This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Problem Oriented Software Engineering: solving the package router control problem

Jon G. Hall Lucia Rapanotti Michael A. Jackson
Centre for Research in Computing, The Open University
Walton Hall, Milton Keynes, MK7 6AA
{J.G.Hall,L.Rapanotti}@open.ac.uk, jacksonma@acm.org

Abstract— **Problem orientation is gaining interest as a way of approaching the development of software intensive systems and yet a significant example that explores its use is missing from the literature.**

In this paper, we present the basic elements of Problem Oriented Software Engineering (POSE) which aims to bring both non-formal and formal aspects of software development together in a single framework. We provide an example of a detailed and systematic POSE development of a software problem, that of designing the controller for a package router. The problem is drawn from the literature, but the analysis presented here is new. The aim of the example is twofold: to illustrate the main aspects of POSE and how it supports software engineering design, and to demonstrate how a non-trivial problem can be dealt with by the approach.

Index Terms— **Software Engineering, Design methodology, Problem orientation, Genzten-style sequent calculus**

I. Introduction

Software engineering includes the identification and clarification of system requirements, the understanding and structuring of the problem world, the structuring and specification of a hardware/software machine that can ensure satisfaction of the requirements in the problem world, and the construction of adequacy arguments, convincing both to developers and to customers, users and other interested parties, that the system will provide what is needed. These activities are much concerned with non-formal domains of reasoning: the physical and human world, requirements expressed in natural language, the capabilities of human users and operators, and the identification and resolution of apparent and real conflicts between different needs. In a software-intensive system these informal domains interact with the essentially formal hardware/software machine: an effective approach to system development must therefore deal adequately with the non-formal, the formal, and the relationships between them. As Turski has pointed out [1]:

> There are two fundamental difficulties involved in dealing with non-formal domains (also known as the real world):
>
> 1. Properties they enjoy are not necessarily expressible in any single linguistic system.
> 2. The notion of mathematical (logical) proof does not apply to them.

These difficulties, which are well known in the established branches of engineering, have sometimes led to a harmful dichotomy in approaches to software development: some approaches address only the formal concerns, usually in a single formal language; others address only the non-formal concerns, using several languages, which often cannot be reconciled. Problem Oriented Software Engineering (POSE [2]) aims to bring both non-formal and formal aspects of software development together in a single framework. POSE is intended to provide a structure within which the results of different development activities can be combined and reconciled. Essentially the structure is that of the progressive solution of a system development problem; it is also the structure of the adequacy argument that must eventually justify the developed system. POSE does not prescribe a development process, but rather identifies discrete development steps and their connections as may be needed within a chosen development process.

Whilst sharing much of its conceptual basis with Problem Frames [3], POSE both extends and generalises that approach 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 IV-C);
- 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.

One advantage of the extensions into the solution domain is that POSE supports iterative design processes—processes that span both problem and solution domains.

The paper presents the POSE framework, and argues its suitability for software engineering design through its application in solving a software-intensive system problem. Various elements of the POSE framework are illustrated in this process.

The paper is structured as follows: Section II introduces the conceptual basis of the POSE framework, including the notions of software problem, problem transformation and adequacy argument. Section III describes the relationship between POSE and some of the work of others. Section IV discusses a significant part of the problem solving process for the package router problem. A discussion, conclusions and future work are in Section V.

II. Problem Oriented Software Engineering

The Problem Oriented Software Engineering framework of [2] is akin to a Gentzen-style sequent calculus [4] for 'solving' software problems. The basis of a Genzten-style sequent calculus

Digital Object Indentifier 10.1109/TSE.2007.70769

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). Simplifying only slightly, POSE includes transformations that operate on software problem sequents to transform them to other sequents whilst preserving solutions. 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 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 adequacy 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.

A. 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*. [3]). 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 will affect;
- *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]:

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

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].

B. 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:

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

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

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

[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.

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 stepwise 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.

The problem transformation form presented above emphasises the formal relationship between the transformed problem elements and the resulting structure of the development tree. In this paper a more flexible description of a problem transformation is needed, so that it is easier to interleave development prose, descriptions and figures with the explanatory narrative. To do this we will describe what is, usually, the largest part of a transformation—the justification—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 justification of the application of transformation NAME to P. The body of the justification can have many components—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: issues arising from the step that will need to be considered and eventually discharged as part of the development. These may include known or suspected deficiencies in the justification, such as simplifications that have been used to assist early development.

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

Resulting problem(s): giving the problem(s) that are the result of the application of the rule, which become the basis of further development.

Any element may contain figures, *etc.*, that are needed or useful in describing the transformation.

C. 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). The schema also includes a *justification obligation* from which the justification in a problem transformation based on that schema stems. We will discuss several transformation schemata in Section IV during the exemplar that illustrates their use.

Problem transformation schema detail how a problem is transformed: pattern matched elements of the conclusion problem are repeated as appropriate to specialise the premise problem(s) and justification obligation.

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

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

The justification obligation is the condition that must be discharged for CONTEXT INTERPRETATION to be solution preserving; here the obligation is to argue why the new context, $\mathcal{W}'$, is preferred to the original, $\mathcal{W}$; the meaning of 'preferred' will, in general, be defined by developmental context, and there are many examples of the application of the schema and others below.

D. The adequacy argument

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.

One corollary is that the adequacy argument, and consequently the individual justifications, must be constructed with respect to all stake-holders' views of what adequacy might mean. A POSE development will, therefore, be influenced by the validation needs of the stake-holders. To an engineer, this will not be surprising—harder to satisfy stake-holders will require more closely reasoned development steps than those that are easier to satisfy. To a formalist, it may appear that, were it not for this stake-holder perspective, rules could omit the relative notion of justification to use an absolute correctness notion, such as proof, in its stead. However, many solutions are adequate for many stake-holders without being provably correct: one obvious example is that Microsoft Word proves adequate for document preparation in many organisational settings without being formally proven correct in that role (if even such a notion could be formulated). Many solutions could be missed by insisting on any absolute correctness notion.

III. RELATED WORK

There are four main areas to which POSE contributes: the use of formality in software engineering design; the use of non-formal description languages; the structuring of the early software engineering design life-cycle; transformational approaches to software engineering design. Many others have made notable contributions in these areas: in this section, we compare our approach with some already in the literature.

A. Formal specification and refinement

The late 70s and early 80s saw many approaches to software development focused on the transformation of software specifications into code using processes that work within the solution domain, some supported by automatic tools.

Feather [5] proposes an approach to the formal specification of closed systems based on a specification language named *Gist*. A closed system is self-contained: it has no interaction with anything which is outside the specification. As such a closed-system specification must include the system of which the software is a component as well as the environment in which the system operates.

There are some methodological similarities between that work and POSE: the notion of problem is closed-world in nature as we ignore any aspect of the world which is not part of the problem context. However, we make no assumption of formality of descriptions, nor that a single reasoning mechanism exists throughout. In POSE, formality is used to structure a development and its adequacy argument and to transform problems.

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 [6] and Back [7] and, more recently, the categorical refinements of Smith [8]. Many of these transformations, in addition, are partial in their application, e.g., the *weaken precondition* rule [6], used in the refinement of the specification $w : [pre, post]$ by the specification $w : [pre', post]$, is sound only when the proof obligation $pre \Rightarrow pre'$ can be discharged. Proof obligations serve a similar purpose to justification obligations in our framework—they guard transformation application. POSE differs in that we do not require formality in the descriptions that are transformed or in the discharge of justification obligations.

Unlike POSE, Model-Driven Development (MDD) (see, for instance, [9])—ostensibly a transformational approach—assumes the existence of a unified description language in which all models are expressed, and for which model transformations are (semi-)automated once model mappings are defined. From our perspective, MDD does not distinguish between S and W: the underlying assumption is that a domain model can be transformed subsequently into design and implementation models based on a set of rules specified in the model mappings. The conditions under which this assumption actually holds are not clear at this point of MDD development, which is still rather preliminary.

B. The use of natural language in specification

Balzer *et al*. [10] describe the transformation of formal specifications from partial descriptions in a constrained natural language. In further work, Balzer [11] goes on to implementation via transformations starting from a specification. The paper [11] also observes the importance of documenting decisions during transformation. Similarly, [12] proposes an automated transformation system from high-level specifications into code, with manual intervention needed to guide the process.

From our perspective, this work is again firmly in the solution domain, i.e., concerned with the form of S. The scope of the transformations are also quite narrow: transforming an informal S into formal S, from which the generation of a programme can then be fully automated. In POSE, these are all solution transformations.

Similar to our approach, the work emphasises the use of informal descriptions, although with different motivation: informality is a useful way to start up what is otherwise a perfectly formalisable description; for us informality is a necessity as parts of W—the physical world in all its complexity—escape formalisation.

C. Goal-oriented Requirements Engineering

Goal-oriented Requirements Engineering [13] proposes an approach to early software development specifically for dealing with the requirements of a system. The underlying principle is that requirements can be expressed as goals—specific objectives that a system must meet. Such goals initially may be very abstract, whence the aim of a goal-oriented analysis is to refine them to operational goals that can be assigned as responsibilities to be discharged by software and/or other agents. Goal refinement takes the form of a tree, whose root is a high level system goal and whose leaves are operational goals. Such operational goals constitute (or are closely related to) a requirements specification for the software. Many of these characteristics are shared by POSE.

Recent effort has aimed at extending at least one variant of goal-orientation, KAOS [14], beyond early requirements with a view to integrate it within later stages of software development. For instance, [15] associates UML models to the goal refinement tree in order to provide structural specification descriptions; also, function specifications are associated to goals to make them operational; in [16] high level architectural design is also derived from the tree structure by allocating architectural software components to the various software agents to whom goal responsibilities are assigned to in the goal tree. In this extension, quality goals are stated and used to evaluate the resulting architectural design.

From our perspective, goal-orientation gives the requirement, R, the predominant role: goals are a form of requirement, and goal refinement is a form of requirement transformation that generates the set of sub-goals of a goal. Contextual information, W, is not explicitly considered; instead, it is incrementally added to a goal tree development in the form of partial views of the agents which become responsible for discharging goals; no transformation of the context is defined (and without its explicit inclusion, it is difficult to see how it could be). The solution, S, is derived from the subset of leaf goals that are assigned to software agents. Because of this, no transformation of S is defined either. In goal-oriented descriptions, the distinction between indicative and optative is missing, making it unclear from the model which agents are the object of design and which are provided ready-made by the context. Consequently it is not clear how adequacy arguments should be constructed.

D. Adequacy and assurance

Our notion of adequacy argumentation shares some of its motivation with work on assurance for critical systems. In the safety-critical context, for instance, a safety case is seen as a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment [17]. More than formality, what is important in a safety case is that it should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in its deployment context [18]. Kelly and Weaver observe [18] that, due to the nature of the evidence in assurance cases, a provably valid and sound case is unobtainable.

It is not surprising then, that the most widely spread approaches to assurance are far from formal. For instance, the goal-structuring notation (GSN) [19] is a graphical argumentation notation which allows the representation of elements of a safety argument and their interrelation. Similarly, Adelard's Claim-Argument-Evidence approach [17] is based on Toulmin's work

on argumentation [20] and includes: *claims* (same as Toulmin's *claims*), *argument* (combination of Toulmin's *warrant* and *backing*), and *evidence* (same as Toulmin's *grounds*).

Recent changes in safety-critical standards have made it much more desirable to build a safety (a form of adequacy) argument during development. Strunk and Knight [21] have recently proposed Assurance Based Development (ABD) in which a safety-critical system and its assurance case are developed in parallel, an approach that is akin to what might be achieved with POSE. Strunk and Knight have developed detailed examples of the use of their techniques in, as yet, unpublished work.

Adequacy and assurance in POSE include the notion of concern, broadly that considered by Jackson (for instance, [3]): by a concern we indicate a matter of interest or importance to some stake-holder. In this sense, the meaning is similar to that ascribed by the aspect-oriented community (for instance, [22]); however, for us, concerns do not form a necessary component of a development; one could imagine a development without concerns arising.

IV. The Exemplar

We consider the following problem from [3], in turn adapted from [10]. A package router (see the schematic of Figure 1) is used to sort packages according to bar-coded destination labels affixed to the packages. Packages slide under gravity through a tree of pipes and binary switches into bins that correspond to regional areas.

The problem with which we are concerned is the design of a software controller to ensure that:

- packages are routed appropriately, with misroutes reported; and
- the *Operator*'s commands to start and stop the conveyor are obeyed.

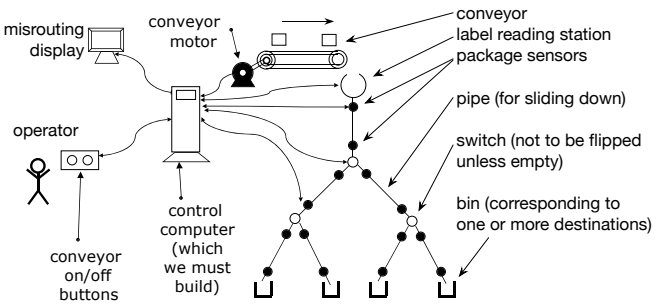
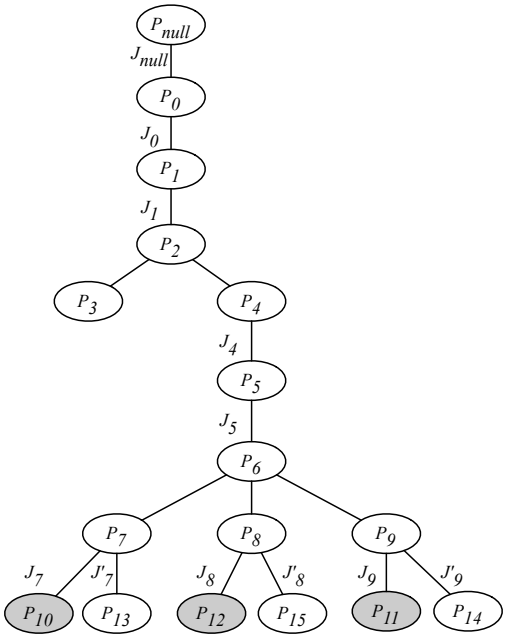


Fig. 1. Schematic of the problem (based on [3]). The switch network is a complete binary tree of height n

Figure 2 is a graphical summary of the whole of the POSE development that follows, in which we emphasise the relationships between the named problems (nodes) with justifications labelling arcs. Each problem and justification is discussed in detail as we work through the exemplar in this section.

A. The starting point for problem solving

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



Problem	Description
P_{null}	The *null* problem
P_0	Context expansion
P_1	Interpretation of P_0
P_2	Intepretation of P_1 as a 2-separable problem
P_3	*Conveyor Controller* sub-problem
P_4	*R&R Controller* sub-problem
P_5	Progressed *R&R Controller* sub-problem
P_6	Introducing the *R&R Controller MVC* architecture
P_7	*M* sub-problem
P_8	*V* sub-problem
P_9	*C* sub-problem
P_{10}	*M* sub-problem, interpreted
P_{11}	*C* sub-problem, interpreted
P_{12}	Progressed *V* sub-problem
P_{13}	*M* sub-problem, re-interpreted
P_{14}	*C* sub-problem, re-interpreted
P_{15}	Progressed *V* sub-problem, interpreted

Fig. 2. A summary of the POSE development presented as an (inverted) tree. Nodes represent problems, arcs are transformations labelled by justifications

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

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.

B. Capturing knowledge

The *null* problem is of interest only from a theoretical viewpoint. Practically, a major part of early software development concerns the elicitation of real-world knowledge and the construction of the descriptions of the elements of the problem derived therefrom. Problem descriptions are captured in POSE through the various transformation schemata for *interpretation*, including that of CONTEXT INTERPRETATION introduced in Section II-C. In what follows we begin with a number of interpretations in

order to provide a POSE characterisation of the Package Router problem.

1) Structuring the problem context: An early use of CONTEXT INTERPRETATION is that which structures the problem context W. To do this, we identify domains and descriptions for them.

Application of CONTEXT INTERPRETATION *to* P_{null}

Justification J_{null}*:* We assume that the schematic of Figure 1 shows the context domains and phenomena of interest. The behaviour we are interested in this problem concerns:

- a package label being read by the reading station;
- a package leaving the reading station;
- a package entering or leaving a switch;
- a switch state at any particular time; and
- a package being dropped into a bin

We have aggregated into a single domain a switch domain and its incoming and outgoing sensors; similarly, we have aggregated a bin domain with its incoming sensor.

From the figure, we have identified the following domains as part of the context:

$$On/Off\ Buttons, Operator,\\ Display, Conveyor\ Motor\ and\ Belt,\\ Reading\ Station, Switch[i], Bin[j], Package[id]$$

Concerns: We note that we have not represented pipes explicitly. This choice will prevent us from, for instance, considering possible events which may occur in pipes, such as package overtaking. Should this turn out to be a concern, pipe domains may need to be introduced in the model. Moreover, we consider each sensor's behaviour as bundled with that of the real world domains to which they are connected. For brevity and simplicity, we have disregarded sensor reliability in this initial analysis.

An identity concern [3] arises through the interaction of packages with bins and switches: we need to be able to identify the individual switches through which a particular package is routed, and the particular bin in which it falls. We do this by uniquely indexing the corresponding domains and phenomena (notationally, the index appears in square brackets in domain and phenomena names).

Phenomena: Although Figure 1 shows which domains share phenomena, we will need to describe in more detail the domains before being able to identify the phenomena they share.

Resulting problem:

$$P_0 : \begin{array}{l} On/Off\ Buttons : null, Operator : null,\\ \quad Display : null,\\ \quad Conveyor\ Motor\ and\ Belt : null,\\ \quad Reading\ Station : null, Switch[i] : null,\\ \quad Bin[j] : null, Package[id] : null,\\ S : null \vdash R : null \end{array}$$

Typically, the fit of an early context structure to the real-world problem context will not be adequate, and early work will be necessary to validate it against the real-world structure. Indeed, all assumptions made during development should be validated. The reader will also note that this justification makes some (limited) discussion of the modelling choices, indicating their potential impact, and what might be required should they turn out to be incorrect. An incorrect initial description could have profound implications on subsequent analysis. It is therefore good to alert stake-holders of the possibilities for mistakes. Making sure that our assumptions are justifiable is necessary to the adequacy of our analysis.

2) Domain behaviour and phenomena: Another part of the analysis is to provide domain descriptions for the chosen domains; that is, to replace the various *null*s in P_0 with adequate descriptions. This is another application of the CONTEXT INTERPRETATION at the level of the individual domains. The simpler domains—*Operator*, *On/Off Buttons*, *Bin*s, *Display*—are described in natural language. For the others we adopt state machines [23] as a description language. We consider each of the context domains in turn.

The *Conveyor Motor and Belt*:

Application of CONTEXT INTERPRETATION *to* P_0

Justification $J_{0.CMB}$*:* On receiving *on* and *off* commands from the controller the conveyor motor will activate or deactivate the belt accordingly. Our initial model is shown in Figure 3

Phenomena: The commands *on* and *off* are issued by the controller and shared between the controller and the *Conveyor Motor and Belt*. $dropPkg[id]$ corresponds to a package being dropped from the belt. As multiple packages exist in the system, by convention, we use the package identifier in square brackets to indicate that it is the package with identifier *id* which is being dropped. The package identity is not shared between the package and the router—the shared event is the fact that a package is being dropped, not its identity.

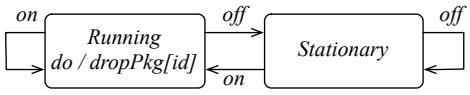


Fig. 3. *CMB_beh: Conveyor Motor and Belt* behaviour

The reader will note that, in describing the domain, we have placed constraints on the solution—that the controller controls and shares the phenomena *on* and *off* with the conveyor belt and motor. The detailing of the context will often reveal the detail of its relationship with the solution.

The *Operator*:

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0.Op}$*:* The *Operator* uses the *On/Off Buttons* to control the *Conveyor Motor and Belt* through the controller. This description of the *Operator*'s behaviour we name *Op.beh*.

The *On/Off Buttons*:

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{On/Off}$*:* The *On/Off Buttons* act as a trivial connection domain [3] between *Operator* and machine. This description of the *On/Off Buttons*'s behaviour we name *OnOff.beh*.

The *Reading Station*:

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0.RS}$*:* A package arrives at the *Reading Station* in the *Ready* state. The station enters the *Reading* state where it reads the label on the package sharing this information with the controller through the parametrised $readPkg(id, dst)$ phenomena; and, after some seconds, entering the *Releasing* state as the package leaves the *Reading Station*, an event (*out*) indicated by a sensor on the pipe; pipe sensor data do not include the identity of the passing package.

Concerns: Depending on the length of time a package waits in the *Reading Station* (the *n seconds* annotation in Figure 4), other packages may arrive during *Reading*. Careful analysis of behaviours with stake-holders will be required to establish whether this is an acceptable simplification, and will perhaps require a subsequent re-interpretation.

Phenomena: $arrivedPkg[id]$ is a phenomenon controlled by $Package[id]$ and shared with *Reading Station*, while $releasePkg[id]$ and *out* are both controlled by *Reading Station*, shared with $Package[id]$ and the controller, respectively.

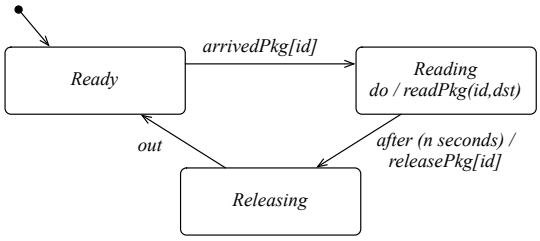


Fig. 4. *RS.beh*: *Reading Station* behaviour

The *Switches*:

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0.Sw}$*:* $Switch[i]$ corresponds to a switch and its incoming and outgoing pipes and sensors. When package *id* passes in front of a sensor associated with switch *i*, then the following events occur:

causing	sensor involved	effect
$in[i][id]$	incoming sensor	$in[i]$
$outLeft[i][id]$	left out	$outLeft[i]$
$outRight[i][id]$	right out	$outRight[i]$

Note that, as for the *Reading Station*, switch sensor data do not include the identity of the passing package.

Concerns: A switch has two positions, left or right, that determine the exit pipe for a package. $Switch[i]$ responds to $left[i]$ and $right[i]$ by changing its position when empty; when not empty the switch may break. The initial model does not take into consideration possible delays in state transition when a switch position is set.

Phenomena: $Switch[i]$ controls, and shares with the controller, phenomena $in[i]$, $outLeft[i]$ and $outRight[i]$. It also shares phenomena $in[i][id]$, $outLeft[i][id]$ and $outRight[i][id]$ with $Package[id]$, which controls them.

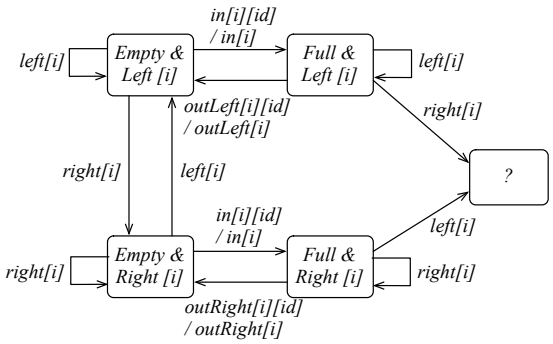


Fig. 5. *Sw.beh*: $Switch[i]$'s behaviour

The *Bins*:

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0.B}$*:* The $Bin[j]$ sensor will issue a signal ($bin[j]$) whenever it senses $Package[id]$ passing (event $bin[j][id]$). As for the *Reading Station*, sensor data does not include the identity of the passing package. This description of the bins' behaviours we name *Bin.beh*.

Phenomena: $bin[j]$ is shared with the controller; $bin[j][id]$ with $Package[id]$.

The *Packages*:

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0.Pkg}$*:* To model a package, we discretise the possible positions of a package as it interacts with the package router. The result is shown in Figure 6[3]. In terms of sensor signals, the journey of a package from the reading station to a bin can be characterised by a unique sequence of events that is characterised by:

- $dropPkg[id]$ is followed by $arrivedPkg[id]$, representing the removal of a package from the belt, and its subsequent arrival at the reading station;
- $releasePkg[id]$ is followed by $in[1][id]$;
- $in[i][id]$ is followed by either $outLeft[i][id]$ or $outRight[i][id]$;
- for a router network which is a binary tree of height $n > 0$ and for $2^{n-1} \leq i \leq 2^n - 1$, each $outLeft[i][id]$ is followed by a $bin[j][id]$ ($j = 2i - 2^n + 1$), each $outRight[i][id]$ is followed by a $bin[j][id]$ ($j = 2i - 2^n + 2$).

Concerns: For simplicity, we have abstracted the router pipes away in our model. Packages may get jammed within pipes and switches and careful consideration with stake-holders will be required to establish acceptable behaviours, perhaps leading to a subsequent re-interpretation. In this initial analysis, we assume that a package will never get stuck in the switch network.

Phenomena: $dropPkg[id]$ is shared with the *Conveyor Motor and Belt*; $arrivePkg[id]$ with the *Reading Station*, $bin[j][id]$ with the $Bin[j]$; $in[i][id]$, $outLeft[i][id]$ and $outRight[i][id]$ with $Switch[i]$.

The *Display*:

[3]For brevity, so that we do not need to name intermediate states, we will use event sequences (with separator a ';') to annotate transitions. For instance, the transition of a package from being in the reading station ($InRS[id]$) to occupying the first switch is indicated by the sequence $releasePkg[id]$; $in[1][id]$.

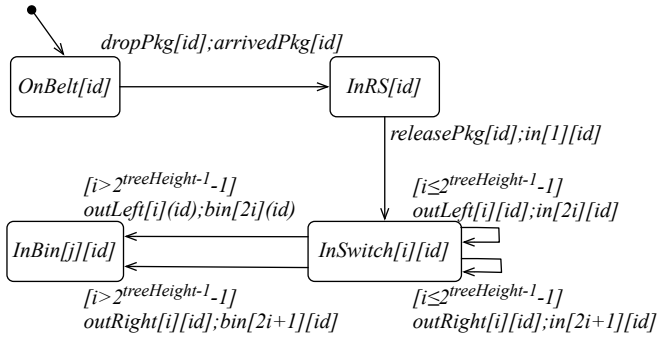


Fig. 6. *Pkg.beh*: *Package[id]* behaviour: from the *InSwitch* state, $j = 2i - 2^n + 1$.

Application of CONTEXT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0,Disp}$*:* The display is used to alert the *Operator* to a package misroute. At this point, we do not detail the messages used to alert the *Operator*: $display(id, dst, k)$ indicates what will be displayed when $Package[id]$, with intended destination dst, is misrouted to $Bin[k]$. This description of the *Display*'s behaviour we name *Disp.beh*.

Phenomena: $display(id, dst, k)$ is shared with the *Operator*.

3) Adding knowledge of the requirements: The problem is to control the operation of the package router so that packages are routed to their appropriate bins, obeying the *Operator*s commands to start and stop the conveyor and reporting any misrouted packages. We now detail the requirement using the REQUIREMENT INTERPRETATION problem transformation (*cf.* CONTEXT INTERPRETATION):

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

Application of REQUIREMENT INTERPRETATION *to* P_0 *(cont'd)*

Justification $J_{0.Req}$*:* The problem with which we are concerned is the design of a software controller so that:

- packages are routed appropriately, with misroutes reported; and
- the *Operator*'s commands to start and stop the conveyor are obeyed.

There are therefore two parts to the requirement:

Obey command: An *on* command from the *Operator* should result in the conveyor belt state *Running*; an *off* command from the *Operator* should result in the conveyor belt state *Stationary*.

and

Route and report: A package arriving at the reading station ($arrivePkg[id]$) with identity id and destination dst should eventually reach $Bin[j]$, where $Bin[j]$ corresponds to destination dst. If conflict at a switch makes this impossible, then $misrouted(id;\ dst;\ k)$ should be reported, where $Bin[k]$ is the actual bin reached.

Concerns: Sensors do not, as noted earlier, return the identity of a package passing in front of them. We therefore have a potential information deficit [3] for package routing (and misrouting), one corollary of which is that the controller will need to infer a package's position in the router from 'information poor' sensor data.

Phenomena: The requirement should reference $arrivedPkg[id]$, *on* and *off*; the requirement constrains $bin[j][id]$, $misroute(id, dst, k)$, *Stationary* and *Running*.

Development summary: As we have taken such large steps, it is worth reviewing progress on the problems. Starting from the *null* problem (P_{null}), though a number of interpretations, we have reached problem P_1:

$$\begin{array}{rl} & Operator : Op.beh, Display : Disp.beh, \\ & \quad On/Off.Buttons : OnOff.beh, \\ & \quad Package[id] : Pkg.beh, Bin[i] : Bin.beh, \\ P_1 : & \quad Switch[j] : Sw.beh, \\ & \quad Reading\ Station : RS.beh, \\ & \quad Conveyor\ Motor\ and\ Belt : CMB.beh, \\ & S : null \\ & \vdash Obey\ command \wedge Route\ and\ report \end{array}$$

The adequacy argument generated for the development so far is:

$$\begin{aligned} CA_1 = J_{null} \wedge J_{0.CMB} \wedge J_{0.Op} \wedge \\ J_{0.OnOff} \wedge J_{0.RS} \wedge J_{0.Sw} \wedge \\ J_{0.B} \wedge J_{0.Pkg} \wedge J_{0.Disp} \wedge J_{0.Req} \end{aligned}$$

A problem diagram [3] for this problem, emphasising the topology of the context, is shown in Figure 7 (*cf.* Figure 1).

C. Architectures and sub-problems

Our requirement has two parts, suggesting that there are actually two separate sub-problems. POSE characterises *separable problems* as follows. A problem $P = W, S \vdash R$ is said to be *n-separable* if there are partitions of W into $W_{io_i}^{c_i}$, $i = 1, ..., n$, and R into $R_{irefs_i}^{cons_i}$, $i = 1, ..., n$, for which no phenomena from one sub-problem can interfere with any other's; if this is the case, P consists of n independently solvable sub-problems.

When a problem is n-separable, SOLUTION INTERPRETATION:

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

is used to interpret the solution as an instance of the *nSep* operator:

$$nSep(S1_{p_1}^{d_1}, ..., S_{np_n}^{d_n})$$

to introduces n independent solution domains. (The solution domains are independent because the sets $(d_i \cup p_i)$ are required to be pair-wise disjoint.) The justification obligation for this SOLUTION INTERPRETATION is to confirm that the problem is *n-separable*:

Application of SOLUTION INTERPRETATION *to* P_1

Justification J_1*:* We wish to interpret P_1 as a 2-separable problem, i.e., to interpret S as:

$$2Sep(Conveyor\ Controller, R\&R\ Controller)$$

where the solution domains are as shown in Figure 8.

To do this we must show that the problem is 2-separable. First, we consider the *Obey command* component of the requirement. It defines a relation that should be true between *Operator* commands and conveyor belt states. In particular, belt states are constrained by the requirement based on *Operator* commands. For such a relation to be established, we can exploit causal relations existing between phenomena of *Operator*, *On/Off Buttons*, *Conveyor Controller* and *Conveyor Motor and Belt*, identifiable from their domain properties. Hence these parts of the context are all to be considered in the sub-problem analysis.

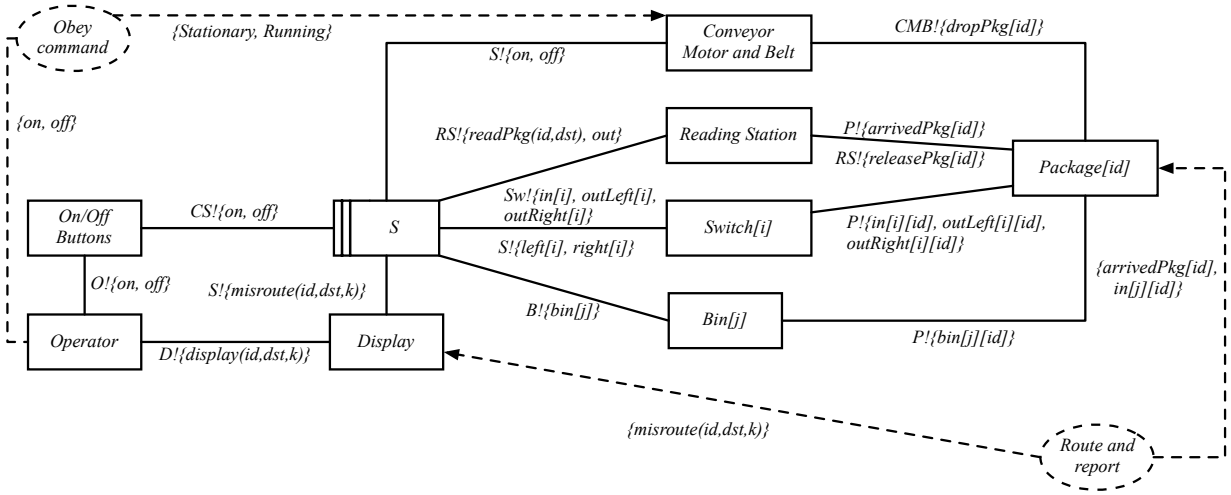


Fig. 7. Problem P_1, including the identified domains, their phenomena and the requirement. The problem topology is highlighted using a problem diagram-like notation [3]: rectangles are domains (the decorated rectangle being the solution); requirements are inscribed dotted ovals; annotations on arcs indicate the sharing of phenomena.

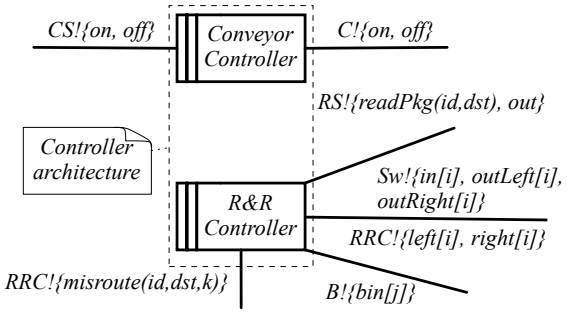


Fig. 8. Separable components of the *Controller*

We should also consider whether there exist other phenomena shared between the *Conveyor Motor and Belt* and the remainder of the problem context, which may influence the state of the belt, and hence the satisfaction of the requirement. From the problem diagram and behaviour descriptions, *Conveyor Motor and Belt* only shares phenomenon $dropPkg[id]$ with $Package[id]$, which is controlled by the conveyor and appears not to have any effect on the belt state, hence we can disregard it. We can therefore ignore the remainder of the context in the analysis of the sub-problem.

With the solution interpreted as a n-separable problem, the SEPARABLE PROBLEM rule makes the separation without generating a justification obligation (because of the preceding justified SOLUTION INTERPRETATION). The transformation is:

$$\frac{W_1, S_1 \vdash R_1 \qquad \ldots \qquad W_n, S_n \vdash R_n}{W, S : nSep(S_{1p_1}^{d_1}, \ldots, S_{np_n}^{d_n}) \vdash R} \text{ [SEPARABLE PROBLEM]}$$

Back to the development: after applying the SEPARABLE PROBLEM rule, we have two sub-problems remaining to be solved[4]:

$$P_3: \quad \begin{array}{l} Operator, On/Off\ Buttons, \\ Conveyor\ Motor\ and\ Belt, \\ Conveyor\ Controller \vdash Obey\ command \end{array}$$

[4]We have omitted the domain descriptions for brevity.

and

$$P_4: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ Reading\ Station, Package[id], \\ R\&R\ Controller \vdash Route\ and\ report \end{array}$$

D. Problem progression

It is widely recognised that real-world requirements are, typically, not expressed in terms of solution phenomena, often being deeply embedded in a complex problem context and described in that context's vocabulary. For instance, *Route and report* is expressed in terms of the phenomena that involve packages, rather than the commands of the controller.

During development, however, real-world requirements will, typically, be re-expressed in terms that are closer to the machine, a process we call *problem progression*. For instance, in the *Route and report* sub-problem (P_4), a progressed requirement would be expressed entirely in terms of the phenomena the *R& R Controller* shares with its context, as opposed to the current requirement constraining *Package[id]* phenomena, which are not accessible by the machine. The result of the transformation, which is discussed in the following, is illustrated in Figure 9: it has the combined effect of re-expressing the requirement in terms of specification phenomena and narrowing the problem context by removing the package domains (greyed out in the figure).

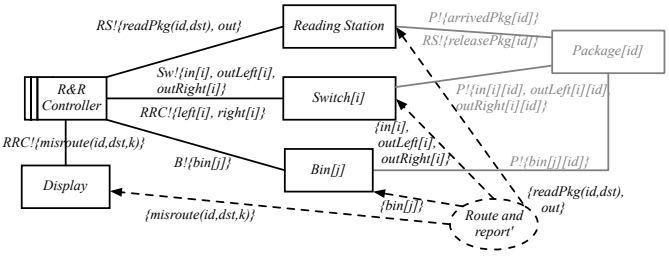


Fig. 9. Problem progression (to problem P_5)

Inspired by the problem progression of [3, Page 103], there has been work to discover the detail of this development step;

Rapanotti *et al.* [24], Li *et al.* [25], and Seater *et al*. [26] have each explored in detail the nature of this underlying transformation. In POSE, we use PROBLEM PROGRESSION:

$$\frac{D_1, ..., D_{n-1}, S \vdash R'}{D_1, ..., D_{n-1}, D_n, S \vdash R}$$ [PROBLEM PROGRESSION] ⟨⟨Explain and justify why R' in the progressed problem is equivalent to R in the original⟩⟩

For application here, the justification for the progression of problem P_4 by removal of the *Package* domain is:

Application of PROBLEM PROGRESSION *to* P_4

Justification J_4*:* Let $route(i)$ denote the correct switch output sensor sequence a package should take on its journey to $Bin[i]$. $route(i)$ will be a sequence of *outLeft* and *outRight* event occurrences.
Consider package *id* with destination *dst* as it leaves the reading station. The happy day scenario is that the package will fall under gravity through the switches on its journey to $Bin[j]$, where $Bin[j]$ corresponds to *dst*. As the package passes various sensors in the router, we can build $route(id, dst)$ as the sequence of output sensors that are actually triggered by the package.
A package is routed properly whenever $route(id, dst)$ is a prefix of $route(j)$. From this observation, we can abstract away the $Package[id]$ domain, and re-express the requirement in terms of sensor signals as follows:
Route and report$'$: A $readPkg(id, dst)$ at the reading station should result in $route(id, dst)$ being a prefix of $route(j)$, the correct path to the appropriate bin. If conflict at a switch makes this impossible, then $misrouted(id;\ dst;\ k)$ should be reported, where $Bin[k]$ is the actual bin reached.
Concerns: We assume all sensors will behave reliably. The controller will have to have knowledge of the package router topology.
Resulting problem:

$$P_5: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ Reading\ Station, \\ R\&R\ Controller \vdash Route\ and\ report' \end{array}$$

E. A further architectural expansion

At this point in the development, we further detail the software structure. The work of this section illustrates how a well-known architectural pattern can be reused within a POSE development.

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 (with certain constraints on the phenomena sets, which we omit here for brevity; the interested reader is referred to [2] for the full definition). (the C_i) with solution components yet to be found (the S_j):

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

with name *AStructName*.

Application of SOLUTION INTERPRETATION *to* P_5

Justification J_5*:* The machine controls a package's route by the timely setting of the package router's switches, according to the destination on the package label. A properly configured controller will know the identity of each switch and the position of each bin.

At this point, we address the information deficit concern from *Justification* $J_{0.Req}$ through the standard strategy of building a model [3]. Moreover, the problem is a control problem in which inputs and outputs are mediated by a model.
This preliminary analysis leads us to the conclusion that a variant of the MVC architecture as defined by Lea [27] and adopted in control applications, such as avionics, may be appropriate. Originally introduced for software applications with graphical user interfaces [28], an MVC architecture includes controller (C) components which receive user inputs and update a model (M) component accordingly; changes in M are communicated to its dependents, called view (V) components, which interpret them and generate user outputs appropriately. In the Lea variant, instead of user inputs, environmental sensor information is received by the controller and used to update the model, while the view generates outputs based on changes in the model's state. Outputs are either user outputs to a display (as in the traditional MVC) or actuator signals through which the controller influences the environment.
The following *AStruct* encodes Lea's MVC variant for use in POSE:

$$R\&RCAStruct(M, V, C)$$

with phenomena as detailed in Figure 10.

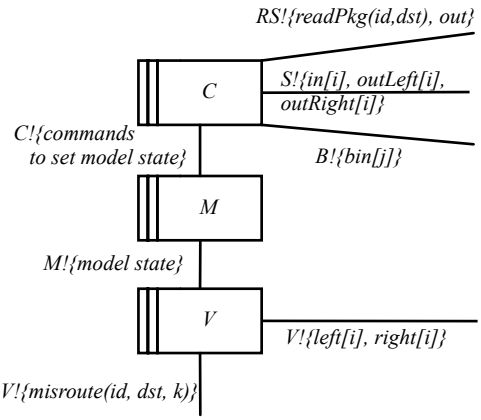


Fig. 10. MVC architecture of the *R&R Controller*

Resulting Problem:

$$P_6: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ Reading\ Station, \\ R\&R\ Controller : R\&RCAStruct(M, V, C) \\ \vdash Route\ and\ report' \end{array}$$

Once the solution is interpreted, 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 sub-problems. SOLUTION EXPANSION has the following form:

$$\frac{\begin{array}{c}\mathcal{W}, \mathcal{C}_1, ..., \mathcal{C}_m, \mathcal{S}_2{:}null, ..., \mathcal{S}_n{:}null, \mathcal{S}_1 \vdash \mathcal{R} \\ \vdots \\ \mathcal{W}, \mathcal{C}_1, ..., \mathcal{C}_m, \mathcal{S}_1{:}null, \mathcal{S}_{j-1}{:}null, \mathcal{S}_{j+1}{:}null, \mathcal{S}_n{:}null, \mathcal{S}_j \vdash \mathcal{R} \\ \vdots \\ \mathcal{W}, \mathcal{C}_1, ..., \mathcal{C}_m, \mathcal{S}_1{:}null, ..., \mathcal{S}_{n-1}{:}null, \mathcal{S}_n \vdash \mathcal{R}\end{array}}{\mathcal{W}, S : AStructName[\mathcal{C}_1, ..., \mathcal{C}_m](\mathcal{S}_1, ..., \mathcal{S}_n) \vdash \mathcal{R}}\ [\text{SOLUTION EXPANSION}]$$

SOLUTION EXPANSION is a deceptively complex rule in that it creates a number of premise problems, each of which requires solving, and each of which contributes its solution to the other premise problems. However, given that the architecture that it expands will already have been justified, SOLUTION EXPANSION does not generate a justification obligation: its role is simply the syntactic separation of the various sub-problems introduced by the architecture.

The *R&RCAStruct* has three 'to-be-designed' components meaning that, from a SOLUTION EXPANSION, we are left with three sub-problems, one for each component, as follows:

$$P_7: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ \quad Reading\ Station, \\ \quad C : null, V : null, M : null \\ \quad \vdash Route\ and\ report' \end{array}$$

$$P_8: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ \quad Reading\ Station, \\ \quad M : null, C : null, V : null \\ \quad \vdash Route\ and\ report' \end{array}$$

$$P_9: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ \quad Reading\ Station, \\ \quad M : null, V : null, C : null \\ \quad \vdash Route\ and\ report' \end{array}$$

It may appear that, in order to solve the model sub-problem (P_7) we need a complete description of the controller and view whereas these will only be available when problems P_8 and P_9 are solved. Moreover, in order to solve either of these problems, we would appear to need a complete description of the model. In this sense, P_7, P_8 and P_9 are co-dependent. In POSE, we *co-design* in the presence of co-dependent sub-problems. One way to resolve the dependencies in co-design is to make an attempt to solve one identified sub-problem. Although it may not be successful (relying as it does on the solutions to other sub-problems), some partial characterisation of the sub-problem's solution may be revealed that can be utilised in attempting to solve a different dependent sub-problem. Iteration between sub-problems will, in the absolute worst case, lead nowhere. However, more often that not, there is some asymmetry in the problems that makes this approach productive.

Consider, for instance, the current co-dependent problems: both controller and view components depend on the model, which is also the most complex component; we will therefore try to solve the model problem first. Notably, this is also the guidance that appears in the literature, for instance [29], [30], with the motivation being that in this way the risk of failing to specify the model correctly is minimised.

F. Solution interpretation for M

Designing M is approached through SOLUTION INTERPRETATION:

Application of SOLUTION INTERPRETATION *to* P_7

Justification J_7*:* We will define M as an analogic model [3] of the package router, which will allow the *R & R Controller* dynamically to track information about bins, switches and their states, and packages and their routes. During operation the analogic model will provide a faithful representation of the package router state.

Designing an analogic model involves making decisions about the machine's data structures and their behaviour, rather than capturing new given properties of the world. Such design decisions are dependent on which questions the model is built to answer – in our example, the identity and position of each package in the router, the switch states, *etc.*

We will, therefore, adopt the more general abstraction of Figure 11 proposed in [3]. Packages travelling in the router can be seen as joining and leaving queues at various points of the router: at the reading station, inside switches and in the pipes that connect reading station, switches and bins. We have reified the abstraction so that the reading station queue has capacity one.

Concerns: This abstraction is sensible only if packages cannot overtake each other while travelling through the router.

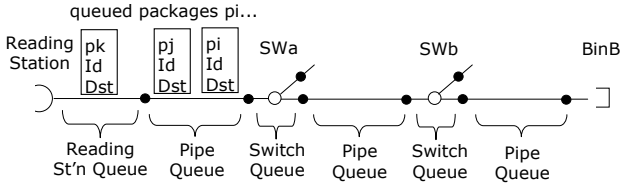


Fig. 11. Using queues as an abstraction

Adopting this abstraction, and by taking an object-oriented design view, we obtain the class diagram description M_{desc} for M shown in Figure 12.

We follow [30] in that classes are defined to represent relevant parts of the real world, with an 'orchestrating class' (OC) added with the responsibility to initialise and orchestrate the behaviour of the model. Class associations reflect the static topological relations between the various parts of the router: for instance, the switch network topology is reflected in the fact the each switch is associated with exactly three package queues, one incoming and two outgoing. The dynamic relationship between packages and queues is also captured by an association: each package can be at most in one queue at any one time.

Phenomena: The operations of class OC specified in Figure 12 are invoked by C on M, and so should be considered as phenomena shared between the two, controlled by C.

Resulting Problem:

$$P_{10}: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ \quad Reading\ Station, \\ \quad C : null, V : null, M : M_{desc} \\ \quad \vdash Route\ and\ report' \end{array}$$

The behaviour of M would typically be detailed as a collection of sequence diagrams [23], each defining the response of M to the various incoming sensor data. This behaviour would include the initialisation of M to reflect the actual topology of the router (i.e., which particular switches are connected to each other, etc.,

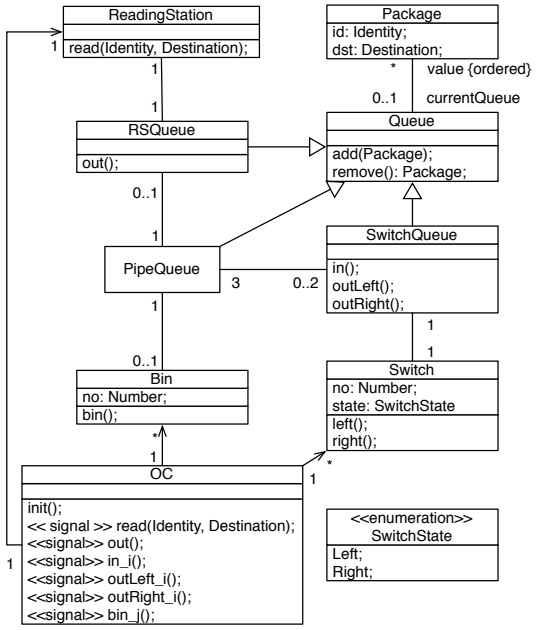


Fig. 12. M_{desc}: Structural description of M

discharging one of the concerns raised in J_4) and the initial state of each switch. We omit the details for brevity, and continue the development under the assumption that such behaviour faithfully models the router in operation.

Note that during detailed design or implementation, a decision has to be made as to how changes in the model are propagated to the view. Standard approaches can be applied—see, for instance, the *push* and *pull* approaches described in [31]—whose choice will depend on efficiency considerations or the availability of appropriate mechanisms at the programming level. Although important, these considerations are outside the scope of this paper, which only deals with problem analysis up to early design. In any case, once a decision is made it would be possible within POSE to base further design on it.

G. Solution interpretation for C

Designing C is a simple step of SOLUTION INTERPRETATION:

Application of CONTEXT INTERPRETATION *to* P_9

Justification J_9*:* Under the assumption that M is described above, the description of C is almost trivial: C is entirely decoupled from V and only responsible for translating inputs from the environment into operation invocations on the only instance of the orchestrating class OC of M.

C_{desc} is defined as follows:

input	operation invocation
$in[i]$	in_i()
$outLeft[i]$	outLeft_i()
$outRight[i]$	outRight_i()
$bin[j]$	bin_j()
$readPkg(id, dst)$	read(Identity,Destination)

Resulting problem:

$$P_{11}: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ \quad Reading\ Station, \\ \quad M : M_{desc}, V : null, C : C_{desc} \\ \quad \vdash Route\ and\ report' \end{array}$$

H. Progressing the View

As part of solving problem P_8, we will progress it by removing the *Reading Station* and *Bin* domains expressing the requirement in terms of V's shared phenomena with the switches. This is illustrated in Figure 13.

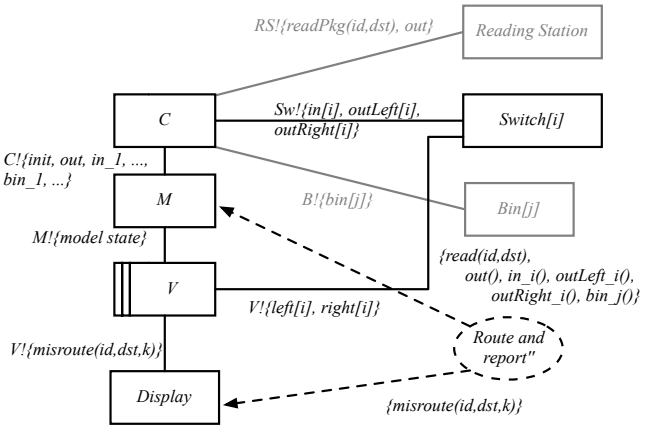


Fig. 13. Progression of the V problem (P_8) to P_{12}

The justification for the progression is:

Application of PROBLEM PROGRESSION *to* P_8

Justification J_8*:* C_{desc} determines that inputs from the environment are translated into operation invocations on the model M. M_{desc} provides a faithful representation of the router in operation. Let route(id,dst) denote the actual route of a package when restated in terms of invocations of outLeft_i() and outRight_i() operations in the model; let route(j) denote the sequence of outLeft_i() and outRight_i() that would result from a correct traversal of the router. To remove *Reading Station* and $Bin[j]$ we may rewrite the requirement as follows:

Route and report'': A read(id,dst) in M should result in route (id,dst) being a prefix of route(j). If, with respect to route(j), an incorrect outLeft() or outRight() is received by C, then misrouted (id; dst; k) should be reported, where bin_k() corresponds to the last element of route(id, dst).

Resulting problem:

$$P_{12}: \quad \begin{array}{l} Display, Switch[i] \\ \quad M : M_{desc}, C : C_{desc}, V : null \\ \quad \vdash Route\ and\ report'' \end{array}$$

I. Backtracking the development

At this point, we can continue with the development of V which (as will be argued presently) introduces unnecessary complexity,

or backtrack to revisit a previous design choice. Although either is a feasible development step, we choose to backtrack to show how, in POSE, iterative development can be done. The point to which we backtrack is that at which P_6 is expanded; the justification of the backtracking is as follows[5]:

Backtracking from P_{12}

Justification J_{Back}*:* For V to be a solution to the progressed sub-problem P_{12}, it needs to be able to determine which route each package should follow and set switch states accordingly. The current description of M does not include this piece of information. A design choice is needed: to include it as part of the specification of V, or to revisit the current design of M. The former would require V to duplicate much of the specification of M.
We prefer therefore to rewind to the point at which P_6 was expanded, and to change the model of Figure 12 so that route information is included in M.

The forward development from the backtracked state is then justified as:

Application of SOLUTION INTERPRETATION *to* P_7

Justification J_7'*:* Add to the class diagram of Figure 12 the class SQueue of Figure 14. This class represents an ordered sequence of switch states and is used for the following three different purposes.

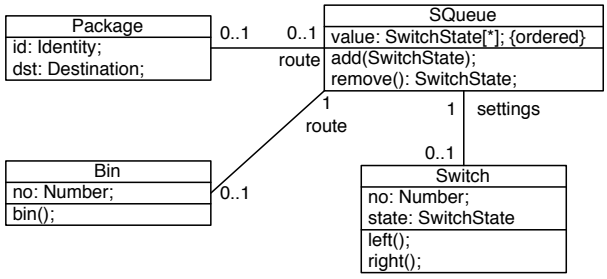


Fig. 14. Routing information added to M_{desc} to give M'_{desc}

For each bin, an instance of the SQueue class represents the route from the reading station to the bin, which is statically determined based on the router topology and set in the model at initialisation.
For each package, an instance of the SQueue class represents the route a package still has to travel in order to reach its destination. This is initially set when the package destination is read at the reading station (through invocation of operation read in the model) and decreases in length as a package goes through each switch on its route.
For each switch, an instance of the SQueue class represents the sequence of settings of the switch which allows the incoming packages (i.e., packages in the in queue of the switch) to be routed correctly. This sequence changes dynamically as packages enters the switch's incoming and outgoing queues.
Includes: J_7 and J_{Back}.

[5]Note that, as backtracking applies to the development rather than to any particular problem, there is no backtracking problem transformation *per se*. However, justification of backtracking is a critical part of development, and so we use the same graphical device for presenting its justification.

Resulting Problem:

$$P_{13}: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ Reading\ Station, \\ C : null, V : null, M : M'_{desc} \\ \vdash Route\ and\ report' \end{array}$$

As before, the behaviour of M could be detailed as a collection of sequence diagrams. (We omit the details for brevity.) With this design, all the complexity is in the model. All that is left for V to do is to sense when switch state changes occur in the model and propagate them to the corresponding router switches through actuators *left*$[i]$ and *right*$[i]$.

Note that the backtracked development requires no change to the design of C:

Application of SOLUTION INTERPRETATION *to* P_9

Justification J_9'*:* Under the new description M'_{desc} of M, J_9 still holds.
Includes: J_9 and J_{Back}.
Resulting problem:

$$P_{14}: \quad \begin{array}{l} Display, Bin[j], Switch[i], \\ Reading\ Station, \\ M : M'_{desc}, V : null, C : C_{desc} \\ \vdash Route\ and\ report' \end{array}$$

and we can complete that of V:

Application of SOLUTION INTERPRETATION *to* P_8

Justification J_8'*:* Under the new description M'_{desc} of M, J_8 for the progression of P_8 still holds, and the description of V, V_{desc} is that on receiving left() and right() invocations on model switches, V translates them into *left* and *right* to the appropriate router switch.
Includes: J_8 and J_{Back}.
Resulting problem:

$$P_{15}: \quad \begin{array}{l} Display, Switch[i] \\ M : M'_{desc}, C : C_{desc}, V : V_{desc} \\ \vdash Route\ and\ report'' \end{array}$$

V. DISCUSSION AND CONCLUSION

The paper has given an account of how the solution to a software-ntensive problem can be approached using the Problem Oriented Software Engineering (POSE) approach proposed in [2]. The development is based on an explicit representation of the problem and its parts and their systematic transformation under a sequent calculus, the definition and use of which has been briefly illustrated by the problem solving process.

The many processes that together comprise software engineering described in the introduction—identification, clarification, understanding, structuring, and justification—each have an explicit place in the POSE problem transformations that we have used here through the various interpretation, expansion and progression schemata.

We have also suggested that POSE provides a structure within which the results of different development activities can be

combined and reconciled. The problem solving process we have carried out in our work on the package router is summarised in Figure 2. It has led to a tree structure that collects the steps of the development together with arguments justifying how each step preserves the adequacy of the solution (alongside any assumptions and concerns that are associated with the step). This structure highlights the progression towards the solution of a system development problem, interleaved with the structure of the adequacy argument that justifies the developed system, and illustrates how tightly the different development activities are combined and reconciled.

A. Early evaluation

Recently, we have worked closely with a safety authority working in the UK safety critical systems industry to evaluate POSE as applied to real-world embedded avionics systems: the goal of the work was to improve the front-end of an existing, successful safety critical development process. This was deemed an appropriate choice to exercise POSE's various features and gather some evidence on its performance in terms of scalability and adequacy. The constraints were that the results should integrate with the existing process and enhance compliance with stringent standards' requirements. We have reported our findings in this area in a series of published papers: [32] shows that POSE transformations can be combined to form a re-usable process pattern for safety-critical development; [33] and [34] focus on process improvement and provide some evidence that POSE in conjunction with Alloy [35] is capable of detection of anomalies early in development, with consequent saving when compared to previous validation work in the original process. Most recently, [36] demonstrates how the POSE notion of transformation and related justification obligation can be exploited for the co-development of both an assurance case and product design.

B. Backtracking, iterative development and development concerns

It is well known that development is an iterative activity. Figure 2's greyed-out nodes—nodes P_{10}, P_{11} and P_{12}—indicate those problems that have been backtracked from, providing a record of the choices that were made and remade during development. In the non-graphical POSE of [2], backtracking rationale is captured in the justification that accompanies a second (and/or subsequent) forward transformation from a backtracked step, again providing a complete record of the design decisions that had been made to arrive at any particular solution. In either case, a review of either development record could determine both why choices were and were not made as part of the development.

Concerns uncovered during development, as appear in the justifications to problem transformations, will often lead to backtracking and iteration if the current development line does not address them and they are not found to be lacking in substance. The result is a process that combines synthetic and analytic steps, similar to those to and from problem P_{12} in Section IV-I. [33] and [36], mentioned above, provide an analysis of iterative development under POSE for a real-world safety-critical software intensive system that also has these characteristics. That *process pattern* maybe be broadly applicable in different contexts, when parameterised appropriately by specific contextual analytical tools, is work in progress.

C. Analogies for reasoning and development

There are two useful analogies that explain choices we have made in representing designs. The first is with the propositional calculus in the way that atomic propositions can be described in any languages. The way this is achieved is that atomic propositions are related though their possible truth values combined through the propositional connectives. If the description language supports it, interpretation of atomic propositions can occur under these connectives so that, for instance, a predicate can have an outermost quantifier removed. The connective we use to bind the various descriptions is the ',' which works not to combine truth values but the occurrences of shared phenomena.

The second analogy is that of mathematical proof, which has led to our Gentzen-style sequent calculus encoding in POSE. It is notable that mathematical proof has many forms, and only in the most highly formal contexts is a 'pure' Gentzen-style derivation produced or required. Rather, most mathematical proofs work at a much more abstract level than is possible in building a Gentzen proof tree, on the understanding that, should it be necessary to provide absolute certainty each high level proof step can be reduced to a combination of lower level steps with the limit being manipulations at the level of very basic steps. The analogy with mathematical proof suggests that POSE might find use as a *touchstone* for software engineering design: most software engineering design will be conducted at a much higher level than is possible within the very small and detailed steps available under POSE. As a foundation, however, for software engineering design we suggest that it should be possible for a developer to reduce their claimed adequate design to a combination of lower level steps within POSE, so that the adequacy of each high level design step can be checked.

D. Future work

Our notion of problem is derived from a proof obligation that appears in [37], there used as one of the criteria for the completeness of requirements engineering. In POSE, this 'problem component' of the proof obligation is made a first-class object as a syntactic problem sequent $W, S \vdash R$, to be synthesised under POSE. The precise nature of the relationship between the two approaches can only be clear when we have finalised the details of our approach, although we are led naturally to the conjecture that whenever (all) the conditions of [37] hold for $W, S \models R$, then $W, S \vdash R$ will be derivable within our framework. This result will, most likely, be our aim for arguments of soundness of our framework. At some point we hope also to consider completeness; but we believe that this must depend on a more exact notion of adequacy than we have so far been able to formulate. Work is in progress to address this.

Of the relationship in the opposite direction, that a solution is found within our framework does not necessarily mean that it will satisfy the conditions of [37], because:

- we do not limit ourselves to the development of solution specifications, i.e., the relationships between events at the machine interface that mediate problem and solution spaces; rather our scope spans the full problem and solution space, including computational artefacts such as code; and
- we ask only for solution adequacy, not a proof of correctness.

Gentzen-style sequent calculi are well known in computing science, not least because they appear naturally automatable, and

have led to many useful tools, such as PVS [38] as well as others. Martin *et al.* ([39]) provide for the direct capture and reply of detailed proofs through programming-like tactic languages. We are currently working on the detail of the 'software engineering design tactic' language that accompanies our system. Interestingly, tactic languages provide higher level transformations even within a Gentzen-style sequent calculus, and so may give an approach to the formal representation of higher level design steps.

Finally, we note that our concept of problem has no inherent bias towards software as a solution, although it is entirely appropriate for it. This leads us to conjecture an extension of problem-orientation to other areas of problem solving and thus, following Vincenti's observation quoted earlier, to other areas of engineering.

ACKNOWLEDGEMENTS

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Awards, and SE Validation Limited, particularly Colin Brain. The comments of the anonymous reviewers helped improve the paper greatly. Thanks also go to our colleagues in the Centre for Research in Computing at The Open University.

REFERENCES

[1] W. M. Turski, "And no philosophers' stone, either," *Information Processing*, vol. 86, 1986.

[2] J. G. Hall, L. Rapanotti, and M. Jackson, "Problem-oriented software engineering," Department of Computing, The Open University, Tech. Rep. 2006/10, 2006.

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

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

[5] M. S. Feather, "Language support for the specification and development of composite systems," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 2, pp. 198–234, April 1987.

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

[7] R.-J. Back and J. von Wright, "Trace refinement of action systems," in *International Conference on Concurrency Theory*, 1994, pp. 367–384. [Online]. Available: http://citeseer.nj.nec.com/back94trace.html

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

[9] S. Mellor, A. Clark, and T. Futagami, "Model-driven development - guest editor's introduction," *IEEE Software*, vol. 20, no. 5, pp. 14– 18, 2003.

[10] W. Swartout and R. Balzer, "On the inevitable intertwining of specification and implementation," *Commun. ACM*, vol. 25, no. 7, pp. 438–440, 1982.

[11] R. Balzer, "Transformation implementation: An example," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 1, pp. 3–14, January 1981.

[12] D. S. Wile, "Program developments: Formal explanations of implementations," *Communications of the ACM*, vol. 26, no. 11, pp. 902–911, November 1983.

[13] A. van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE2001)*, Toronto, 2001, pp. 249–263, 27–31 August 2001.

[14] A. van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy, "The kaos project: Knowledge acquisition in automated specification of software," in *Proceedings AAAI Spring Symposium Series*. Stanford University: American Association for Artificial Intelligence, March 1991, pp. 59–62.

[15] E. Letier and A. van Lamsweerde, "Deriving operational software specifications from system goals," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, 2002.

[16] A. van Lamsweerde, "From system goals to software architecture," in *Formal Methods for Software Architectures*, ser. LNCS, M. Bernardo and P. Inverardi, Eds. Springer-Verlag, 2003, vol. 2804, pp. 25–43.

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

[18] T. Kelly and R. Weaver, "The Goal Structuring Notation — a safety argument notation," in *[40]*.

[19] T. P. Kelly, "Arguing safety - a systematic approach," Ph.D. dissertation, Department of Computing, University of York, 1998.

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

[21] 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.

[22] G. Kiczales and M. Mezini, "Separation of concerns with procedures, annotations, advice and pointcuts," in *Proceedings of 19th European Conference on Object Oriented Programming (ECOOP) 2005*, ser. Lecture Notes in Computer Science, A. P. Black, Ed., vol. 3586. Springer, 2005.

[23] OMG, "Unified Modeling Language (UML), version 2.0," http://www.omg.org/technology/documents/formal/uml.htm. [Online]. Available: http://www.omg.org/technology/documents/formal/uml.htm

[24] L. Rapanotti, J. G. Hall, and Z. Li, "Problem reduction: a systematic technique for deriving specifications from requirements," *IEE Proceedings – Software*, vol. 153, no. 5, pp. 183–198, 2006.

[25] Z. Li, J. G. Hall, and L. Rapanotti, "From requirements to specification: a formal perspective," in *Proceedings of the 2nd International Workshop on Advances and Applications of Problem Frames*, J. G. Hall, L. Rapanotti, K. Cox, and Z. Jin, Eds. ACM, 2006.

[26] R. Seater and D. Jackson, "Problem frame transformations: Deriving specifications from requirements," in *Proceedings of 2nd International Workshop on Advances and Applications of Problem Frames*, Shanghai, China, 2006.

[27] D. Lea, "Design patterns for avionics control systems," State University of New York, Tech. Rep. DSSA Adage Project ADAGE-OSW-94-01, November 1994.

[28] G. E. Krasner and S. T. Pope, "A cookbook for using the model-viewcontroller user interface paradigm in smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, August 1988.

[29] P. Kruchten, *The Rational Unified Process: An Introduction*, ser. Object Technology Series. Addison-Wesley, 2000, vol. 2nd Edition.

[30] C. Larman, *Applying UML and patters*, 2nd ed. Prentice Hall, 2002.

[31] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[32] D. Mannering, J. G. Hall, and L. Rapanotti, "Towards normal design for safety-critical systems," in *Proceedings of ETAPS Fundamental Approaches to Software Engineering (FASE) '07*, ser. Lecture Notes in Computer Science, M. B. Dwyer and A. Lopes, Eds., vol. 4422. Springer Verlag Berlin Heidelberg, 2007, pp. 398–411.

[33] ——, "Safety process improvement with POSE & Alloy," in *Proceedings of The 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2007)*, ser. Lecture Notes in Computer Science, F. Saglietti and N. Oster, Eds., vol. 4680. Nuremberg, Germany: Springer-Verlag, September 2007, pp. 252–257.

[34] ——, "Safety process improvement: Early analysis and justification," in *Proceedings of the 2nd Institution of Engineering and Technology Conference on System Safety 2007*, 2007.

[35] D. Jackson, "Micromodels of software: Lightweight modelling and analysis with alloy," 2001, software Design Group MIT Lab for Computer Science. http://alloy.mit.edu/reference-manual.pdf.

[36] J. G. Hall, D. Mannering, and L. Rapanotti, "Arguing safety with problem oriented software engineering," in *10th IEEE International Symposium on High Assurance System Engineering (HASE)*, Dallas, Texas, 2007.

[37] P. Zave and M. A. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 1, pp. 1–30, 1997.

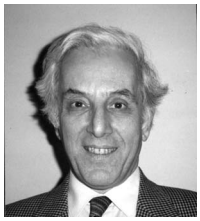
[38] S. Owre, J. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Springer-Verlag, 1992.

[39] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock, "A Tactical Calculus," *Formal Aspects of Computing*, vol. 8, no. 4, pp. 479–489, 1996.

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

Jon G. Hall is a Senior Lecturer in the Computing Department at The Open University. Previously, he held research positions at York and Newcastle-upon-Tyne Universities. His research mission is to provide complementary practical and theoretical foundations for computing as an engineering discipline, to which this papers contributes. Jon is co-Editor-in-Chief of Expert Systems, The Journal of Knowledge Engineering. He has published over 60 academic papers. For more information, please see http://mcs.open.ac.uk/jgh23/.

Lucia Rapanotti is a Senior Lecturer in the Computing Department at The Open University. Previously, she has held both research and software development positions. Her current interest is problem-oriented engineering with applications to both socio-technical and safety-critical systems. She is co-Editor-in-Chief of Expert Systems, The Journal of Knowledge Engineering, and Secretary of the British Computer Society (BCS) Requirements Engineering Specialist Group (RESG). Lucia holds a Laurea Cum Laude in Computer Science from the University of Milan, and a PhD, also in Computer Science, from the University of Newcastle upon Tyne, UK.



Michael Jackson has worked in computer software since 1961. Since 1989 he has worked as an independent consultant and researcher in software development method. He has described his work in many papers and in four books: Principles of Program Design (1974); System Development (1983); Software Requirements & Specifications (1995); and Problem Frames (2001). He has held a number of visiting posts at universities in England and Scotland. He is currently a visiting research Professor both at the Open University and at the University of Newcastle upon Tyne. He is a Fellow of the Royal Academy of Engineering.