Assurance-driven design in Problem Oriented Engineering*

Jon G. Hall Lucia Rapanotti
Department of Computing
The Open University, UK
{J.G.Hall,L.Rapanotti}@open.ac.uk

Abstract

The design of assurance cases is hampered by the posit-and-prove approach to software and systems engineering; it has been observed that, traditionally, a product is produced and then evidence from the development is looked for to build an assurance case. Although post-hoc assured development is possible, it often results in errors being uncovered late—leading to costly redevelopment—or to systems being over-engineered—which also escalates cost. As a consequence, there has been a recent move towards the proactive design of the assurance case. Assurance-driven design sees assurance as a driving force in design. Assurance-driven design is suggestive of how the design process should be shaped for assurance. It is not, however, a prescriptive method; rather it allows an organisation to assess their assurance needs according to their developmental needs, including their attitude to risk, and to adapt their processes accordingly.

We have situated the work within Problem Oriented Engineering, a design framework inspired by Gentzen-style systems, with its root in requirement and software engineering. In the paper we present the main elements of the approach and report on its application in real-world projects.

Keywords: Dependability, Software Engineering, Assurance Case, Problem Oriented Engineering, Engineering Design

1 Introduction

By engineering design (shortly, design), we refer to the creative, iterative and often open-ended endevour of conceiving and developing products, systems and processes (adapted from [2]).

Engineering design by necessity includes the identification and clarification of requirements, the understanding and structuring of the context into which the engineered system will be deployed, the detailing of a design for a solution that can ensure satisfaction of the requirements in context, and the construction of arguments to assure the validating stake-holders that the solution will provide the functionality and qualities that are needed. The last of these is the concern of this paper.

Typically, for software at least, even though evidence is gathered during development the collation, documentation and quality injection of the assurance argument follows construction; perhaps this is because software development is currently sufficiently difficult without having to serve the needs of two masters: code *and* assurance. If software and assurance argument could be developed together, then developmental risk could be managed better–development errors that weaken an assurance argument could be found earlier in the process—as could developmental cost—by removing the compensating tendency to over-engineer.

Assurance-driven design (ADD), introduced in [1], does not make development any simpler; rather, it makes the building of an assurance argument a driver for development. Accepting this, however, ADD can guide the developer: by providing a more specific focus on those parts of a system that *require* assurance; by providing early feedback on design decisions; by capturing coverage of the design space; and, last but not least, by delivering an assurance argument alongside the product.

Our work on assurance-driven design is situated within Problem Oriented Engineering (POE), our framework for engineering design (instantiated for software in [3, 4]). The techniques we propose have no particular dependence on a software development context; indeed, our main example combines software and educational materials design and it is the assurance of their combined qualities that will drive our development.

The paper is structured as follows. Section 2 provides the briefest introduction to POE. In Section 3 we develop assurance-driven design, and in Section 4 illustrate its use through its application to a real-world problem. Section 5

*An expanded version of [1]

relates our work to that of others, and Section 6 reflects on what has been achieved and concludes the paper.

2 Problem Oriented Engineering

Problem Oriented Engineering is a framework for engineering design, similar in intent to Gentzen's Natural Deduction [5], presented as a sequent calculus. As such, POE supports rather than guides its user as to the particular sequence of design steps that will be used; the user choosing the sequence of steps that they deem most appropriate to the context of application. The basis of POE is the *problem* for representing *design problems* requiring designed solutions. *Problem transformations* transform problems into others in ways that preserve solutions (in a sense that will become clear). When we have managed to transform a problem to axioms[1] we have solved the problem, and we will have a designed solution for our efforts. A comprehensive presentation of POE is beyond he scope of this paper and can be found in [3, 4].

2.1 Problem

A problem has three descriptive elements: that of an existing real-world problem context, W; that of a requirement, R; and that of a solution, S. We write the problem with elements W, S and R as $W, S \vdash R$. What is known of a problem element is captured in its description; descriptions can be written in any appropriate language: examples include natural language, Alloy ([6]), and machine language. Solving a problem is finding S that satisfies R in the context of W.

Figure 1 gives an example of engineering design problem (shortly problem), described in a Problem-Frame-like notation ([7]). The problem (from a real world case study [8, 9]) is that of defining a controller to release decoy flare from a military aircraft: essentially decoy flares provide defence against incoming missile attack. The context includes a Pilot, a Defence system and some other existing hardware, represented in the figure as named undecorated rectangles. The solution to be designed is Decoy Controller, represented as a named decorated rectangle. The arc annotations are shared phenomena: for instance, the Pilot can send an ok command to the Decoy Controller. The solution needs to satisfy the Safe decoy control requirement, represented as a named dotted ellipse, for the safe release of decoys. Formally, in POE, this problem is represented as:

$$\text{Defence System}^{con}, \text{Dispenser Unit}^{out}_{fire,sel}, \text{Aircraft Status System}^{air}, \text{Pilot}^{ok}, \text{Decoy Controller}^{fire,sel}_{con,out,air,ok} \vdash \text{SDC}^{fire,sel}_{con,out,air,ok}$$

but we use both notations interchangeably.

[1] An *axiomatic problem* is a problem whose adequate, i.e., fit-for-purpose, solution is already known.

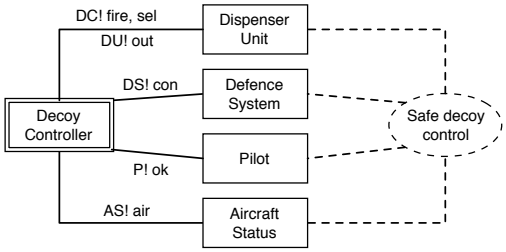

Figure 1. The Decoy Controller Problem

2.2 Problem transformations and justification obligations

Problem transformations capture discrete steps in the problem solving process. Many classes of transformation are recognised in POE, reflecting a variety of 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 (whose notation is based on that of [5]). Suppose we have *conclusion* problem $P : W, S \vdash R$, *premise* problems $P_i : W_i, S_i \vdash R_i$, $i = 1, ..., n$, $(n \geq 0)$ and *justification* J, then we will write:

$$\frac{P_1 : W_1, S_1 \vdash R_1 \quad ... \quad P_n : W_n, S_n \vdash R_n}{P : W, S \vdash R}\ \begin{matrix}[\text{NAME}] \\ \langle\langle J \rangle\rangle\end{matrix}$$

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* $(AA_1 \wedge ... \wedge AA_n) \wedge J$ 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 $AA_1, ..., AA_n$, respectively.

Engineering design under POE proceeds in a step-wise manner with the application of problem transformation schemata, examples of which appear below: the initial problem forms the root of a *development tree* with transformations applied to extend the tree upwards towards its leaves. A problem is solved for a stake-holder S if the development tree is complete, and the adequacy argument constructed for that tree convinces S that the solution is adequate. For technical reasons[2], we write

$$\overline{P}$$

to indicate that problem $P = W, S \vdash R$ is solved. As P will be fully detailed in determining the solution — to the satisfaction of stake-holders — the indication that P is solved is without justification. For the technical details, see [4].

A partial development tree is shown in Figure 2.

[2] Simply, that we may indicate an axiom in a Gentzen system thus.

$$\dfrac{\dfrac{\overline{\mathsf{P}_3 : \mathsf{W}_3, \mathsf{S}_3 \vdash \mathsf{R}_3} \qquad \mathsf{P}_4 : \mathsf{W}_4, \mathsf{S}_4 \vdash \mathsf{R}_4}{\mathsf{P}_2 : \mathsf{W}_2, \mathsf{S}_2 \vdash \mathsf{R}_2}\,{\scriptstyle [\mathsf{N}_2]}\ {\scriptstyle \langle\langle \mathsf{J}_2 \rangle\rangle}}{\mathsf{P}_1 : \mathsf{W}_1, \mathsf{S}_2 \vdash \mathsf{R}_1}\,{\scriptstyle [\mathsf{N}_1]}\ {\scriptstyle \langle\langle \mathsf{J}_1 \rangle\rangle}$$

Figure 2. A POE partial development tree

The figure contains four nodes, one for each of the problems P_1, P_2, P_3 and P_4. The problem transformation that gave the problem solver P_1 is justified by J_1, whereas the branching to problems P_2 and P_3 is justified by J_2. From the tree, we see that P_3 is solved. P_4 remains unsolved, so that the adequacy argument for the tree is incomplete; from the definition above, the incomplete adequacy argument is:

$$\mathsf{J}_2 \wedge \mathsf{J}_1$$

2.2.1 "Have we done enough?"

At any point in a development we can ask if we have done enough, i.e., if we were to declare our development complete would we be able to satisfy the validating stake-holders? This question is most obviously asked of the complete development, in which case an affirmative answer convinces all stake-holders that we have an adequate solution to the whole problem.

As previously mentioned, a completed development in POE is represented as a complete development tree, i.e., a tree in which no problems exist to be solved. The development is successful if the adequacy argument, AA, satisfies the stake-holders of the adequacy of the solution. For any stake-holder S, then, we have done enough if

$$\mathsf{AA} \text{ convinces } \mathsf{S}.$$

Consider again the form of the adequacy argument given a partial tree, such as that in Figure 2. Suppose that S is a stake-holder for problem P_1. Should P_1 be solvable, we would wish to find justification J_3 and solved problem P_5, say, such that

$$\mathsf{J}_3 \wedge \mathsf{J}_2 \wedge \mathsf{J}_1 \text{ convinces } \mathsf{S} \qquad \text{and} \qquad \dfrac{\overline{\mathsf{P}_5}\,{\scriptstyle [\mathsf{N}_3]}}{\mathsf{P}_4}\ {\scriptstyle \langle\langle \mathsf{J}_3 \rangle\rangle}$$

If we were free to choose P_5 without any reference to the requirements of the argument that establishes it as fit-for-purpose (that formed when J_3 is added to the adequacy argument) it would be unlikely to result in something that could be justified. Of the techniques mentioned in the introduction to this paper, the 'posit' of 'posit and prove' is moving towards this 'free' choice; moreover, over-engineering a solution simply allows the engineer a freer choice.

As we begin to balance the choice of P_5 and J_3, we move towards the position of assurance-driven design, in which the requirements for justification motivate the design. The techniques we introduce in this paper allow us to structure the development so that this balance can occur. Primary amongst them is the construction of projections from the overall development tree into, what might be called, 'stake-holder spaces' in which validation takes place.

2.2.2 A formal backwater: the weakest pre-justification

Although it is — currently — only of theoretical interest, by inspection, there is a best such justification that, given an incomplete development tree, completes the adequacy argument so as to just satisfy the stake-holder, S. By analogy to other formal systems, we term this the *weakest pre-justification*, $\mathsf{J}_{\mathsf{wpj}}$, such that, if IAA is the current (incomplete) adequacy argument for that tree, then for any K

$$(\mathsf{K} \wedge \mathsf{IAA} \text{ convinces } \mathsf{S}) \Rightarrow (\mathsf{K} \Rightarrow \mathsf{J}_{\mathsf{wpj}})$$

3 Assurance-driven design

A metaphor for engineering design under POE is that one grows a forest of trees. Each tree in the forest grows from a root problem through problem transformations that generate problems like branches; with happy resonance, the tree's *stake*-holders guide the growth of the tree. Some trees, those that have root problems that are validatably solvable for its stake-holders will grow until they end with solved problem leaves.

There are many reasons why the forest has many trees: described elsewhere [10], but only of note in this paper, is the preservation of a record of unsuccessful design steps, i.e., design steps that are not validatable for the current stake-holders, which cause a development to backtrack to a point where a different approach can be taken. The backtracked sub-trees are kept as record of unsuccessful development strategies[3].

For this paper, we note simply that development trees grow through the developer's careful choice of effective design steps. To produce an effective design step, the developer must consider both the problem(s) that the step will produce towards solution *and* what is the justification obligation that will satisfy the *validating stake-holders*. With the discharged justification obligations forming the basis of the adequacy argument, the result of a sequence of effective design steps is a solution *together with its assurance*

[3] Backtracked trees are not 'deadwood'; rather they stand as proof of design space exploration, with their structure being reusable for, for instance, other stake-holders' problems. Unsolved problems that remain in backtracked trees do not affect the completed status of a development.

argument[4]. We have observed the interplay of design steps and their justification under POE (for instance, [11]), and have developed a simple, composable process pattern—the POE Process Pattern—that guides their effective interleaving. The structuring of the problem solving activity through the POE process pattern is the basis of assurance-driven design.

We note that a problem transformation schema is applied to a conclusion problem, and that the development tree is extended up by the application. There is no necessity for any premise problem to be determined before the justification is added. Indeed, one could see the problem solver saying "It is fashionable to have a fan oven in stainless steel", and then searching for an oven that fits the bill[5].

It is determining the needs for the justification, rather than for the premise problem(s), that motivates us to introduce assurance-driven design: assurance-driven design determines the justification first, and then looks for the corresponding premise problem.

3.1 The POE process pattern

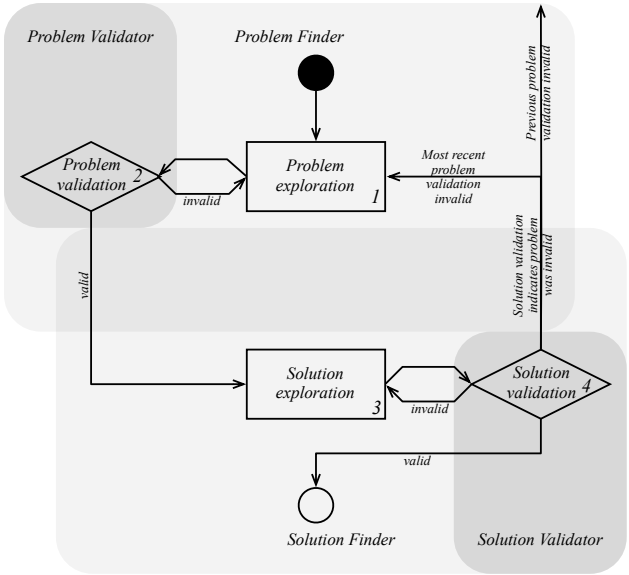

Figure 3. The POE Process Pattern for assurance-driven design

The POE process pattern shown in Figure 3 is described in a variant of the UML activity diagram notation [12]: rectangles are resource consuming activities; diamonds indicate choice points; the flow of information is indicated by arrows; the scope of the various roles is indicated by shading, overlapping shading indicating potential need for communication between roles. Referring to the numbers in the figure: first explore the problem better to understand it (*1*), checking that understanding through problem validation (*2*), iterating problem exploration as necessary; then explore the solution better to understand its design (*3*), checking that understanding through solution validation (*4*), iterating solution exploration as necessary.

The role of a *problem finder* during problem exploration is to explore their understanding of the problem (or part thereof), perhaps with the help of others. The goal of problem exploration is to produce descriptions of the problem that will satisfy the problem-validators(s) at problem validation. Similarly, the role of *solution finder* during solution exploration is to explore their understanding the solution (or part thereof) to the problem, again perhaps with the help of others. The goal of solution exploration is to produce descriptions of the solution that will satisfy the solution validator(s) at solution validation.

The role of a *problem validator* is to validate a candidate problem description. There are many familiar examples of problem validator. These include, but are not limited to:

- the customer or client — those that pay for a product or service;
- the regulator — those whose remit is the certification of safety of a safety of a safety-critical product, for instance;
- the business analyst — whose role is to determine whether the problem lies within the development organisation's business expertise envelope;
- the end-user — those who will use the product or service when commissioned.

It is a problem validator's role to answer the question "Is this (partial) problem description valid?" Depending on a problem validator's answer, the Problem Finder will need to re-explore the problem (when the answer is "No!"), or task the Solution Finder to find a (partial) solution (when the answer is "Yes!").

The role of the *solution validator(s)* is to validate a (candidate or partial) solution description, such as a candidate architecture (a partial solution) or choice of component (something of complete functionality). Although present in every commercial development, the roles of solution validator may be less familiar to the reader. They include, but are not limited to:

- a technical analyst — whose role is to determine whether a proffered solution is within the development organisation's technology expertise envelope;

[4]If there are no validating stake-holders for a development, the justification obligations can be ignored.

[5]Of course, we could have written such a statement as part of the requirement, but that would have been the stake-holder's statement, not the problem solver's.

- an oracle — who determines, for instance, which of a number of features should be included in the next release;

- a unit, module, or system tester; a project manager— who needs to timebox particular activities.

It is the solution validator's role to answer the question "Is this (candidate or partial) solution description valid?" Depending on their response, the problem solver may need to re-explore the solution (when the answer is "No!"), move back to exploring this or a previous problem (when the answer is "No, but it throws new light on the problem!"), or moving on to the next problem stage (when the answer is "Yes!").

The potential for looping in the POE process pattern concerns unsuccessful attempts to validate, and is indicated by arrows labelled *invalid* in the figure. Those leading back to exploration activities, of which there are two, continue their respective exploration activities in the obvious way. The other two invalid arrows lead from a failed solution validation to restart a problem exploration when the indication is that it was wrong. Examples of this latter form of failure are well known in the literature. For instance, Don Firesmith, in an upcoming book [13], talks about the need for architecture *re*-engineering in the light of inadequately specified quality requirements [part of an earlier problem exploration]:

> [...] it is often not until late in the project that the stakeholders recognize that the achieved levels of quality are inadequate. By then [...] the architecture has been essentially completed [solution exploration], and the system design and implementation has been based on the inadequate architecture.

In this way, recognising late that inadequately specified quality requirements (as discovered through problem exploration and validated at problem validation) have not been met can be very difficult and expensive to fix; leading to revisiting a long past problem, that of re-establishing the architecturally significant quality requirements[6].

Although we do not consider developmental risk explicitly in this paper, we note that feedback within the process has an impact on resources: an unsuccessful validation indicates that some previous exploration was invalid, to a greater or lesser extent. Moreover, some proportion of the development resource that will have expended during and subsequent to that exploration — the impact of the failed validation — will have been lost[7].

[6]Firesmith cites Boeing's selection of the Pratt and Whitneys PW2037 Engine for the Boeing 757 [14] as an instance of this problem.

[7]Work on risk management in POE is in preparation at the time of writing.

After successful problem validation, handover between the problem and solution finders occurs. In problem and solution finder are the same person, this raises no issues. Otherwise, it is possible to consider the solution finder as a problem validator, so that they receive a description of the problem that they have validated as the basis of their solution exploration. Symmetry dictates that the problem finder should have a role in solution finding too.

3.1.1 Building potent design processes

Although the POE process pattern provides a structure for problem solving, in its raw form, a problem will only be solved (i.e., the end state in Figure 3 is reached) when, after iteration, a validated problem is provided with a validated solution. This 'bang-bang' approach is suitable for simple problems, but is unlikely to form the basis of any realistically complex problem encountered in software engineering.

To add the necessary complexity, the POE process pattern combines with itself in three basic ways; in combination, it is again a process that can be combined. The three ways it can be combined are in sequence, in parallel and in a fractal-like manner, as suggested in Figure 4, and as described in the sequel.

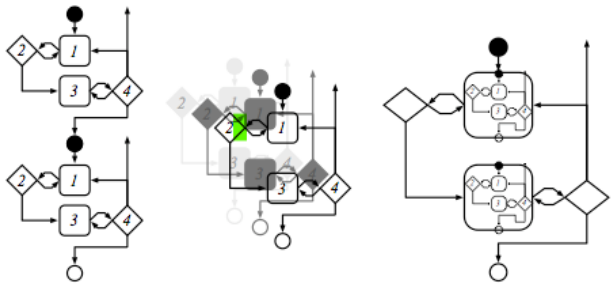

Figure 4. (a) Sequential, (b) Parallel and (c) Fractal-like combination

Sequential Design By identifying the end of one complete problem solving cycle with the start of another (see Figure 4(a)), we move a partially solved problem to the next phase: using the validated solution to explore the problem further. In [4], we show how a partial solution in the form of an architecture can lead to more detailed problem exploration: in that paper, we use the Model-View-Controller architecture to structure the solution of a problem, simplifying the problem to one of defining first the Model, then the View and finally the Controller.

In sequence, the POE process pattern models (more or less traditional) design processes in which architectures are

used as structure in the solution space according to architecturally significant requirement and qualities, and according to developmental requirements.

Parallel Design By identifying many instances of the POE process pattern through the start state, many problem solvers can solve problems in parallel. Architectures that admit such concurrent problem solving, and that might be discovered in a sequential prelude to such a process, are evident in many areas. One of timely relevance, given their current popularity, is open source projects, such as the GNU Classpath project whose goal is to provide

> 'a 100% free, clean room implementation of the standard class libraries for compilers and runtime environments for the java programming language.'

Concurrent development may place demands on the resources shared throughout the concurrent design. For instance, during problem and solution validation should access to the various stake-holders be co-ordinated, or should individual problem and solution finders be allowed access to them as and when necessary?

Communications between those involved in parallel development is an issue on the GNU Classpath project, and it is not surprising that explicit guidance exists to i) partition work through a task list and a mailing list, ii) contact the central maintainer of the project when the developer wishes to make certain non-trivial additions to the project, iii) global announcements whenever important bugs are fixed or when 'nice new functionality' is introduced.

Fractal-like Design Fractal structures are self-similar in the sense that the whole structure resembles the parts it is made of [15]. Another way to look at it is that the whole is generated from simple building blocks, with complexity emerging through recursion of the simple generators. By analogy, problem solving under the POE process pattern is structurally simple and admits recursive application in that problem solving activity can occur in the Problem Exploration and Solution Exploration parts of the POE process pattern. In the next section, we show how this leads to our notion of assurance-driven design.

3.1.2 The 'fractal' nature of validation

Given that problem and solution exploration can both be instances of the POE process pattern, let us consider the problems and solutions they work with.

As Problem Exploration leads to Problem Validation, it is 'complete' when we have delivered a problem description that satisfies the problem validator. That is, Problem Exploration is complete when we have found a

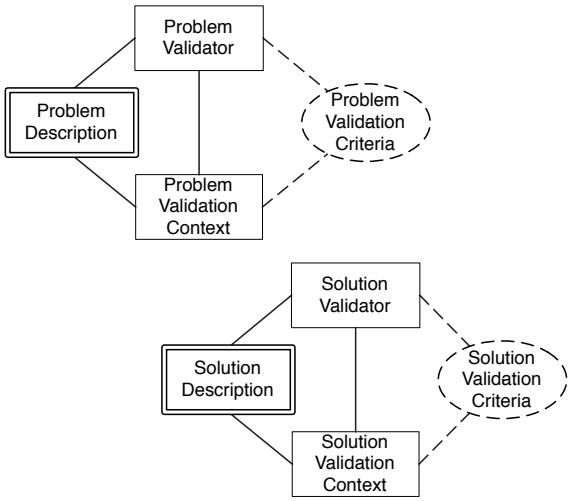

Figure 5. (a) Problem exploration as a problem validation problem, and (b) Solution exploration as a solution validation problem.

Problem Description that solves the following *problem validation problem*, illustrated in Figure 5(a):

Problem Validation Context, Problem Validator,
Problem Description ⊢ Problem Validation Requirements

The Problem Validation Context (PVC) is a description of the context in which the validation of the problem will be undertaken, and will need to be found as part of the fractal problem exploration phase of the outer problem exploration, as will the Problem Validation Requirements (PVR), i.e., the requirements that will need to be met for the problem to be validated. Note that the Problem Validator (PV) is an explicit domain in the context.

Symmetrically, solution exploration can be seen as complete when we have found a Solution Description that, when considered in the Solution Validation Context (SVC), satisfies the Solution Validation Requirements (SVR) of the Solution Validator (SV). As a POE problem, this is the *solution validation problem*, illustrated in Figure 5(b):

Solution Validation Context, Solution Validator,
Solution Description ⊢ Solution Validation Requirements

Although the fractal-like nature makes an easy clarity somewhat difficult, the view we have just presented fits well with practice. Indeed, discussions that lead to an agreed (i.e., validated) collection of use-cases [16] can be seen as a technique for producing a problem description that satisfies the problem validation problem. Moreover, discussions that lead to an agreed collection of acceptance tests can be seen as a solution description that satisfies the solution validation

problem. Requirements engineering, consisting of elicitation, analysis, specification can be seen as a technique for partial problem exploration; pattern oriented analysis and design is a technique for partial solution exploration.

In terms of Section 2.2, each validation is a projection of a whole development tree's adequacy argument into the stake-holder space determined by the validation context and validation requirements and, for a properly engineered solution, considering each of the adequacy problems is important.

4 Assurance-driven design in practice

The companion paper, [1], presented the assurance-driven design of a safety-critical subsystem of an aircraft. In this paper, we present a very different problem, that of the assurance-driven design of a research programme for The Open University. Whereas the aircraft example involved just a single stake-holder — the regulator for the system — this paper's example involved over 50 stake-holders as problem and solution validators. The project manager for the programme is the second author. For more information about that project, and a discussion of how POE was adopted in practice, please refer to [17, 18].

4.1 Notation

Because of the needs of the problem, we have augmented the traditional Gentzen-style notation to support better the separation of the problem and solution explorations, and to link validation problems to the justification of which they form a part. Figure 6 illustrates the differences. In the figure, we see the traditional transformations involving the problems labelled 'design problem' that will be familiar from Section 2.2. The triangular structures that extend the horizontal bar indicates the collection of validation problems associated with the step: by convention, when written on the right they are problem validation problem, when on the left they are solution validation problem[8].

4.2 Example

The Computing Department at The Open University is in the process of developing a new part-time MPhil programme to be delivered at a distance, supported by a blend of synchronous and asynchronous internet and web technologies — the *eMPhil*. The eMPhil is innovative in many ways in its adoption and use of emergent technology, like Second Life and Moodle, to support the core processes of the programme (the interested reader is directed to [19] for details).

[8]Because of the separation of problem and solution validations, never will problem and solution validations need to appear in the same step.

The eMPhil project team was faced with a complex socio-technical problem, that of the adoption and development of appropriate software systems and the definition of new processes and practices, of the design and delivery of induction and training activities for staff and students, and the institution of a framework for quality assurance, monitoring and continuous process improvement. The project also found itself with many stake-holder groups, those who would play problem validators, such as the Head of the Department of Computing, and solution validators, including Head of the Research Degrees Committee and Pro-Vice Chancellor for Research and Enterprise. The difficulties of managing the design and validation of the programme partially motivated the development of and application of the techniques described in this paper.

4.2.1 The problem

The eMPhil was required to meet a number of objectives:

- for the Head of the Department of Computing: to enhance and develop the department's provision to its graduate community; to increase the overall amount of research supervision that takes place within the department;
- for at-a-distance students: to make available technology for their support; to provide as a forum for that student community; to allow those unable to commit time for a PhD a research degree to study for;
- for academics wishing to promote research in their area: to create cohorts of research students on specific research themes and projects;
- for the Head of the Research Degrees Committee and Pro-Vice Chancellor Research and Enterprise: to support the development of research skills; to comply with university policy on research student induction and training; to comply with national standards [20].

The eMPhil core project team — the problem and solution finders — was composed of four academics, with the second author as project leader. The POE process pattern was used as described below to shape the project, with the techniques described earlier in the paper used to drive and manage its development and risks, as well as to identify the eMPhil project's needs for resource and communication.

4.2.2 The process

Figure 6 illustrates the early design steps taken by the development team towards a solution to the problem. The first transformation (bottom of the figure) achieves a first characterization of the problem context and requirement (from

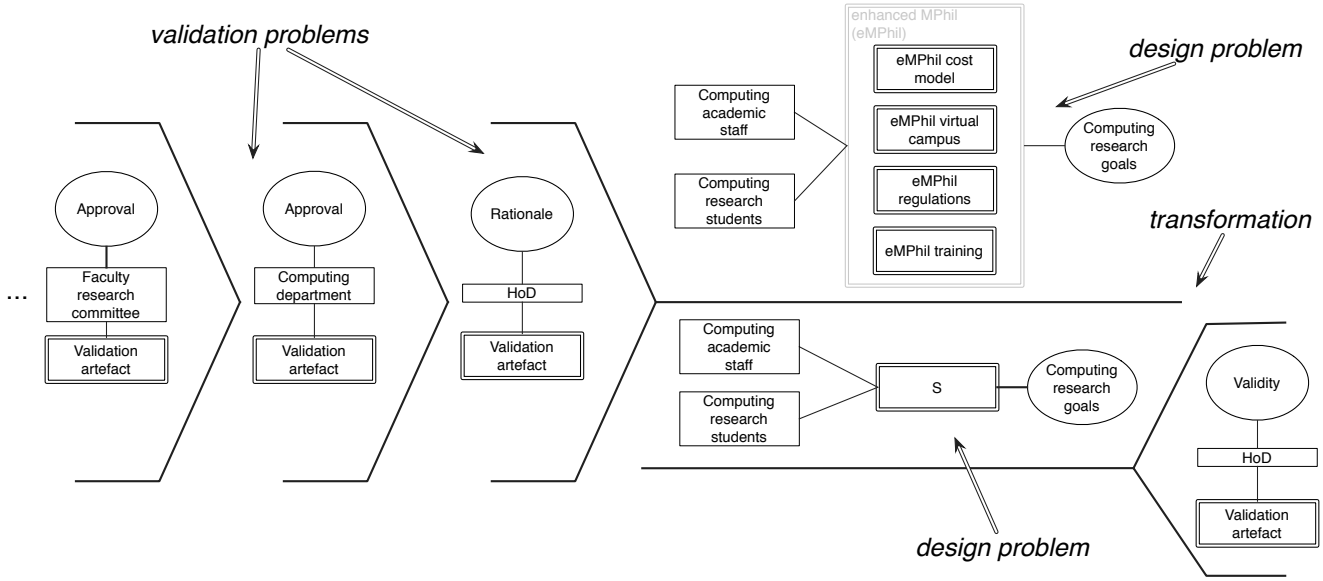

Figure 6. Early design steps

an empty conclusion problem—the start of all POE Design explorations), with a problem validator identified as the Head of the Computing Department (HoD). The HoD set the strategic goals which constituted the initial requirement description, and led to the inclusion of Computing academic staff and research students as a first approximation for the problem context. This initial problem exploration was coupled with the solution of the associated validation problem, consisting in making sure that the HoD's strategic intent was understood correctly by the problem solver.

The next transformation (top of Figure 6) captures an early solution exploration activity, in which a candidate solution architecture is starting to emerge, that of a new research degree, an MPhil, to be delivered in part-time mode at a distance. Note the validation problems to the left. The initial buy-in for the new degree was sought from the HoD, as the person in charge of releasing resources for the project, and with whom the rationale for the proposed solution was discussed. Approval from the HoD then triggered a comprehensive approval process throughout the organisation, reflecting its power structure (each validation problem concerns stake-holders at different management levels). Downside risks at these point were very high, with assurance taking precedence and greatly influencing the design.

Subsequent design alternated between further problem and solution explorations, and related validation, as illustrated in Figure 8, which provides a snapshot of the design tree after the first 10 months' development from the developer team perspective, assuming both problem and solution finder roles. Transformations labelled A and B at the bottom of the figure correspond to the early steps we have just described. From the initial solution architecture, a number of sub-problems were then identified (transformation C) each addressing complementary aspects of the solution, such as the design of its technological infrastructure, a related cost model, a programme of user induction and training, a system of monitoring and evaluation, etc. Each sub-problem was then taken forward through further transformations and related stake-holder validation, with the design problems at the top representing either solved sub-problems or open problems in the process of being addressed.

Note that some of the steps introduce sub-problems, which lead to branches in the design tree. This happens when a number of solution components have been identified, each to be designed, together with their architectural relations and mutual constraints. The POE transformation which generates them, called *solution expansion*, generates appropriate sub-problems for each to-be-designed component, based on such architectural knowledge. Each sub-problem then becomes the root of a (sub-) design tree.

4.2.3 Fractal Problem Validation

As introduced in Section 3, validation problems are problems too, and so their solution can be arrived at through a problem solving process, and hence (should they have solution) solvable in our POE framework, i.e., they should be treated as any other problem, with problem finder exploring the problem, obtaining problem validation, and so on. In the augmented notation, the obvious place for the validation problem development is in the extension to the horizontal bar; see Figure 7. However, such diagrams quickly become unwieldy, and a more pragmatic approach was nec-

essary in which the validation problem development was placed in a separate file, with indicators (again, Figure 7, on the left) for the state of each validation problem and hyperlinks used for easy access to the embedded validation problem development. It became apparent that the indicators formed a useful proxy for developmental risk associated with an unsolved validation problem, that risk being associated with the progress made in the solution of the main problem as opposed to the validation problem. We used a simple semaphore system for the risk indicators. Given the lack of tool support, this was deemed a simple, but useful tool from a project management perspective; of course, a more accurate estimation of risk would have required more sophisticated tools. Figure 7 gives an intuition of the meaning of the risk indicator: to the right is the equivalent fully expanded validation problem.

4.3 Early evaluation

The experience on the project so far has been very encouraging, and has clearly indicated that the conceptual tools offered by POE, including assurance-driven design were able to cater for all relevant aspects of the project. Design forests provide a powerful summary of the development, with all critical decision points clearly exposed, and all sub-problems (solved and unresolved) and their relation clearly identified. The risk indicators, despite their lack of sophistication, were considered very useful in signposting critical parts of the development. The notation was also considered an effective communication tool: its relative simplicity and abstraction allowed even non technical stake-holders, like senior managers in academic and academic-related units, to grasp the essence of the project with very little explanation required. The inclusion of validation problems within the development tree, with the explicit acknowledgement of all relevant stake-holders, was also considered a valuable tool to gauge the criticality of each design step, as well as to focus attention on the aspects of the problem of significance to each stake-holder. For instance, the high criticality of initial approval process is evidenced by the large set of validation problems in the early stages of development, in which the validation effort largely outweighed the effort to produce an initial outline for the solution, but greatly reduced the risk of the programme not to be deemed viable by management later on.

5 Related Work

Work on assurance cases is found in the area of dependability, from which two main structured notations for expressing safety cases have emerged. One is the goal-structuring notation (GSN) [21], 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 [22], which is based on Toulmins work on argumentation 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 [23] have also suggested ways in which product and process evidence could be combined in GSN assurance cases. One of the difficulties of these approaches is that they were not conceived to provide an integrated approach to safety development and, by and large, use artifacts and processes which may parallel but not integrate with software development. Instead, a main aim of our work is to allow for the efficient co-design of both software and assurance case based on artefacts and processes which are common to both. Some very recent work by Strunk and Knight [24] 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 [7] and GSN. Although this work shares some of our goals, it is still rather preliminary for a meaningful comparison with POE.

A more mature process model, which shares something with POE, is the CHOAS model and lifecycle of Raccoon [25]. In this model fractal invocations of problem solving processes are combined to provide a rich model of software development, which is then used as the basis for a critical review of software engineering, of its processes and its practices. Raccoon's review leads to the conclusion that neither separately nor together do top-down or bottom-up developments tell the whole story; hence, a 'middle out theory' is proposed, based on the work that developers do to link high level project issues to, essentially, code structures. It is an attractive theory, and we wish to explore the ways in which fractal invocation in POE and assurance-driven design satisfy the criteria laid down for it.

6 Discussion and conclusion

The POE notion of problem suggests 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 the assumptions upon which they are based to scrutiny and validation. That all descriptions are generated through problem transformation forces the inclusion of an explicit justification that such assumptions are realistic

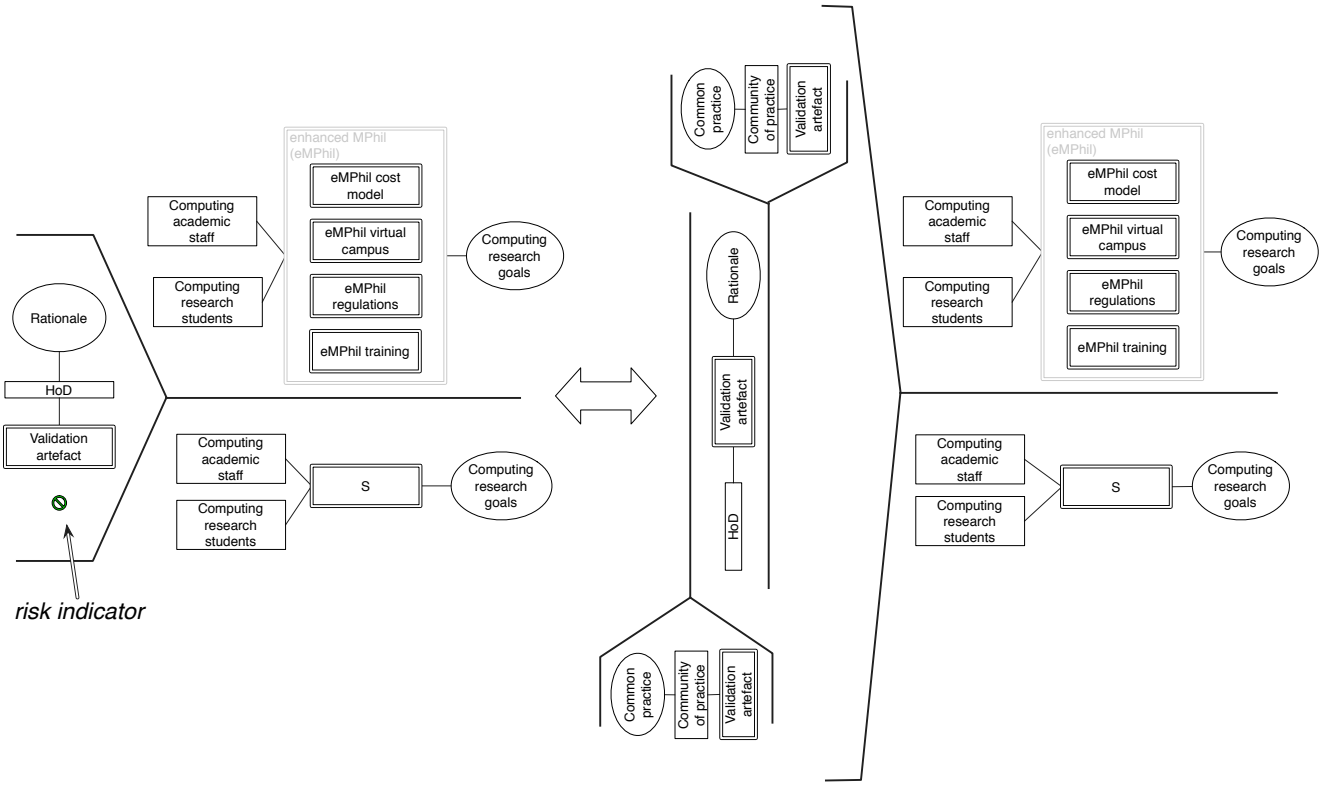

Figure 7. Risk indicator and its 'fractal' validation problem

and reasonable. In particular, requirements are justified as valid, are fully traceable with respect to the designed system (and *vice versa*), and evidence of their satisfaction is provided by the adequacy argument of a completed POE development tree.

We have shown (a) how (partial) problem and solution validation can be used to manage developmental risk and (b) how an assurance arguments can be constructed alongside the development of a product. Developmental risks arise from 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.

Although other approaches provide a focus on an assurance argument, the possibility of having the assurance argument *drive* development is an option that appears unique to ADD and POE.

Finally, POE defines a clear formal structure in which the various elements 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 POE signifies that all required forms of reasoning can be accommodated, from deductive to judgemental, within a single development.

Acknowledgments

We acknowledge the financial support of IBM, under the Eclipse Innovation Grants, and of SE Validation Limited. Our thanks go to Derek Mannering at General Dynamics UK, Lucy Hunt at Getronics plc, Jens Jorgensen and Simon Tjell of Aarhus University, Andrés Silva of the University of Madrid, Colin Brain of SE Validation Ltd, Anthony Finkelstein at UCL (who suggested the discussion of "Have we done enough?"), and John Knight at UVA. L. B. S. Raccoon has read all of our work and made truly insightful comments. Finally, thanks go to our many colleagues in the Computing Department at The Open University, particularly Michael Jackson.

References

[1] Jon G. Hall and Lucia Rapanotti. Assurance-driven design. In *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA 2008)*. Published

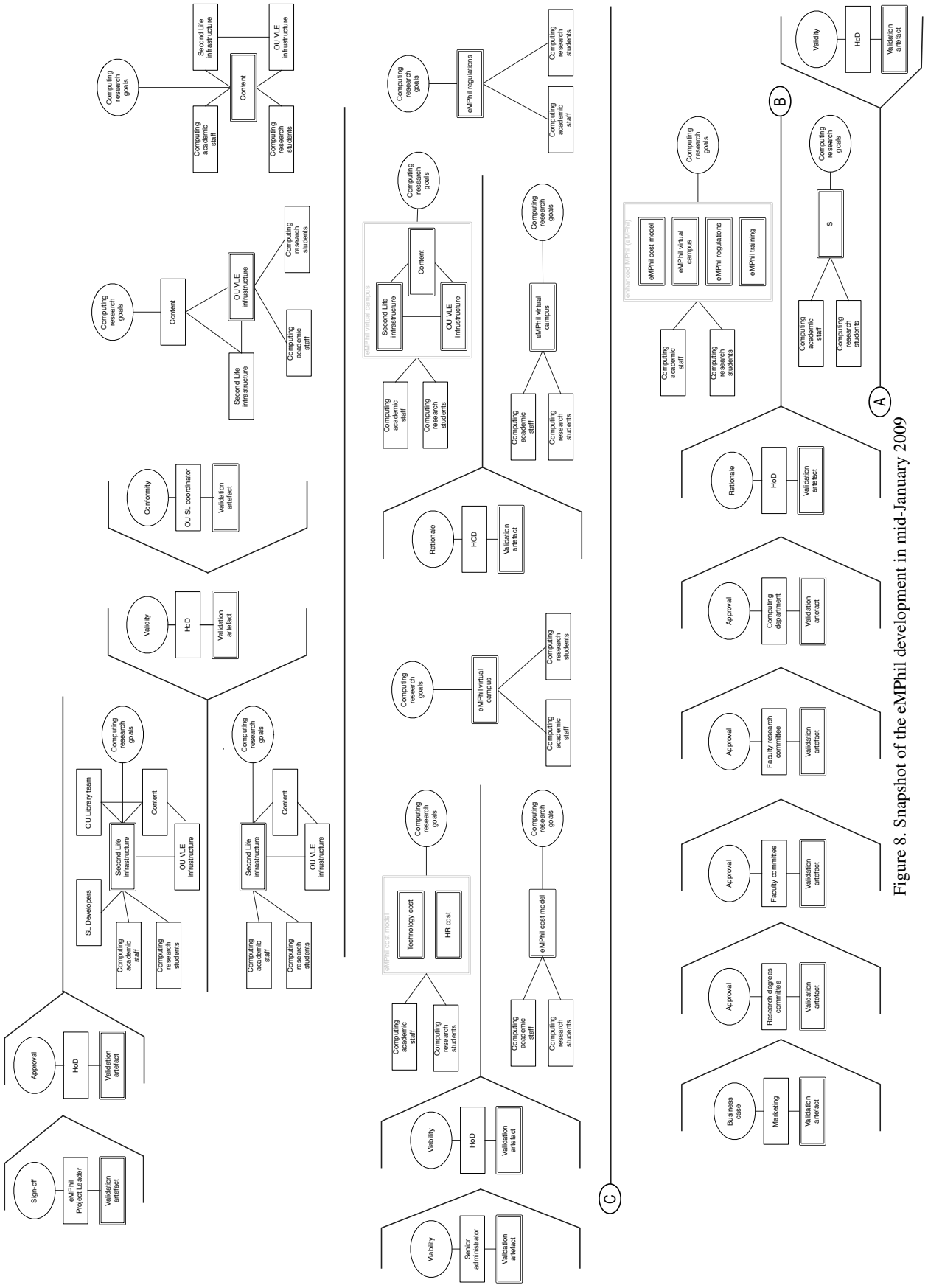

Figure 8. Snapshot of the eMPhil development in mid-January 2009

by the IEEE Computer Society, 2008. Also available as Open University Computing Department Technical Report #2007/15.

[2] Engineering Council of South Africa Standards and Procedures System Definition of Terms to Support the ECSA Standards and Procedures System.

[3] Jon G. Hall, Lucia Rapanotti, and Michael Jackson. Problem oriented software engineering: A design-theoretic framework for software engineering. In *Proceedings of 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 15–24. IEEE Computer Society Press, 2007. doi:10.1109/SEFM.2007.29.

[4] Jon G. Hall, Lucia Rapanotti, and Michael Jackson. Problem-oriented software engineering: solving the package router control problem. *IEEE Trans. Software Eng.*, 2008. doi:10.1109/TSE.2007.70769.

[5] M. E. Szabo, editor. *Gentzen, G.: The Collected Papers of Gerhard Gentzen*. Amsterdam, Netherlands: North-Holland, 1969.

[6] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, 2006.

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

[8] Derek Mannering, Jon G. Hall, and Lucia Rapanotti. Towards normal design for safety-critical systems. In M. B. Dwyer and A. Lopes, editors, *Proceedings of ETAPS Fundamental Approaches to Software Engineering (FASE) '07*, volume 4422 of *Lecture Notes in Computer Science*, pages 398–411. Springer Verlag Berlin Heidelberg, 2007.

[9] Jon G. Hall, Derek Mannering, and Lucia Rapanotti. Arguing safety with problem oriented software engineering. In *10th IEEE International Symposium on High Assurance System Engineering (HASE)*, Dallas, Texas, 2007.

[10] Jon G. Hall and Lucia Rapanotti. The discipline of natural design. In *Proceedings of the Design Research Society Conference 2008*. Design Research Society, 2008.

[11] Derek Mannering, Jon G. Hall, and Lucia Rapanotti. Safety process improvement with POSE & Alloy. In Francesca Saglietti and Norbert Oster, editors, *Computer Safety, Reliability and Security (SAFECOMP 2007)*, volume 4680 of *Lecture Notes in Computer Science*, pages 252–257, Nuremberg, Germany, September 2007. Springer-Verlag.

[12] OMG. Unified Modeling Language (UML), version 2.0. http://www.omg.org/technology/documents/formal/uml.htm. Last checked: May 2009.

[13] Donald Firesmith. *The Method Framework for Engineering System Architectures*. CRC Press, 2008.

[14] James P. Womack and Daniel T. Jones. *Lean Thinking – Banish Waste and Create Wealth in Your Corporation*. Simon and Schuster, 1996.

[15] Kenneth Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. Wiley-Blackwell, 2nd edition, 2003.

[16] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[17] Lucia Rapanotti and Jon G. Hall. Designing an online part-time master of philosophy with problem oriented engineering. In *Proceedings of the Fourth International Conference on Internet and Web Applications and Services*, Venice, Italy, May 24-28 2009. IEEE Press.

[18] Lucia Rapanotti and Jon G. Hall. Problem oriented engineering in action: experience from the frontline of postgraduate education. Technical Report TR2008/16, The Open University, 2008.

[19] Lucia Rapanotti, Leonor M. Barroca, Maria Vargas-Vera, and Shailey Minocha. deepthink: a second life campus for part-time research students at a distance. Technical Report TR2009/1, The Open University, 2009.

[20] UK GRAD, Joint Skills Statement of Skills Training Requirements. http://www.grad.ac.uk/jss/ Last checked: May 2009.

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

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

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

[24] Elisabeth A. Strunk and John C. Knight. The essential synthesis of problem frames and assurance cases. *Expert Systems*, 25(1):9–27, 2008.

[25] L. B. S. Raccoon. The Chaos model and the Chaos cycle. *SIGSOFT Softw. Eng. Notes*, 20(1):55–66, 1995.