ISSN 1744-1986

The Open University

Technical Report N° 2007/04

Arguing safety with Problem Oriented Software Engineering

Jon G Hall
Derek Mannering
Lucia Rapanotti

30th March 2007

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

http://computing.open.ac.uk

Arguing safety with Problem Oriented Software Engineering

J.G. Hall
Centre for Research in Computing
The Open University
Milton Keynes, UK
J.G.Hall@open.ac.uk

D.Mannering
General Dynamics UK Ltd.
St Leonards on Sea, UK
Derek.Mannering@generaldynamics.uk.com

L.Rapanotti
Centre for Research in Computing
The Open University
Milton Keynes, UK
L.Rapanotti@open.ac.uk

ABSTRACT

Standards demand that assurance cases support safety critical developments. It is widely acknowledged, however, that the current practice of post-hoc assurance—that the product is built and only then argued for safety—leads to many engineering process deficiencies, extra expense, and poorer products. This paper shows how the Problem Oriented Software Engineering framework supports the concurrent design of a safe product and its safety case, by which these deficiencies can be addressed.

The basis of the paper is a real development, undertaken by the second author of this paper, of safety-related subsystems of systems flying in real aircraft. The case study retains all essential detail and complexity.

Keywords

Software Engineering, Safety critical systems, Safety case, Concurrent design, Problem Oriented Software Engineering

1. INTRODUCTION

System engineering processes by necessity include the identification and clarification of system requirements, the understanding and structuring of the context into which the system will be deployed, the specification of a design for a solution that can ensure satisfaction of the requirements in context, and the construction of arguments, convincing for all validating stake-holders, that the system will provide the functionality and qualities that are needed.

In critical contexts, such as the safety-critical systems found in civil and military aircraft, standards and regulations typically constrain designs at a number of different system levels, ranging from the integrated system operating in its design environment through to specific constraints on the choice of software languages and component type for achieving the requisite confidence in the product. Importantly, they introduce a requirement for safety cases to be built.

ESEC/FSE 2007 Dubrovnik Croatia

A safety case [7, 3] is a documented body of evidence providing a compelling, comprehensive and valid argument that a system is adequately safe for a given application in a given environment. Safety standard DS 00-56 ([7]) mandates that a safety case should address:

- the management of risk commensurate with the potential risk posed by the system and its complexity. In this context, risk management is the activity through which hazards and potential accidents are identified, and appropriate mitigation strategies chosen, e.g., hazard elimination or risk reduction. It is assumed that the ALARP principle applies: a risk is ALARP (for "As Low As Reasonably Practicable") if the cost of any further risk reduction is demonstrably grossly disproportionate to its benefits.
- the validity of the safety requirements, i.e., that they are derived through thorough analysis and are traceable with respect to the system as designed and implemented, together with evidence of their satisfaction.
- the well-foundedness of assumptions about the system, its operating environment or modes of use upon which the safety argument is based, with a justification that such assumptions are realistic and reasonable.

The standard also recommends that the safety case should contain not just evidence about the product, but also process evidence—attesting to good practice in development, maintenance and operation—and evidence on good engineering judgement and design.

Although the most convincing possible safety case is one which is valid and sound, Kelly and Weaver observe [13] that, due to the nature of the evidence in safety cases, a *provably* valid and sound case is unobtainable. In practice, there is wide-spread reliance on (necessarily) subjective expert judgement and claims of adherence to standards. The basis of conviction is simply a belief that a claim is true, argued and evidenced by strong chains of reasoning from believably adequate grounds. Again, this is recognised in current safety standards, which require only that the safety argument should be structured and based on evidence.

There remain acknowledged difficulties in the construction of safety cases. Here we characterise them as:

- The difficulty of Disparate Descriptions: that of combining disparate pieces of the evidence, such as narrative, requirements, claims, plans, activities or goals [4];

- The difficulty of Post-Hoc Assurance: that traditionally, safety cases are developed post-design and testing with known drawbacks including: expensive re-design when the current design is indefensible; expensive system over-engineering so that the design can be defended; and loss of the rationale for the safety aspects of the design [12].

In previous work [16, 17], we have shown how POSE can be used to link disparate descriptions, arguments and evidence in a safety-critical context, by which the first difficulty can be addressed. In this paper, we argue that POSE also provides a framework in which the second of these difficulties can be addressed. As evidence we apply POSE to a real-world case study, for which we produce and manage the safety case *alongside* and *with influence on* the safety-critical design of a safe solution. In addition, we also address those elements of a safety case mandated by [7] mentioned above, i.e., risk, validity, and the well-foundedness of assumptions.

The paper is structured as follow. Section 2 provides a brief introduction to POSE. Section 3 discusses related work. Section 4 discusses POSE for safety-critical development and assurance. Section 5 presents the case study. Section 6 concludes the paper with a discussion and conclusions.

2. PROBLEM ORIENTED SOFTWARE ENGINEERING

The Problem Oriented Software Engineering framework of [9] is a Gentzen-style sequent calculus [15] for 'solving' software problems. The basis of a Genzten-style sequent calculus is a *sequent*: a well-formed formula, traditionally representing a logical proposition. The purpose of a sequent is to provide a vehicle for the representation of a logical proposition and for its transformation into other logical propositions in truth-sense preserving ways. In traditional Gentzen-style sequent calculi, if we can transform a logical proposition to the axioms of the system, we have shown its universal truth; the collection of transformations used form a proof that stands as definitive record of the demonstration.

In POSE, sequents represent *software problems*, i.e., problems that have a software solution (see below). The transformations defined in POSE transform software problems as sequents into others in ways that preserve solutions (in a sense that will become clear). When we have managed to transform a problem to 'axioms' we have solved the problem, and will have a software solution to show for our efforts.

The Gentzen-style sequent calculus used in POSE has features that extend its traditional form. The most important of these is the guarding of transformations by *justification obligations*, the discharge of which establishes the 'soundness' of the application *with respect to some developmental stake-holder*. This is a radical departure from the universality of truth that an unguarded traditional Genzten-style sequent calculus can show, and it is unique to POSE. As to the benefits of such guarding, freed from the need to demonstrate that a solution is universally correct, we can think about the forms of justification that are needed during design to convince the actual stake-holders of the adequacy of the solution. For instance, perhaps a development with rigorous or formal proofs of correctness and one with a testing-based justification of adequacy would both suffice for the resource constrained corporate buyer; our point is that the one based on testing will be more affordable and deliverable, as long as formal correctness is not amongst the needs of the customer.

We do not eschew formality; indeed, POSE is a formal system for working with non-formal and formal descriptions. Moreover, formality may sometimes be appropriate when strict stake-holders—such as regulatory bodies governing the development of the most safety-critical of software—are involved. However, as we know from the real world, only when focused is formality appropriate.

Our claim is that POSE offers a practical approach to engineering design in which the possible roles of formality are separated out, and made clear.

2.1 Software problems

A software problem has three elements: a real-world context, W, a requirement, R, and a solution, S.

The problem context is a collection of *domains* ($W = D_1, ..., D_n$) described in terms of their known, or *indicative*, properties, which interact through their sharing of *phenomena* (i.e, events, commands, states, *etc.* [11]). More precisely, a *domain* is a set of related phenomena that are usefully treated as a behavioural unit for some purpose. A domain $D(p)^c_o = N : E$ has *name* (N) and *description* (E), the description indicating the possible values and/or states that the domain's phenomena (in $p \cup c \cup o$) can occupy, how those values and states change over time, how phenomena occur, and when. Of the phenomena:

- c are those *controlled* by D, i.e., visible to, and sharable by, other domains but whose occurrence is controlled by D;
- o are those *observed* by D, i.e., made visible by other domains, whose occurrence is observed by D;
- p are those *unshared* by D, i.e., sharable by no other domain.

A problem's requirement states how a proposed solution description will be assessed as the solution to that problem. Like a domain, a requirement is a named description with phenomena, $R^{cons}_{refs} = N : E$. A requirement description should always be interpreted in the optative mood, i.e., as expressing a wish. As to the requirement's phenomena:

- *cons* are those *constrained* by R, i.e., whose occurrence is constrained by the requirement, and whose occurrence the solution affects in providing a solution;
- *refs* are those *referenced* by R, i.e., whose occurrence is referred to but not constrained by the requirement.

A software solution is a domain, $S(p)^c_o = N : E$, that is intended to solve a problem, i.e., when introduced into the problem context will satisfy the problem's requirement. The possible descriptions of a solution range over many forms, from high-level specification through to program code. As a domain, a solution has controlled, observed and unshared phenomena; the union of the controlled and observed sets is termed the *specification phenomena* for the problem.

A problem's elements come together in POSE in a *problem sequent*[1]:

$$D_1(p_1)^{c_1}_{o_1}, \ldots, D_n(p_n)^{c_n}_{o_n}, S(p)^c_o \vdash R^{cons}_{ref}$$

[1]As here, for brevity, we will sometimes omit the phenomena decorations and descriptions in W, S and R whenever they can be inferred by context.

Here $\vdash$ is the *problem builder* and reminds us that it is the relation of the solution to its context and to the requirements that we seek to explore. By convention, the problem's solution domain, S, is always positioned immediately to the left of the $\vdash$.

The descriptions of a problem's elements may be in any language, different elements being described in different languages, should that be appropriate. So that descriptions in many languages may be used together in the same problem, POSE provides a *semantic meta-level* for the combination of descriptions; notationally, this is a role of the ',' that collects into a problem sequent the domains that appear around the turnstile, formally making each visible to the others[2].

2.2 Problem transformation

Problem transformations capture discrete steps in the problem solving process. Many classes of transformations are recognised in POSE, reflecting a variety of software engineering practices reported in the literature or observed elsewhere. Problem transformations relate a problem and a justification to a (set of) problems. Problem transformations conform to the following general pattern. Suppose we have problems $W, S \vdash R$, $W_i, S_i \vdash R_i$, $i = 1, ..., n$, $(n \geq 0)$ and justification J, then we will write:

$$\frac{W_1, S_1 \vdash R_1 \qquad ... \qquad W_n, S_n \vdash R_n}{W, S \vdash R} \begin{array}{l}[\textsc{Name}] \\ \langle\!\langle \mathrm{J} \rangle\!\rangle\end{array}$$

to mean that, derived from an application of the NAME problem transformation schema (discussed below):

> S is a solution of $W, S \vdash R$ with *adequacy argument* $(CA_1 \wedge ... \wedge CA_n) \wedge J$ whenever $S_1, ..., S_n$ are solutions of $W_1, S_1 \vdash R_1, ..., W_n, S_n \vdash R_n$, with adequacy arguments $CA_1, ..., CA_n$, respectively.

Software engineering design under POSE proceeds in a step-wise manner: the initial problem forms the root of a *development tree* with transformations applied to extend the tree upwards towards its leaves. Branches are completed by problem transformations that leave the empty set of premise problems.

2.3 Problem transformation schemata

A *problem transformation schema* defines a named class of problem transformations, describing the way in which the *conclusion* problem (that below the line) is related to the *premise* problem(s) (those above the line). How a problem is transformed is given in a problem transformation schema by pattern matching of the elements of the conclusion problem, with those matched elements repeated as appropriate to specialise both the premise problem(s) and *justification obligation* (explained below).

Here is the transformation schema for CONTEXT INTERPRETATION by which the context W is *interpreted* as W':

$$\frac{\mathcal{W}', \mathcal{S} \vdash \mathcal{R}}{\mathcal{W}, \mathcal{S} \vdash \mathcal{R}} \begin{array}{l}[\textsc{Context Interpretation}] \\ \langle\!\langle \text{Explain and justify the use of } \mathcal{W}' \text{ over } \mathcal{W} \rangle\!\rangle\end{array}$$

The justification obligation is a condition that must be discharged for an application of a schema to be solution preserving. Each schema has its own general form of justification obligation; that for CONTEXT INTERPRETATION is shown in the rule. However, the specific form will depend upon the development context as well as other factors.

In Section 4, we discuss in more detail the form of the justification when the development context is safety-critical.

[2]A situation similar to that found in the propositional calculus in which conjunction and disjunction serve to combine the truth values of the atomic propositions.

3. RELATED WORK

Problem Frames are a conceptual framework for Requirements Engineering, based on the problem-oriented foundation laid by Michael Jackson over a number of years, the culmination being [11]. The popularity of Problem Frames is growing: see [5, 6] for some recent work. Whilst sharing a conceptual basis, POSE is both an extension and generalisation of Problem Frames in the following ways: all forms of description in the solution space are admitted: specifications, high- and low-level design, code, *etc.*; structuring of the solution space is possible using *Architectural Structures* (see Section 5.3); problem solving is transformational, providing rich traceability between problem and solution domains; the range of POSE problem transformations goes beyond Problem Frames' problem decomposition; and problem transformations are accompanied by justification obligations that confirm the adequacy of the transformation, with respect to various criteria. The fact that POSE extends into the solution domain implies that iterative design processes—processes that span both problem and solution domains—can be supported.

Approaches to software development based on various logics and calculi have been the subject of computer science for many years, and much has been learned about the logics, calculi, and their derivatives, that are best suited to describe software. Transformations of a similar nature to those in POSE are sometimes found in these formal approaches to software development; examples include the transformations of specifications through to program code found in the refinement calculi of Morgan [19] and Back [2] and, more recently, the categorical refinements of Smith [22].

Two structured notations for expressing safety cases have emerged. One is the goal-structuring notation (GSN) [14], a graphical argumentation notation which allows the representation of individual elements of a safety argument and their relations. Elements include: goals (used to represent requirements and claims about the system), context (used to represent the rationale for the approach and the context in which goals are stated), solutions (used to give evidence of goal satisfaction) and strategies (the approach used to identify sub-goals). The other, is Adelard's Claim-Argument-Evidence (ASCAD) approach [3], which is based on Toulmin's work on argumentation [24] and includes: claims (same as Toulmin's claims), evidence (same as Toulmin's grounds) and argument (combination of Toulmin's warrant and backing). More recently, Habli and Kelly [8] have also suggested ways in which product and process evidence could be combined in GSN assurance cases.

Some very recent work by Strunk and Knight [23] proposes Assurance Based Development (ABD) in which a safety-critical system and its assurance case are developed in parallel through the combined used of Problem Frames [11] and GSN.

4. POSE FOR SAFETY-CRITICAL DEVELOPMENT

We have already applied POSE in support of safety-related

developments [16, 17]. In those papers our focus was on the evaluation for safety of proposed candidate solution structures (i.e., partial solutions; architectures) early in development. From those papers, and work supporting them, a 'process pattern' has emerged. Its elements are shown in Fig. 1 as a UML activity diagram. The activities in the figure include the following:

Context and Requirement Interpretation to capture (increasing) knowledge and detail in the context and requirement of the problem (Activity 1);

Solution Intepretation and Expansion to structure the solution (or part thereof) according to a candidate architecture (Activity 2);

Preliminary safety analysis (PSA) for early assessment of a candidate architecture (Activity 3).

The choice point (labelled 4) in the figure depends on the outcome of the PSA, which determines whether the current candidate architecture is viable as the basis of a solution or whether backtracking is needed so that the problem should be changed in some way or another candidate architecture chosen. The techniques applied for the PSA will depend on the level of criticality of the system under design and may include Functional Failure Analysis (FFA) [21], functional Fault Tree Analysis (FTA) [25], or the use of fully formal specification languages and logical proof. The level of criticality is determined by whether the system is *safety critical* (highest integrity required) or *safety related* (high integrity, but not as high as safety critical).

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

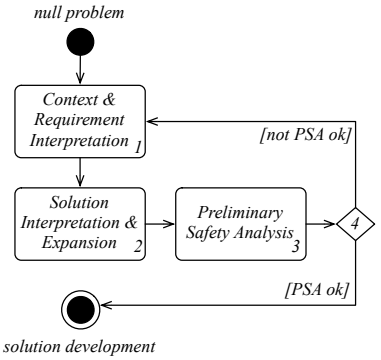


Figure 1: POSE Safety Pattern: to move towards the solution of a safety-critical problem, we first understand the problem better (Activity 1), use engineering judgement to determine a candidate solution architecture (Activity 2), then test the candidate for satisfaction of safety concerns, iterating if necessary.

4.1 Building a safety case

In this paper, we focus on safety critical development, whence the justification obligation must satisfy the interested stake-holder that their *concerns* (similar in nature to those in [11]) about safety are discharged. The justification obligation for a schema will contain, as well as other elements, the imperative that the concerns associated with the problem transformation step it defines should be discharged. A concern therefore leads to a *claim* stated within a justification, the claim being that the concern is discharged by the development step. The justification will, eventually, contain arguments and evidence that the claim is valid. We say eventually because some concerns can only be discharged after the ramifications of a problem transformation are known which is, typically, later in the development tree.

For reasons of space, we are not able to give an exhaustive presentation of the concerns related to each of the POSE schemata. However, for those schemata encountered in this paper, the following concerns arise:

- the *well-founded concern*, for which the associated claim is that a particular domain description is well-founded, i.e., realistic and reasonable;
- the *reliability concern*, for which the associated claim is that a particular domain, described in an interpreted problem, is adequately reliable;
- the *feasibility concern*, for which the associated claim is that a chosen architecture, i.e., (partial) solution structure, should not prevent an adequately safe solution from being found;
- the *sound judgement concern*, for which the associated claim is that a particular development choice is based on sound engineering judgement.

A summary of the relationship between the schemata and concerns used within this paper is shown in Table 1.

SCHEMA / *Concern*	*Well-foundedness*	*Reliability*	*Sound judgement*	*Feasibility*
INTERPRETATION SCHEMATA				
CONTEXT	X	X		
SOLUTION	X	X	X	X
REQ[MNTS]	X			

Table 1: Transformation Schemata and their standard concerns

The justifications in a development tree combine to give the *adequacy argument* for the development. Eventually, a developer will have to construct an argument (or many arguments) for customers and/or other validating stake-holders that convinces them of the solution's adequacy with respect to their criteria: the adequacy argument constructed during development is intended to be the definitive source of such arguments. When one validating stake-holder is a safety regulator, the adequacy argument must, as already mentioned, include a safety case, and it is this aspect of adequacy with which we are most concerned in this paper.

Of course, the concerns of the stake-holders have always driven development, to a greater or lesser extent. However, the need to discharge concerns explicitly *as a development step is taken* adds two strong drivers to development:

- the claims, arguments and evidence needed can guide the detail of a development, including the justification of choices as and when they are made;

- a very richly traceable development path through which a record of both the steps that lead to the delivered product, and those that were discarded—presumably because they were not able to discharge some concern or other—is built.

This tight *concurrent design* (or *co-design*) of the product and its safety case is the basis of our claim that we can, in POSE, address what we have termed the difficulty of Post-Hoc Assurance.

4.2 Notational conventions

So that it is easier to interleave development prose, descriptions and figures with the explanatory narrative, we will describe each transformation step by separating it from the narrative, using the following graphical device. Suppose we wish to transform the problem $P(= W, S \vdash R)$ under the NAME transformation schema, then we will write:

Application of NAME *to problem P*

JUSTIFICATION J: describing the named justification (J) for the application of transformation NAME to P. The name can be used elsewhere in the development to refer to this justification. The body of the justification can have many components—prose, formal descriptions, figures, for instance—and any or all elements of the following structure may be present:

INCLUDES: identifying any relationships between this justification and others in the development such as those that occurred from an earlier step, that was subsequently discovered to be inadequate and so backtracked from. The inadequacy can also be described here.

CONCERNS: A collection of standard concerns associated with each transformation schema. Each concern is discharged by argument and evidence supporting a claim of the following form:

CLAIM: *Claim statement*

ARGUMENT & EVIDENCE: The reason to believe the claim, or the reason it does not hold.

PHENOMENA: Should the schema introduce phenomena, or need to detail their sharing, the details can be included here.

5. CASE STUDY

The case study is a real development, underdone by the second author of this paper, based on systems flying in real aircraft. The case study is cut-down only in the sense that some detail has been removed for brevity, and it retains all essential complexity. It concerns the development of the *Decoy Controller* component of a defensive aids system on an aircraft, whose role is to control the release of decoy flares providing defence against incoming missile attack.

5.1 The starting point

In POSE, all problem solving starts from the *null problem* (P_{null}), that of which we know nothing; its introduction requires no justification, and it can be thought of as the beginning of all POSE developments:

$$P_{null}: \quad W : null, S : null \vdash R : null$$

Here, *null* is used as the description for W, R and S to indicate that nothing is known about them: *null* has less information than any description that can be written in any language chosen for descriptions. It is a point of contact between all description languages used in a problem. Moving from the *null* problem to the specific problem of this case study was done in the following way: after a successful bid to tender, the mechanical outline, approximate weight and power envelope of the system was established. Subsequent communications were used to clarify the requirements and properties of the system environment. The remainder of the system was designed in response to the post bid revised customer requirements including their allocation to software and hardware as appropriate. The details are given below.

5.2 Context and Requirement Interpretation

Problem descriptions are captured in POSE through the various transformation schemata for *interpretation*, including that of CONTEXT INTERPRETATION introduced in Section 2.3. In what follows, for brevity we summarise the various interpretations of [17] which from P_{null} leads us to the following problem:

$$P_1: \begin{array}{l} \mathit{Defence\ System}^{con}, \mathit{Dispenser\ Unit}^{out}_{fire,sel}, \\ \quad \mathit{Aircraft\ Status\ System}^{air}, \\ \quad \mathit{Pilot}^{ok}, \mathit{Decoy\ Controller}^{fire,sel}_{con,out,air,ok} \\ \vdash R^{fire,sel}_{con,out,air,ok} \end{array}$$

The justification obligation of all interpretation schemata requires us to justify a newly provided description over an existing one. Here is the (collated) justification for all intepretation transformations from P_{null} to P_1, which add knowledge of the problems and its parts starting from *null* descriptions.

Application of CONTEXT AND REQUIREMENT INTERPRETATION *to problem* P_{null}

JUSTIFICATION J_1: The identified requirement, domains and their relevant properties are summarised below:

Name	Description
Defence System	The computer responsible for controlling and orchestrating all defensive aids on the aircraft
Dispenser Unit	Mechanical device for releasing decoy flares used as defence against incoming missile attack. It has number of different flare types, and includes a safety pin that, when in place, prevents flares from being released

continued

Name	Description
Aircraft Status System	The system which monitors the status of certain key aircraft parameters, including whether the aircraft is in the air
Pilot	The pilot, who can signal the controller that flare release should be allowed
Decoy Controller	*null*
R	The requirement is the conjunction of: R_a: On receiving a *con* command from *Defence System*, *Decoy Controller* shall obtain the selected flare type information from the relevant field in *con*, and use it in its *sel* message to the *Dispenser Unit* to control flare selection in that unit. R_b: *Decoy Controller* shall issue a *fire* command only on receiving a *con* command from *Defence System*. This shall be the only way in which a flare can be released. R_c: *Decoy Controller* shall cause a flare to be released by issuing a *fire* command to the *Dispenser Unit*, which will fire the selected flare. R_d: *Decoy Controller* shall only issue a fire command if its interlocks are satisfied, i.e. aircraft is in air ($air = yes$), safety pin has been removed ($out = yes$) and pilot has issued an allow a release command ($ok = yes$). R_S: *Decoy Controller* shall mitigate H_1 and H_2, where: H_1: Inadvertent firing of decoy flare on ground. Safety Target: safety critical, 10^{-7} fpfh[3]; and H_2: Inadvertent firing of decoy flare in air. Safety Target: safety critical, 10^{-7} fpfh.

PHENOMENA: Phenomena and their control and sharing (see P_1) are known from the existing system components as:

Name	Description
fire	Command to release the selected flare type
sel	Command to select flare type
out	Pin status: $out = yes$ when pin removed
con	Command to select and release a flare type
air	Aircraft status: $air = yes$ when aircraft airborne
ok	Pilot intention: $ok = yes$ when allow release

CLAIM: *The interpretations are well-founded*

ARGUMENT & EVIDENCE: The choice of domains follows from the aircraft level safety analysis and the required choice of interlocks. The *Defence System*, *Dispenser Unit*, *Aircraft Status System* are existing components of the avionics system, with well-known properties (that could be validated through direct inspection). The *Pilot* is trained to follow protocol rigorously.

The customer requirement was provided as an input to the developer team. Hazard H_1 and H_2 came from an aircraft level safety analysis which allocated safety requirements to the main aircraft systems, including the *Decoy Controller*. Hazards H_1 and H_2 have both systematic (safety related) and probabilistic components. To counter these hazards, the following safety interlocks were required as input to the *Decoy Controller* to provide safety protection: 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 release 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. These interlocks provide extra assurance for hazard H_1, but not for H_2. Therefore, the safety task is to demonstrate that H_2 can be satisfied, with the knowledge that if H_2 can be satisfied, then so can H_1.

CLAIM: *The in-air indicator in Aircraft Status System is reliable*

ARGUMENT & EVIDENCE: The in-air indicator is obtained from the weight on wheels and landing gear up indications: if the landing gear is up and there is no weight on the wheels then the aircraft is assumed to be in the air. The landing gear is detected as being up by a number of sensor switches. The switches use a multi-pole arrangement of appropriately selected "Normally open/Normally closed" contacts. This imbues an error detection capability that is used to achieve very good failure rates, well within the required margins.

There are other claims that would be made of each interpretation that we do not include here for reasons of space.

5.3 Solution Interpretation and Expansion

An *AStruct* (short for *Architectural Structure*) is used to add structure to a solution domain, through an application of SOLUTION INTERPRETATION. An *AStruct* combines, in a given topology, a number of known solution components[4] (the C_i below) with solution components yet to be found (the S_j below). Its general form is:

$$AStructName[C_1, ..., C_m](S_1, ..., S_n)$$

with *AStructName* the *AStruct* name. Once the solution is interpreted by providing and justifying an *AStruct*, SOLUTION EXPANSION generates premise problems by moving the already known components C_i to the environment—expanding the problem context—whilst simultaneously refocussing the problem to be that of finding the solution components S_j that remain to be designed. The requirement and context of the original problem is propagated to all subproblems.

[4]There are also constraints on the phenomena sets, which we omit here for brevity; the reader is referred to [9] for the full definition.

A particular case, which is relevant to our case study, is when there is only one component to be found, that is, the *AStruct* has the following form:

$$AStructName[C_1, ..., C_m](S)$$

In this case expansion only generates one premise problem as follows:

$$\frac{\mathcal{W}, \mathcal{C}_1, ..., \mathcal{C}_m, \mathcal{S}{:}null \vdash \mathcal{R}}{\mathcal{W}, S : AStructName[\mathcal{C}_1, ..., \mathcal{C}_m](\mathcal{S}) \vdash \mathcal{R}} \text{ [Solution Expansion]}$$

In the case study, the following *AStruct* encodes the initial candidate architecture chosen for the *Decoy Controller*:

$$DecoyContAS[II^{int}_{ok,air,out}, DM^{sel,fire?}_{con}](Safety\ Controller^{fire}_{int,fire?})$$

which includes two extant components, *II* and *DM* and one to be found component *Safety Controller*. Therefore, a subsequent expansion leads to problem:

$$P_2 : \begin{array}{l} Defence\ System^{con}, Dispenser\ Unit^{out}_{fire,sel}, \\ \quad Aircraft\ Status\ System^{air}, Pilot^{ok}, \\ \quad II^{int}_{ok,air,out}, DM^{sel,fire?}_{con}, Safety\ Controller^{fire}_{int,fire?} \\ \vdash R^{fire,fire?,sel}_{con,out,air,ok} \end{array}$$

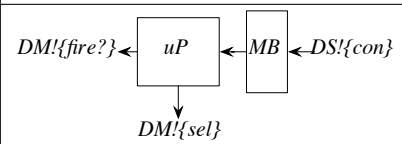
Here is the combined development step:

Application of Solution Interpretation and Expansion *to problem* P_1

Justification J_2: The identified architecture[5], its components and relevant properties are summarised in the table below:

Name	Description
Decoy Controller	$DecoyContAS[II^{int}_{ok,air,out}, DM^{sel,fire?}_{con}](Safety\ Controller^{fire}_{int,fire?})$
II	Collects together the interlock inputs and passes their status to *Safety Controller* (*int*)
DM	A microcontroller used to decode *con* messages from *Defence System* and when appropriate issue a fire command request, *fire*?, to the *Safety Controller*. A schematic diagram of its hardware components is given below: the message buffer *MB* holds the received control message *con*; the micro-controller *uP* decodes it to extract: a) a fire command request (leading to *fire*?), and b) the selected flare type (leading to *sel*).
	continued

[5]Note that a requirement interpretation is often required after solution interpretation and before expansion in order to adjust the names of those domains introduced as part of the architecture. Usually, as in this case, such renaming is straight-forward.

Name	Description
	
Safety Controller	*null*

Claim: *The choice of candidate solution architecture exhibits sound safety engineering judgement*

Argument & Evidence: The architecture is chosen to minimise the number and extent of the safety related functions, localising them to simple, distinct blocks in accordance with best practice. The *Safety Controller*, the only safety-critical component, is a simple block that handles the safety-critical elements of the interlocking.

Claim: *The chosen solution architecture does not prevent the satisfaction of R.* **This claim is not yet supported.**

Phenomena: The new phenomena introduced by the architecture are:

Name	Description
fire?	Command to release the selected flare type
int	Status of combined interlocks

5.4 Preliminary Safety Analysis

The justification of the previous transformation step is not complete. The *feasibility concern* remains to be discharged. The related claim is that the chosen architecture candidate should not prevent an adequately safe solution, and yet, as we shall see, the chosen architecture candidate does prevent an adequately safe solution. In this (and the general) case, to continue the design without checking feasibility uncovers the risk that there will need to be (expensive) redevelopment if the final product cannot be argued safe. Although, traditionally, this risk is mitigated through what might be called over-engineering of the solution, such over-engineering is yet another development cost. This risk can be managed in POSE through an eager Preliminary Safety Analysis (PSA) that can be applied as the means to discharge the feasibility concern.

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. Many techniques can be applied to perform a PSA. In [17] we used a combination of mathematical proof, Functional Failure Analysis (FFA) [21] and functional Fault Tree Analysis (FTA) [25].

Note that PSA is not a POSE transformation per se (no POSE schema defines a PSA). Instead it is a technique which we use to discharge one of the concerns in the justification obligation for Solution Interpretation.

Application of SOLUTION INTERPRETATION AND EXPANSION *to problem P_1 (cont'd)*

CLAIM: *The chosen solution architecture does not prevent the satisfaction of R.* **This claim does not hold.**

ARGUMENT & EVIDENCE: We applied FFA to each architectural component in turn. The significant results[6] from applying FFA to the *DM* are shown in Table 2, where three problem cases were identified: F2, F3 and F5, with 'Yes' in the Hazard column.

Table 2: FFA Summary for *Safety Controller*

Id	Failure Md	Effect	Haz
F1	No *fire*?	Release inhibited	No
F2	*fire*? at wrong time	Inadvertent release	Yes
F3	*fire*? when not required	Inadvertent release	Yes
F4	Intermittent *fire*?	Could inhibit release	No
F5	Continuous *fire*?	Inadvertent release	Yes

A functional FTA applied to *DM* and using the three FFA problem cases F2, F3 and F5, indicates that a failure in *uP* (systematic or probabilistic) could result in the *fire*? failing on. The *Pilot*'s allow input provides some mitigation, but as soon as this is set ($ok = yes$) a flare will be released, which is undesirable behaviour. In other words, with this architecture, H_2 is only protected by the *Pilot*'s allow input. If *fire*? failed on, then as soon as the *Pilot* indicated an intention to allow flare release, by selecting the switch, then the flare would be released, which is not the design intention.

Therefore the safety analysis indicates that *fire*? needs to have a safety involved (not critical) integrity. This can only be achieved with the existing design by upgrading all of the design to be safety involved. That is, by assigning *fire*? to the *uP*, we require that all *uP* functionality must be of *fire*?s required safety integrity, including much of the *uP*s functionality (timing, BIT, etc.) that is not safety-related. Further, any updates to the *uP* software have to satisfy the safety involved integrity. To make the *uP* safety-involved is not possible. The conclusion of the PSA is that the selected *DM* component, hence the architecture, is not a suitable basis for the design—no adequate solution can be derived from its parametrisation, hence the feasibility concern cannot be discharged.

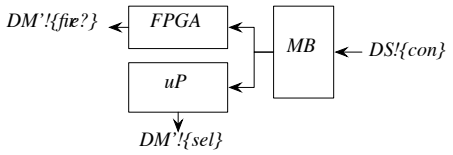
5.5 Backtracking the development

The failed PSA leads to backtracking of the development, and to iteration of the POSE process. In particular, we backtrack development to P_1, choosing a second candidate architecture informed by what we learned from trying to discharge the failed feasibility claim. In this way we have allowed safety concerns for a design choice to influence the product, during its development. We have also managed a development risk very close to where it was discovered.

The second iteration of the POSE process is similar to the first: although there is new information associated with the revised architecture, the remainder of the transformations may be carried across from the first iteration without change, simplifying this second (and any subsequent) iteration. The second candidate architecture differs from the original in that we replace *DM* with higher integrity component DM'. Here is the development step:

Application of SOLUTION INTERPRETATION AND EXPANSION *to problem P_1, after backtracking*

JUSTIFICATION J_2': The newly identified architecture, its components and relevant properties are summarised below (where they differ from J_2):

Name	Description
Decoy Controller	$DecoyContAS[II^{int}_{ok,air,out}, DM'^{sel,fire?}_{con}](Safety\ Controller^{fire}_{int,fire?})$
DM'	A microcontroller used to decode *con* messages from *Defence System* and when appropriate issue a fire command request, *fire*?, to the *Safety Controller*. A schematic diagram of its hardware components is given below: the message buffer *MB* holds the received control message *con*; the micro-controller *uP* decodes it to extract the selected flare type (leading to *sel*); the FPGA (a Field-Programmable Gate Array, [10]) component decodes it to extract a fire command request (leading to *fire*?). 
Safety Controller	*null*

INCLUDES: Includes J_2, with alterations as discussed below.

CLAIM: *The choice of candidate solution architecture exhibits sound safety engineering judgement*

ARGUMENT & EVIDENCE: The chosen architecture is similar to the previous one (see J_2) except that as a result of the PSA we require the *fire*? signal to be safety involved (but not safety critical) so that to allow the overall architecture to satisfy its safety target. We do this by taking the safety involved functions out of the *uP* component and route them through a separate high integrity path. As a result, we choose a new component DM' in which there is a partition between the safety and non-safety elements: the simple safety functions (those associated with the *fire*? request) are routed separately through *MB* and *FPGA*, while the other complex functionality is routed through *MB* and *uP*. This means that

[6]There is insufficient space to present the full PSA, and so we summarise only its main elements to demonstrate the process followed.

only *MB* and *FPGA*, have to be designed to a safety related standard.

CLAIM: *The chosen solution architecture does not prevent the satisfaction of R.* **This claim is not yet supported.**

As a result of this step, we arrive at:

$$P_2': \begin{array}{l} \mathit{Defence\ System}^{con}, \mathit{Dispenser\ Unit}^{out}_{fire,sel}, \\ \quad \mathit{Aircraft\ Status\ System}^{air}, \mathit{Pilot}^{ok}, \\ \quad \mathit{II}^{int}_{ok,air,out}, \mathit{DM}'^{sel,fire?}_{con}, \mathit{Safety\ Controller}^{fire}_{int,fire?} \\ \vdash R^{fire,fire?,sel}_{con,out,air,ok} \end{array}$$

The next step forward is a PSA to discharge the extant claim from the second solution interpretation, i.e., to discharge the feasibility concern for the new architecture. The idea behind this second PSA is that in the revised architecture the *fire*? processing is sent via the FPGA, which can be of simple functionality. For such a dedicated device, the required systematic and failure rate integrity can be demonstrated, leading to the result that the revised architecture is feasible. The remaining detail of the PSA is omitted for brevity.

5.6 Concluding the development

The subsequent design of the *Safety Controller*, which, again, we do not include for brevity, is carried out under POSE through a solution interpretation whose justification includes supporting evidence that the design is indeed a solution to the problem.

In terms of POSE development, we have now reached a leaf in the development tree with no other branches to be pursued and with no justification obligations left pending. Therefore, we have reached a solution where the architecture of the *Decoy Controller* is completely known and whose adequacy is argued through the conjunction of all discharged justifications in the corresponding tree. Such conjunction includes the outcome of the PSA, arguing the feasibility of the architecture in addressing the safety requirement.

6. DISCUSSION

The POSE notion of problem requires a separation of context, requirement and solution, with explicit descriptions of what is given, what is required and what is designed. This improves the traceability of artefacts and their relation, as well as exposing all assumptions to scrutiny and validation. That all descriptions are generated through problem transformation forces the inclusion of an explicit justification that such assumptions are realistic and reasonable. In particular, safety requirements are justified as valid, are fully traceable with respect to the designed system, and evidence of their satisfaction is provided by the adequacy argument of a completed POSE development tree. A choice of candidate solution requires that a justification of good engineering judgement is given—another piece of evidence which is now required for safety assurance.

The case study development has illustrated many essential elements of the use of POSE in support of safety critical system development. In particular, through it we have shown how a safety case can be constructed concurrently with the development of a safe product, and that a regard for safety during development leads to detailed examination of design choices during development whilst managing development risk and reducing the tendency to over-engineer just to be sure that the product is defensible.

The applied POSE safety pattern includes a (development) risk assessment in the form of a PSA; following iterations provide low-cost risk mitigation through redevelopment. The form the PSA takes depends on the level of criticality applied, and hence will follow the ALARP principle. The outcome of the PSA, encapsulated in transformation justifications, contributes to explicit evidence of risk management and is fully contextualised in terms of environment assumptions, safety requirements and chosen design. The proposed PSA has the added bonus of being applicable to early architectural design and provides early evidence of feasibility of the proposed solution. That no extra artefacts are required for this step makes the approach highly cost-effective.

That product and safety argument development are co-designed is a fundamental characteristic of POSE: no transformation should occur without appropriate justification (although such justification may not be immediately available, requiring some exploratory development to be done first). On the other hand, development risks can be taken by tentative transformation which are not completely justified: in such cases concerns can be stated as suspended justification obligations to be discharged later on in the process. This adds the flexibility of trying out solutions, while still retaining the rigour of development and clearly identifying points where backtracking may occur.

Finally, POSE defines a clear formal structure in which the various element of evidence fit, that is whether they are associated with the distinguished parts of a development problem or the justifications of the transformation applied to solve it. This provides a fundamental clarification of the type of evidence provided and reasoning applied. Moreover, that the form of justification is not prescribed under POSE signifies that all required forms of reasoning can be accommodated, from deductive to judgemental, within a single development.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have illustrated how issues of producing and managing the safety reasoning involved in critical system development can be addressed through POSE. In particular, we have provided some evidence on how POSE may contribute to those elements of a safety case arguing requirements validity and satisfaction, explicit context assumptions, design judgement and rationale, and safety risk management, and demonstrated the approach on a real-world example.

In future work, we plan to explore larger software and system engineering designs in POSE. This will require many things, whose development may speed the adoption of POSE ideas in industry. The first is tool support: Gentzen-style sequent calculi are known to be amenable to automated support, *viz.* PVS [20], Gumtree [18], and others. The second is the availability of larger and more realistic case studies, for which we continue to look to industry.

8. ACKNOWLEDGMENTS

We acknowledge the financial support of IBM, under the Eclipse Innovation Grants. We particularly thank Colin Brain of S E Validation Limited for the many comments and insights he has given into POSE. Thanks also go to our colleagues in the Centre for Research in Computing at The Open University, particularly Michael Jackson.

9. REFERENCES

[1] AssWS. Workshop on assurance cases: Best practices, possible obstacles and future opportunities. Florence, Italy, 2004. Co-located with the International Conference on Dependable Systems and Networks.

[2] R.-J. Back and J. von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, pages 367–384, 1994.

[3] R. Bloomfield, P. Bishop, C. Jones, and P. Froome. *ASCAD - Adelard Safety Case Development Manual*, 1998.

[4] P. Caseley, N. Tudor, and C. O'Halloran. The case for an evidence based approach to software certification. Safety standards review committee, UK Ministry of Defence, 2003.

[5] K. Cox, J. G. Hall, and L. Rapanotti, editors. *Journal of Information and Software Technology: Special issue on Problem Frames*, volume 47. Elsevier, November 2005.

[6] K. Cox, J. G. Hall, and L. Rapanotti. A roadmap of problem frames research. *Journal of Information and Software Technology*, 47(14):891–902, 2005.

[7] DS0056-3. Defence Standard 00-56 issue 3, 1997. Safety Management Requirements for Defence Systems.

[8] I. Habli and T. Kelly. Achieving integrated process and product safety arguments. In *Proceedings of 15th Safety Critical Systems Symposium (SSS'07)*. Springer, 2007.

[9] J. G. Hall, L. Rapanotti, and M. Jackson. Problem-oriented software engineering. Technical Report 2006/10, Department of Computing, The Open University, 2006.

[10] A. Hilton and J. G. Hall. Developing critical systems with PLD components. In T. Margaria and M. Massink, editors, *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 72–79, New York, NY, USA, 2005. ACM Press.

[11] M. A. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Publishing Company, 1st edition, 2001.

[12] T. Kelly. A systematic approach to safety case management. In *Proceedings SAE 2004 World Congress*, Detroit, US, 2004.

[13] T. Kelly and R. Weaver. The goal structuring notaion - a safety argument notation. In *[1]*.

[14] T. P. Kelly. *Arguing safety - A systematic approach*. PhD thesis, Department of Computing, University of York, 1998.

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

[16] D. Mannering, J. G. Hall, and L. Rapanotti. Relating safety requirements and system design through problem oriented software engineering. Technical Report TR2006/11, Computing Department, The Open University, 2006.

[17] D. Mannering, J. G. Hall, and L. Rapanotti. Towards normal design for safety-critical systems. In M. B. Dwyer and A. Lopes, editors, *Proceedings of FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 398–411. Springer Verlag Berlin Heidelberg, 2007. To appear.

[18] A. Martin, R. Nickson, and M. Utting. Improving angel's parallel operator: Gumtree's approach. Technical Report 97-15, University of Queensland, Australia, 1997.

[19] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1994.

[20] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1992.

[21] SAE. ARP4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. Technical report, December 1996.

[22] D. Smith. Comprehension by derivation. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 3–9. IWPC, 15-16 May 2005.

[23] E. A. Strunk and J. C. Knight. The essential synthesis of problem frames and assurance cases. In *International Workshop on Advances and Applications of Problem Frames*, 2006.

[24] Toulmin. *The uses of argument*. Cambridge University Press, 1958.

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