGSOFT (2006)
10. SAE: ARP4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. Technical report (December 1996)
11. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology **VI**(1) (1997) 1–30
12. Courtois, P.J., Parnas, D.L.: Documentation for safety critical software. In: 15th International Conference on Software Engineering, Baltimore, USA (1997) 315–323
13. van Lamsweerde, A.: Requirements engineering in the year 00: A research perspective. In: ICSE'00, 22nd International Conference on Software Engineering, Limerick (2000)
14. Bharadwaj, R., Heitmeyer, C.: Developing high assurance avionics systems with the SCR requirements method. In: Proceedings. DASC. The 19th. Volume 1. (2000) pages 1D1/1 –1D1/8
15. Leveson, N.G.: 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 (2000) 2000
16. Leveson, N.G.: Intent specifications: An approach to building human-centered specifications. IEEE Transactions on Software Engineering **Vol. 26**(1) (2000) 15–35
17. de Lemos, R., Saeed, A., Anderson, T.: On the integration of requirements analysis and safety analysis for safety-critical systems. Technical Report http://citeseer.ist.psu.edu/536230.html, University of Newcastle upon Tyne (1998)
18. UK-MoD: Safety management requirements for defence systems part 1 requirements. Interim Defence Standard 00-56 Issue 3, MoD (17 December 2004)
19. RTCA/DO-178B: Software considerations in airborne systems and equipment certification. Technical report (December 1 1992)
20. Zave, P., Jackson, M.A.: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology **6**(1) (1997) 1–30
21. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A reference model for requirements and specifications. IEEE Software **17**(3) (2000) 37–43
22. Coad, P.: Object oriented patterns. Communications of the ACM **35**(9) (1992) 152–160
23. Rapanotti, L., Hall, J.G., Jackson, M.: Problem-oriented software engineering: solving the package router control problem. Technical report 2006/07, Open University, Dept. of Computing (2006)
24. Rapanotti, L., Hall, J.G., Li, Z.: Deriving specifications from requirements through problem reduction. **153**(5) (October 2006) 183–210
25. Mannering, D., Hall, J.G., Rapanotti, L.: A problem-oriented approach to normal design for safety-critical systems. Technical Report 2006/14, Centre for Research in Computing (2006)
26. Vesely, W., Goldberg, F., Roberts, N., Haasl, D.: Fault Tree Handbook. Volume NUREG-0492. U.S. Nuclear Regulatory Commission (1981)
27. Hilton, A.J., Townson, G., Hall, J.G.: FPGAs in critical hardware/software systems. In: FPGA 2003, Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. (2003) 244
28. Hilton, A., Hall, J.G.: Developing critical systems with PLD components. In Margaria, T., Massink, M., eds.: FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems, New York, NY, USA, ACM Press (2005) 72–79