ISSN 1744-1986

T e c h n i c a l R e p o r t N o 2010/ 03

# Problem Oriented Software Engineering

Jon G Hall
Lucia Rapanotti, Michael Jackson

29 January, 2010

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

ISSN 1744-1986

**Technical Report N$^{o}$ 2010/03**

# Problem Oriented Software Engineering

***Jon G Hall***
***Lucia Rapanotti***
***Michael Jackson***

*29th January 2010*

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

***http://computing.open.ac.uk***

Under consideration for publication in Formal Aspects of Computing

# Problem Oriented Software Engineering

Jon G. Hall and Lucia Rapanotti and Michael Jackson

Computing Department, The Open University, UK

**Abstract.** A key challenge for software engineering is to learn how to reconcile the formal world of the machine and its software with the non-formal real world. In this paper, we describe Problem Oriented Software Engineering (POSE), an approach that brings both non-formal and formal aspects of software development together within a single theoretical framework for software engineering design.

We show how POSE captures development as the recordable and re-playable design theoretic transformation of software problems. Their representation and transformation allows for 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 stake-holders, that the system will provide what is needed. Designs are recordable and re-playable through our adaptation of *tactics*, a (now standard) form of programming language used in transformational proof theoretic presentations. This brings to our system many other benefits of such approaches, including the ability to abstract from a captured design, and to combine programmatically captured designs.

This paper provides an example-driven presentation of our framework for software engineering design.

**Keywords:** problem orientation; software engineering; Gentzen-style system; design tactics

## 1. 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 [Tur86]:

*Correspondence and offprint requests to*: Jon G. Hall, The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK. e-mail: j.g.hall@open.ac.uk

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.

This paper describes work in progress on a formal framework for software engineering design—Problem Oriented Software Engineering (POSE)—that 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 software development problem; it is also the structure of the adequacy argument that must eventually justify the developed software. POSE allows one to characterise discrete steps of development together with their connections. The connections are not gratuitous: the reader of a design produced through POSE can easily trace which requirements led to each component of the software, with which rationale and justification.

In this paper, we first introduce the main elements of POSE and provide examples that illustrate their important characteristics. The intention is to show how formal and non-formal descriptions are brought together within the POSE theoretical framework in a way that contributes to software engineering design. In Section 4 we adapt and apply Martin *et als.* proof theoretic Angel ([GPMW96]) to our design theoretic setting, showing how design may be recorded for replay, abstraction and programmatic combination.

## 2. Problem Oriented Software Engineering

Problem Oriented Software Engineering (POSE) is a framework for 'solving' software problems. POSE was defined by analogy to Gentzen's systems for proof [Kle64]. Specifically, the basis of a Genzten's system is a *sequent*: a well-formed formula, traditionally representing a logical assertion. The purpose of a sequent is to provide a vehicle for the representation of a logical assertion and for its transformation into other logical assertions in truth-sense preserving ways. In Gentzen-style sequent calculi, if we can transform a logical assertion 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. By analogy, in POSE, we have a notion of sequents representing well-formed *software problems*, i.e., problems that have a software solution and conform to a general form (see below). The transformations defined in POSE transform software problems as sequents into others in ways that preserve 'solution-hood' (in a sense that will become clear). When we have managed to transform a problem to 'axioms' we have solved the problem, and will have a software design to show for our efforts.

There are important differences between Gentzen-style sequent calculi and POSE. A Gentzen sequent has two parts—the antecedent and succedent formulae of the sequent: the succedent is true on the assumption that the antecedent is true. A POSE sequent also has an antecedent and a succedent; these, however, are made of different kinds of objects (as we will see below): world and solution in the antecedent, and requirement in the succedent. Their relation is that the requirement is satisfied by assuming the solution in the world.

Another important difference is the guarding of a transformation by a *justification obligation*, the discharge of which establishes the 'soundness' of the application *with respect to some developmental stake-holder*. This is a radical departure from the universality of truth that an unguarded traditional Genzten-style sequent calculus can show, and it appears 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 systems — are involved. However, as we know from the real world, only when focused is formality appropriate.

Our claim is that POSE offers the theoretical foundations of a practical approach to software engineering design in which the possible roles of formality are separated out, and made clear. We will return to discuss further the relation between POSE and Gentzen's systems in Section 6.

## 2.1. Software problems

A software problem is a requirement, $R$, in a real-world context, $W$ for which a software solution $S$ is sought. The problem context is a collections 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.* [Jac01]). More precisely, a *domain* is a set of related phenomena that are usefully treated as a behavioural unit for some purpose. A domain has a *name* ($N$) and a *description* ($E$) that indicates the possible values and/or states that the domain's phenomena can occupy, how those values and states change over time, and which phenomena—shared or unshared—are produced and when. Associated with each domain $D = N : E$ there are three alphabets:

- the *controlled* alphabet: the phenomena visible to, and that can be shared by, other domains, but controlled by $D$;
- the *observed* alphabet: the phenomena made visible by other domains, that are shared by, and whose occurrence is observed by, $D$;
- the *unshared* alphabet: all phenomena of $D$ that are not visible to other domains, and so not sharable with them.

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, $R = N : E$. A requirement description should always be interpreted in the optative mood, i.e., as expressing a wish. For a requirement $R$, there are two alphabets:

- *refs*: those phenomena of a problem that are *referenced* by a requirement description.
- *cons*: those phenomena of a problem that are *constrained* by a requirement description, i.e., those phenomena that the solution domain's behaviour may influence as a solution to the problem.

A software solution is simply a domain, $S$, ($= N : E$) that is intended to solve a problem, i.e., when introduced into the problem context, the problem's requirements will be satisfied. As such it may have one of many forms, ranging from a 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 the set of *specification phenomena* for the problem. If a requirement's *refs* or *cons* refer to phenomena of the solution domain $S$, they must be specification phenomena.

Each of a problem $P$'s components come together in POSE in a sequent:

$$P: \qquad D_1(p_1)^{c_1}_{o_1} : Des_1, \ldots, D_n(p_n)^{c_n}_{o_n} : Des_n, S^{cs}_{os} : Des_S \vdash R^{cons}_{ref} : Des_R$$

where $\vdash$ is the problem builder and reminds us that the relationship of the solution to its context and to the requirements is what we are seeking to discharge. By convention, the problem's solution domain, $S$, is always positioned immediately to the left of the $\vdash$. The problem name $P$ is sometime omitted.

### *2.1.1. Interpreting descriptions*

Descriptions of a problem's elements may be in any relevant description language; indeed, different elements can be described in different languages. Parts of a problem could be expressed in natural language—'The operator smokes', for example—with others in first order logic—$x = 0 \land x' = 1$, for instance. So that descriptions in different languages can be used together in the same problem, POSE provides a semantic meta-level for the combination of domain (and requirements) descriptions; notationally, this is a role of the ',' that collects into a problem sequent the domains that appear on the left of the turnstile, formally making each visible to the others.

To make this work, however, we must be able to evaluate any description in a way that allows its determination of the way phenomena are constrained to be associated with those of the other domains.

To this end, we consider each description as a timed relation on the occurrence of phenomena it declares, i.e., by writing $D1(p1)^{c1}_{o1} : Des_1$ we mean that $Des_1$, over each interval, defines a relationship between occurrences of the phenomena in $p1 \cup c1 \cup o1$. We may interpret this as 'knowledge' captured by a description if we consider the precision with which this timed relation is defined. The *null* description is the relation that is provides no further information; another description, such as the 'heating system comes on twice every day' tells us something more; yet another, that 'the heater comes on only at 8:45 every day and turns off only at 16:45 every day,' defines the regime with little scope for misunderstanding. (This interpretation is reminiscent of the hybrid systems literature of the mid-1990s, [ORS96], and many of the languages used there would be suitable for the formal description of domains.)

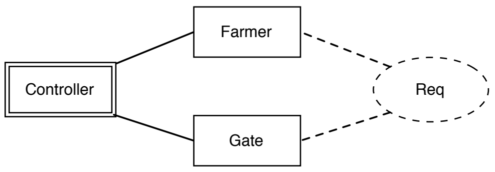

**Fig. 1.** The Farmer Gate Problem

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

The most used notation for capturing problems is natural language. Natural language has no formal interpretation, nor is it subject to formal transformation. Rather, its manipulation is informal, but often much quicker and more convenient than any formal language. It also has the advantage, of course, of being understood by human stake-holders, including those who will validate a problem and its solution, and so real-world developments can seldom make do without it.

The use of natural language introduces no representational problems for POSE problems:

**Example 2.1.** Here is the description of the problem of defining the controller of a Sluice Gate so that a farmer can operate the gate.

| | | | |
|---|---|---|---|
| $P_{Farmer}$ : | *Farmer*: wants be able to raise, lower and stop the *Gate*, *Gate*: installed 3 years ago, *Controller*: *null* | $\vdash$ | *Req*: The *Farmer* should be able to raise, lower and stop the *Gate*. The *Gate* should allow water to flow in the field for ten minutes every three hours. |

Similarly, graphical notations often form an easy basis for communication with non-technical stake-holders:

**Example 2.2.** Figure 1 shows the graphical representation of the farmer's problem, a notation that we often use for illustration alongside our formal expression. The notation is reminiscent of that of Problem Frames [Jac01]. In it a problem has representation provided by graphical elements: domains are undecorated rectangles inscribed with their names, the solution is the double rectangle similarly inscribed, and the requirement is an inscribed dotted ellipse. The sharing of phenomena is indicated by arcs, annotated with detail when useful.

We define a *specification problem* to be a problem that has an empty context ($\emptyset$), a solution *Solution*, to be found, and a single, fully known requirement *Spec* that references and constrains only specification phenomena and that has a description *Desc*:

$$\emptyset, Solution_o^c : null \vdash Spec_o^c : Desc$$

Specification problems are suggestive of the specifications in whose terms formal refinements are written, such as those in Morgan [Mor94] and Back [BvW94], although we make no requirements of formality in a specification problem's descriptions. The following specification problem is adapted from [MV92]:

**Example 2.3.**

$$P1 : \qquad \emptyset, Soln_{sq}^{rt} : null \vdash Spec_{sq}^{rt} : rt = \lfloor\sqrt{sq}\rfloor \tag{1}$$

in which the requirement, *Spec*, constrains the value of $rt$ to be the greatest integer less than the square root of $sq$. $rt$ and $sq$ are specification phenomena.

Actually, specification problems are easy to solve within the framework, but only if one is content with a less than useful, non-computational solution specification: for $P1$, one can simply write (cf., [MV92]):

$$\emptyset, Soln_{sq}^{rt} : rt := \lfloor\sqrt{sq}\rfloor \vdash Spec_{sq}^{rt} : rt = \lfloor\sqrt{sq}\rfloor$$

and call the problem solved. That this simple device solves the specification problem follows from the fact that *Spec* satisfies *Spec*, for any *Spec*.

In passing, this example illustrates an important point: although solution and requirement descriptions are the same, the solution description is an indicative description whereas that of the requirement is optative. So, although descriptions will often be expressed in the indicative or the optative mood, it is actually their position in the problem that defines the actual mood in which the description should be read: indicative descriptions appear on the left, optative descriptions on the right, of the turnstile.

## 2.2. Problem Transformation

Problem transformations capture discrete steps in the problem solving process. Many classes of transformations are recognised in POSE, reflecting a variety of engineering design practices reported in the literature or observed elsewhere. One source of problem transformation is the consideration of architectural trade-offs that can take place during analysis (for instance, [BCK99]). POSE simultaneously generalises and simplifies earlier work of the authors ([HJL$^+$02, RHJN04]), in the SOLUTION EXPANSION transformation illustrated in Section 3.4.

Problem transformations relate a problem and a *justification* to a (set of) problems. Problem transformations are named and 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:

$$\frac{W_1, S_1 \vdash R_1 \qquad \ldots \qquad W_n, S_n \vdash R_n}{W, S \vdash R} \begin{array}{l} \text{[N]} \\ \langle\!\langle \text{J} \rangle\!\rangle \end{array} \tag{2}$$

to mean that $S$ 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. $N$ is the name of the transformation.

*Problem transformation schemata* define classes of problem transformation and include *justification obligations*—from which the justifications in the above rule form stem. Although, from the above, it may appear that the justification has a purely logical nature, there are typically two components: one governing the application of the rule from conclusion problem to premises—detailing how and why the premises are different from the conclusions—the other determining the relationship between the validation of the premises and that of the conclusion—that is, how a validation of the solution to the premise problems extends to the validation of the solution to the conclusion problem.

**Example 2.4.** Consider again the formal specification example above; using the notation of formal refinement from [MV92], and a rule we will name FORMAL REFINEMENT[1] as a transformation, we have the following:

$$\frac{P2: \quad \emptyset, Soln_{sq}^{rt} : null \vdash Spec_{sq}^{rt} : \begin{array}{l} [\![\mathbf{var}\ ru; rt, ru := 0, sq + 1; \\ \quad \mathbf{do}\ rt + 1 \neq ru \rightarrow \\ \quad\quad [\![\mathbf{var}\ rm; rm := (rt + ru)/2; \\ \quad\quad\quad \mathbf{if}\ rm^2 \leq sq \rightarrow rt := rm \\ \quad\quad\quad \mathbf{else}\ rm^2 > sq \rightarrow ru := rm \\ \quad\quad\quad \mathbf{fi}\ ]\!] \\ \quad \mathbf{od}\ ]\!] \end{array}}{P1: \quad \emptyset, Soln_{sq}^{rt} : null \vdash Spec_{sq}^{rt} : rt = \lfloor \sqrt{sq} \rfloor}$$

[FORMAL REFINEMENT]
⟨⟨This solution is constructed by formal refinement, the complete development (with proofs) being contained in [MV92]; we note, however, that as the value of *rt* changes throughout the execution of the code, this computation must be executed atomically for correctness.⟩⟩

with $P2$ giving an (essentially) executable solution for problem $P1$.

This justification would be adequate for even safety-critical developments.

### 2.2.1. *Design Trees and Solved problems*

Engineering design under POSE 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. This model of design is motivated by Gentzen's transformational system for proof.

A metaphor for engineering design under POSE 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,

[1] Which, later, will be seen as an instance of REQUIREMENTS INTERPRETATION.

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: one is that POSE preserves a record of unsuccessful design steps, i.e., design steps that are not validatable for the current stake-holders, that 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[2].

Each tree in the forest grows 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* 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 adequacy argument*[3].

A problem is solved for a stake-holder *SH* if the development tree is complete, and the adequacy argument constructed for that tree convinces *SH* that the solution is adequate. We write

$$\frac{}{P}$$

to indicate that problem $P: \; W, S \vdash R$ is solved.

## 2.3. The nature of justification

Justification is an important element of our framework, and one that distinguishes it from the logical setting of Gentzen. In that setting, the guarding of the applicability of a rules is through the consideration only of syntactically checkable properties of a sequent.

A closer fit for our notion of justification comes from the refinement calculus [MV92] in which proof obligations are generated whenever a refinement is made; its proof justifies the refinement as sound. The strengthen-post-condition refinement, which allows the specification $x$: $[pre, post]$ to be refined by $x$: $[pre, post']$, requires a proof of $post' \implies post$, suitably instantiated with the actual *pre* and *post* predicates, for the refinement to be sound. In this way, strengthen-post-condition carries with it a proof obligation requiring of the refiner a further task, to find a proof. If one sees refinement as a transformation, then the transformation can also be seen as guarded by a proof obligation.

Our notion of transformation guarded by a justification obligation is not so far from this view. Rather than a proof, however, a justification obligation brings with it the task of discovering an argument that will satisfy all stake-holders. The statement of the justification we have found useful to introduce as a CLAIM ([HMR07]) for which arguments and evidence are sought.

We note in passing that, of course, the proof of a proof obligation will satisfy all (rational) stake-holders, and so refinement might be seen as a special case of our system whenever a development can be seen purely formally, and within the remit of the refinement calculi. However, many solutions are adequate without being provably correct: one obvious example is that Microsoft Word proves adequate for document preparation in many organisational settings[4] without being *proven* correct in that role[5]. Many useful solutions would be missed through any absolute correctness notion.

### *2.3.1. Dead-end designs*

In the traditional logical setting, the application of Gentzen-style rules does not always lead to a successful conclusion; the proof of a difficult conjecture will often take many attempts to complete it with many *dead ends* visited on the way. Typically, once encountered, a dead end will lead to a reconsideration of the rules applied and the intermediate statements to see which caused difficulties, with the intent of replacing them with an alternative more successful production. In theorem proving, at least, the main impetus is to construct a proof and so it is unlikely that the 'dead end' development will be captured.

We are no more likely, in POSE, to find fewer dead ends in a design: practically, any problem transformation can

[2] 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. We do not have room to expand on the topic of reuse here.
[3] Of course, if there are no validating stake-holders for a development, the justification obligations can be ignored.
[4] We might say that it adequately solves the document preparation problem for an organisation.
[5] If even such a notion could be formulated.

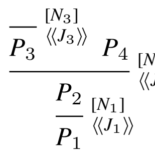

**Fig. 2.** A partial POSE design tree: $P_3$ is solved, but $P_4$ remains unsolved

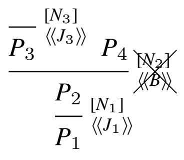

**Fig. 3.** A partial POSE design tree in which $B$ details why the original justification $J_2$ did not discharge its justification obligation, creating a *dead-end* design

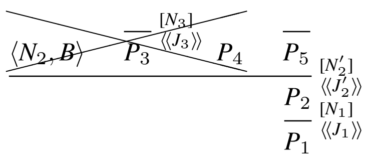

**Fig. 4.** Having recorded the dead-end design of Figure 3, a complete POSE design tree is constructed as forward development from it with new justification $J_2'$.

lead to a problem for which no adequate designed artefact exists. In POSE, we claim, there is a very good reason to capture the routes to 'dead end' designs as they may contain useful information in the intermediate problems they relate or in the justifications used to relate them, which may be of great use in any subsequent design. To this end, we will not discard dead-end designs, rather keeping them in the tree as a record of previous attempts. The rule applications at the base of a dead tree will have the justification crossed through as an indication that it is 'dead'. A tree with dead-end productions can be considered an ordinary design tree by, simply, ignoring the dead-end productions.

A validating stake-holder for a design will be one important source of dead-end designs: a dead-end design will be the result of their withholding validation. It may appear that we introduce unnecessary difficulties into the problem solving process by requiring that the solution be justified as adequate *with respect to a stake-holder view of what adequacy might mean*. Indeed, were it not for this perspective, rules could omit the relative notion of justification to use an absolute correctness notion, such as proof, in its stead. The involvement of validating stake-holders is, however, a necessary evil as the problems we are trying to solve in POSE exist in a real-world context and with requirements both of which are difficult to understand.

Figures 2–4 show partial development trees. Figure 2 contains four nodes, one for each the four problems $P_1$, $P_2$, $P_3$ and $P_4$. The problem transformation that gave the problem $P_2$ is named $N_1$ and justified by $J_1$ whereas the branching to problems $P_3$ and $P_4$ is effected by the transformation named $N_2$ and justified by $J_2$. From the tree, we see that $P_3$ appears solved but that $P_4$ remains unsolved. In Figure 3, we have recorded in $B$ the failure of $J_2$ as justification for $N_2$: presumably, some stake-holder found $J_2$ unconvincing or wrong, and so was unable to validate it. This is recorded for posterity. In Figure 4 further forward development is made, this time using transformation $N_2'$ with justification $J_2'$ leading to $P_5$, which is solved; and leading to adequacy argument

$$J_2' \wedge J_1$$

A complete design is produced when no upward growth of the corresponding POSE development tree is possible. A solved problem is a completed design for which the constructed argument is validated by all interested stake-holders as adequate.

**Example 2.5.** Again considering the formal specification example above; we can complete the tree with the application of another rule, this time called REFLECTION[6], which allows us to complete the development tree shown in Figure 5.

**Example 2.6.** Returning to our farmer's problem, the design tree of Figure 6 shows backtracking and subsequent forward development: it illustrates a small transformation of one natural language description (that emboldened in the conclusion problem) into others (emboldened in the premise). Essentially, the designer has identified the model of the *Gate*. We note that the initial, subsequently backtracked, justification is relatively weak as it relies on the knowledge of the domain engineer rather than any established fact about the *Gate*, but has the correct form: it is an argument that explains and justifies, to a validating stake-holder, how the interpretation was arrived at (a reasoned inference) and why it was made.

Looking at the step from the perspective of the sceptical stake-holder asked to validate the design, we might understand the difficulties of an incorrect identification of the sluice gate—a wrong model may lead to a designed solution that is unable to correctly control the sluice gate—and so may refuse to validate this step, question the

[6] Which will be seen to be an instance of SOLUTION INTERPRETATION.

$P3$ : $\emptyset, Soln^{rt}_{sq}$ :

```
⟦var ru; rt, ru := 0, sq + 1;
  do rt + 1 ≠ ru →
    ⟦var rm; rm := (rt + ru)/2;
      if rm² ≤ sq → rt := rm
      else rm² > sq → ru := rm
      fi ⟧
  od ⟧
```

$\vdash Spec^{rt}_{sq}$ :

```
⟦var ru; rt, ru := 0, sq + 1;
  do rt + 1 ≠ ru →
    ⟦var rm; rm := (rt + ru)/2;
      if rm² ≤ sq → rt := rm
      else rm² > sq → ru := rm
      fi ⟧
  od ⟧
```

[REFLECTION]
⟨⟨A specification satisfies itself.⟩⟩

$P2$ : $\emptyset, Soln^{rt}_{sq} : null \vdash Spec^{rt}_{sq}$ :

```
⟦var ru; rt, ru := 0, sq + 1;
  do rt + 1 ≠ ru →
    ⟦var rm; rm := (rt + ru)/2;
      if rm² ≤ sq → rt := rm
      else rm² > sq → ru := rm
      fi ⟧
  od ⟧
```

[FORMAL REFINEMENT]
⟨⟨This solution is constructed by formal refinement, the complete development (with proofs) being contained in [MV92]; we note, however, that as the value of *rt* changes throughout the execution of the code, this computation must be execute atomically for correctness.⟩⟩

$P1$ : $\emptyset, Soln^{rt}_{sq} : null \vdash Spec^{rt}_{sq} : rt = \lfloor\sqrt{sq}\rfloor$

**Fig. 5.** A complete formal development from specification to code (adapted from [MV92])

~~⟨DOMAIN INTERPRETATION, **Not validated by stakeholder**: "The *SGVert* was the only model of sluice gate produced 3 years ago. A manual describing the *SGVert* is available."⟩~~

~~*Farmer*, *Gate* : The *SGVert* model, *Controller* $\vdash Req$~~

*Farmer*, *Gate* : **The *SGVert* model**, *Controller* $\vdash Req$

[DOMAIN INTERPRETATION]
⟨⟨The sluice gate model has been confirmed by field inspection as the *SGVert* model. A manual describing the *SGVert* is available. ⟩⟩

$P_{Farmer}$ : *Farmer*, *Gate* : **Installed 3 years ago**, *Contoller* $\vdash Req$

**Fig. 6.** Transforming natural language descriptions using interpretation, with backtracking from a dead-end design caused by an unvalidatable justification, and subsequent forward development based on a stronger justification.

designer's knowledge and require a better justification. As shown, this has resulted in a stronger argument being made, based on a field inspection.

Finally, the rationale for the change induced under domain interpretation—that a manual exists for the identified model—is also important as well as providing rich traceability links through the design: interested parties will be able to understand easily the motivation for this step; in an extreme case, the customer who queries an expense claim for this site inspection may see it as a reasonable expense from the justification.

## 3. Software problem solving in POSE

In this section we introduce and illustrate the main POSE features which support design as problem solving.

### 3.1. The starting point

The *null problem* is the problem of which we know nothing:

$W : null, S : null \vdash R : null$

Here, *null* is used as $W$, $R$ and $S$ description to indicate that nothing is known about them: *null* is the bottom element in the information order[7]. The *null* problem has a distinguished place in a POSE development; it can be thought of as the starting point of such a development, with context, requirement and solution descriptions being developed therefrom.

[7] I.e., *null* is seen as having 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.

## 3.2. How improved knowledge is captured

A major part of software development is about eliciting knowledge and providing appropriate descriptions. Importantly, such knowledge and descriptions change as development progresses. The problem interpretation transformation schemata allows for the capture and growth of problem descriptions under POSE. There is an interpretation schema for each of the elements—the context domains, the solution and the requirement—of a problem.

For instance, here is the transformation schema for DOMAIN DESCRIPTION INTERPRETATION (shortly, DOMAIN INTERPRETATION) by which a (non-solution) domain description is interpreted, i.e., re-expressed in some way to make it more suitable for problem solving[8]:

$$\frac{\mathcal{W}, \mathcal{N} : \mathcal{E}', \mathcal{S} \vdash \mathcal{R}}{\mathcal{W}, \mathcal{N} : \mathcal{E}, \mathcal{S} \vdash \mathcal{R}} \;\begin{array}{l}\text{[DOMAIN INTERPRETATION]}\\ \langle\!\langle \text{CLAIM: } \mathcal{E}' \text{ is more useful than } \mathcal{E} \rangle\!\rangle\end{array}$$

in which $\mathcal{E}'$ is the interpretation of $\mathcal{E}$, the original description of the domain named $\mathcal{N}$. The justification of the claim for DOMAIN INTERPRETATION must argue that an 'adequate' interpretation has been found; the meaning of adequate will, in general, be defined by context. The name of the rule is written above the justification.

Note that nowhere is formality used in the manipulation of the problem, and yet we have been able to introduce a rigour into the argument step. This follows directly from the separation of formality from the problem representation and the transformation step.

The other problem interpretation rule is REQUIREMENT INTERPRETATION:

$$\frac{\mathcal{W}, \mathcal{S} \vdash \mathcal{R} : \mathcal{E}'}{\mathcal{W}, \mathcal{S} \vdash \mathcal{R} : \mathcal{E}} \;\begin{array}{l}\text{[REQUIREMENT INTERPRETATION]}\\ \langle\!\langle \text{CLAIM: } \mathcal{E}' \text{ is more useful than } \mathcal{E} \rangle\!\rangle\end{array}$$

## 3.3. Moving towards solutions

Real-world requirements are typically not expressed in terms of solution phenomena, being deeply embedded in a complex problem context and described in the context's vocabulary—our *Farmer* was more concerned to get the regime of water flow in a field correct than with the sequence of commands which should be issued by the *Controller* to the *Gate*. During development, however, real-world requirements should lead the designer to requirements more directly related to the solution, from which detailed solution design can take place.

Problem progression transforms problems in a way inspired by [Jac01, Page 103], and explored in detail in [RHL06, LHR06, SJ06]. In problem progression, requirements on domains that are 'deep in the world' are transformed into requirements on domains that are closer to the machine.

In POSE, there is no PROBLEM PROGRESSION rule *per se*; rather we derive problem progression from two simpler rules. In the first, we 'unshare' phenomena by internalising them into the *to-be-progressed domain D* making an assumption in the requirement equivalent to the constraints that the to-be-progressed domain places on its newly unshared phenomena; in the second, a domain that shares no phenomena is removed from the context of a problem.

The implementation of this rule is done in two steps, as shown in Figure 7:

- in the first, shown in Figure 7(a), we 'unshare' the observed phenomenon $o$ by internalising it into the to-be-progressed domain, $\mathcal{D}$, adding an assumption to the requirement equivalent to the constraints that the to-be-progressed domain places on it: the requirement becomes $\mathcal{R}' : \mathcal{F}'$ where

  $$\mathcal{R}' = \mathcal{R}^{cons \cup \{o\}}_{refs}$$

  and

  $$\mathcal{F}' = \textit{“Assuming o constrained by } \mathcal{E}, \mathcal{F}\textit{”}$$

  The shared phenomenon $o$ is, after transformation, constrained by the rewritten requirement;
- in the second, shown in Figure 7(b), a domain that shares no phenomena may be removed from the context of a problem.

[8] It may be that the intention of the interpretation is to clarify a description, to introduce a domain's shared or unshared phenomena, to make it more formal, to disambiguate it, or any other of a multitude of interpretations.

$$(a)\quad \frac{\mathcal{W}, \mathcal{D}(o) : \mathcal{E}, \mathcal{S} \vdash \mathcal{R}' : \mathcal{F}'}{\mathcal{W}, \mathcal{D}_o : \mathcal{E}, \mathcal{S} \vdash \mathcal{R} : \mathcal{F}}\ [\textsc{Sharing Removal}] \qquad (b)\quad \frac{\mathcal{W}, \mathcal{S} \vdash \mathcal{R}}{\mathcal{W}, \mathcal{D}^{\emptyset}_{\emptyset}, \mathcal{S} \vdash \mathcal{R}}\ [\textsc{Domain Removal}]$$

**Fig. 7.** (a) An assumption is added to the requirement equivalent to the constraints a domain, $\mathcal{D}$, places on one of its shared phenomena, here for a observed phenomenon, $o$ (there is another rule for controlled phenomenon *mutatis mutandis*); (b) a domain that shares no phenomena is removed from the context of a problem.

$$\cfrac{\cfrac{Op(safe, pos)^{cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op\_with\_Assumption^{safe,drop,pos}}{Barrier(pos), Op(safe, pos)^{cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op\_with\_Assumption^{safe,drop,pos}_{cmnds}}\ [\textsc{Domain Removal}]}{Barrier_{pos}, Op(safe)^{pos,cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op^{safe,drop}_{cmnds}}\ [\textsc{Sharing Removal}]$$

**Fig. 8.** A problem progression-like transformation based on sharing removal and domain removal.

**Example 3.1.** Consider the problem of developing software *Soln* to control an automated car body press:

$$Barrier_{pos}, Op(safe)^{pos,cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op^{safe,drop}_{cmnds}$$

which includes a safety barrier, *Barrier*, designed to keep the operator, *Op*, at a safe distance from the press, *Press*. The barrier works by constraining the operator's position, *pos*, to be a safe distance from the press during press operation. The problem requirement states that:

*Safe_Op*: The press should respond to the commands of the operator when safe to do so. The operator should be kept safe at all times.

We wish to remove the *Barrier* domain from the problem context to make the problem easier to solve. We note that the *Barrier* cannot simply be removed from the context of a problem whilst retaining the same requirement as doing so would lead us to the problem:

$$Op(safe, pos)^{cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op^{safe,drop}_{cmnds}$$

in which the operators position is unconstrained and which is, therefore, a very much more difficult problem to solve in software!

Instead, we first apply the SHARING REMOVAL rule to 'loosen' the relationship between the barrier and the operator, rewriting the requirement to constrain the now internal *pos* phenomena of the operator, so:

*Safe_Op_with_Assumption*: *Assuming the operator is at a safe distance from the press,* the press should respond to the commands of the operator when safe to do so. The operator should be kept safe at all times.

leaving the problem

$$Barrier(pos), Op(safe, pos)^{cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op\_with\_Assumption^{safe,drop,pos}_{cmnds}$$

Now, the *Barrier* no longer shares phenomena with other domains and so can be removed through DOMAIN REMOVAL, leaving:

$$Op(safe, pos)^{cmnds}, Press_{drop}, Soln^{drop}_{cmnds} \vdash Safe\_Op\_with\_Assumption^{safe,drop,pos}_{cmnds}$$

Figure 8 shows the two-step 'problem progression.'

Note that the SHARING REMOVAL and DOMAIN REMOVAL rules do not require justification: by construction, by properly accounting for the effects of the domain's behaviour in the requirement, each guarantees that the step is solution preserving[9].

## 3.4. Using normal design expertise

In [Vin90], Vincenti distinguishes *normal* design from *radical* design:

The engineer engaged in [normal] design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has good likelihood of accomplishing the desired task.

[9] The specification problems of Section 2.2 can be simply characterised as problems to which the sharing and domain rules have exhaustively been applied: we simply move the indicative complexity of the context into the optative requirement to derive a specification problem.

If we accept that one form of normal design for software is in the use of software architectures, then normal design is accommodated in POSE through a combination of SOLUTION INTERPRETATION and SOLUTION EXPANSION.

Within a POSE development, the *AStruct* (short for *Architectural Structure*) operator combines, in a given topology, a number of extant (solution) domains with domains yet to be found. An *AStruct* has the following form:

$$AStructName[C_1, ..., C_m](S_1, ..., S_n)_o^c \tag{3}$$

with name *AStructName*, domains $C_i = N_{i o_i}^{c_i} : Desc_i$, and solution domains $S_{j p_j}^{d_j}$.

In essence, by application of an architectural structure to a solution domain, we define its structure (given by the sharing of phenomena between the various components) together with some partial description of its behaviour (given by the descriptions of the extant domains). There may, of course, be more or less known about the extant components—the $C_i$—that make up the architecture. An *AStruct* can be introduced through the SOLUTION INTERPRETATION schema[10]:

$$\frac{\mathcal{W}, \mathcal{N} : \mathcal{E}' \vdash \mathcal{R}}{\mathcal{W}, \mathcal{N} : \mathcal{E} \vdash \mathcal{R}} \begin{array}{l}[\text{SOLUTION INTERPRETATION}] \\ \langle\langle \text{CLAIM: } \mathcal{E}' \text{ is more useful than } \mathcal{E} \rangle\rangle\end{array}$$

**Example 3.2.** Consider a journal editor's problem of producing an editing tool to assist an author working with a mouse and keyboard (*m&k*) to record their expressed ideas as a document (*Doc*) to a specific format (*form*) for a journal. We will assume that a decision has been taken (expressed through a preceeding SOLUTION INTERPRETATION that we do not show) to base the editing tool on a current LaTeX system. We will assume, therefore, that the LaTeX system has architectural structure description:

$$LaTeX[LaTeX\_Env_{form,m\&k}^{stlcds,EditOps}](LaTeX\_Class_{stlcds}^{form})_{m\&k}^{EditOps}$$

where

$$LaTeX\_Env_{form,m\&k}^{stlcds,EditOps} : \quad LaTeX \text{ version } 3.1415926$$

The resulting transformation is shown in Figure 9.

Once an architectural structure is identified as being adequate through SOLUTION INTERPRETATION, SOLUTION EXPANSION generates premise problems by moving the already defined domains $C_i$ to the environment, hence expanding the problem context, whilst simultaneously refocussing the problem to be that of finding the descriptions of the domains $S_j$ that remain. The requirement and context of the original problem is propagated to all sub-problems. SOLUTION EXPANSION is a deceptively complex rule in that it creates a number of premise problems, each of which much be solved, 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 rule is:

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

The transformation of the architecture under the SOLUTION EXPANSION rule is shown in Figure 9. Through it, the editor's original problem is reduced to the simpler problem of developing a class file *LaTeX_Class* that captures the journal's format for use in an expanded context in which the *LaTeX_Env* appears.

### *3.4.1. Completing a design with a known solution*

[10] Of course, solution description forms other than *AStruct* are possible.

| | | | | |
|---|---|---|---|---|
| *User*: | uses mouse and keyboard to express their ideas | | *JEReqs*: | The *EditSys* should allow an author to prepare documents expressing their ideas through keyboard and mouse to make a properly formatted submission to the Journal. |
| *Doc*: | a document to be submitted | $\vdash$ | | |
| *LaTeX_Env*: | *LaTeX* version 3.141592 | | | |
| *LaTeX_Class*: | *null* | | | |

———————————————————————————— [SOLUTION EXPANSION]

| | | | | |
|---|---|---|---|---|
| *User*: | uses mouse and keyboard to express their ideas | | *JEReqs*: | The *EditSys* should allow an author to prepare documents expressing their ideas through keyboard and mouse to make a properly formatted submission to the Journal. |
| *Doc*: | a document to be submitted | $\vdash$ | | |
| *EditSys*: | $LaTeX[LaTeX\_Env](LaTeX\_Class)$ | | | |

**Fig. 9.** SOLUTION EXPANSION of the *EditSys* architecture: after the expansion only the *LaTeX* class remains to be designed.

$$\dfrac{\dfrac{}{User, Doc, EditSys : LaTeX[LaTeX\_Env, LaTeX\_Class]() \vdash JEReqs}\ [\textsc{Solution Expansion}]}{User, Doc, EditSys \vdash JEReqs}\ [\textsc{Solution Interpretation}]$$

⟨⟨LaTeX system with class file is a well-known solution to journal editors' problems⟩⟩

**Fig. 10.** A solved problem resulting from SOLUTION EXPANSION of a completely defined *Atruct*.

The reader will have observed that there is a special case in SOLUTION INTERPRETATION, i.e., that is when there are no 'to-be-found' components (i.e., $n = 0$), as a subsequent SOLUTION EXPANSION will generate no sub-problems. In this case, everything is known of the architecture, and the problem has been solved by the choice of architecture. Recall that the architecture will have been introduced and justified through the solution interpretation rule.

**Example 3.3.** Consider again the journal editor's problem. Let us assume that the editor is willing to re-use another journal's class, contained in the LaTeX class file, which we will assume is called `fac.cls`. In this case, the architecture to be applied under the rule is:

$$LaTeX[LaTeX\_Env^{stlcds,EditOps}_{form,m\&k}, LaTeX\_Class^{form}_{stlcds}]()^{EditOps}_{m\&k}$$

with

$$\begin{aligned} LaTeX\_Env^{stlcds,EditOps}_{form,m\&k} &: \text{LaTeX version 3.141592} \\ LaTeX\_Class^{form}_{stlcds} &: \texttt{fac.cls} \end{aligned}$$

The design tree resulting from the cascaded application is shown in Figure 10 (descriptions are those of Figure 9). The transformation schemata we have defined provide the formal transformational basis of POSE.

## 4. Design Automation through Design Tactics: Introducing Dactics

The computational view of mathematical proof has become popular with the rise of formal methods, and many tools exist to assist the prover. One, with a fully formal semantics and proof theory of its own, is Angel [MGW96, GPMW96, MNU97], a general purpose tactic language tailored to the transformation of logical objects such as sequents. Angel was originally intended for a proof-theoretic setting, but it assumes nothing of the objects it transforms and has found application also in non-logical settings, including the refinement calculus [OC00, OCW03]. Here we extend Angel to apply in our POSE design setting.

### 4.1. Why would one want to automate design?

The development of a design tree has, so far, been presented in general terms through the application of transformation schemata to the open problems that exist in a partially explored design tree. The choice of a particular schema is at the whim of the developer; as is the choice of parameters that should be used to instantiate it and the justification that accompanies its application. The aim of any application is to provide a stake-holder more evidence that the claim of adequacy is well-founded; the ultimate aim is to show the stake-holder a design and an adequacy argument sufficient to convince them of the adequacy of that design.

If that is sufficient then we may stop there. There are times, however, when more is wished for: a customer may wish to produce a number of products that are similar, with minor differences between many lines, or with configuration options for the end-user; a developmental stake-holder, such as a project manager or technical architect may wish to record a particular design as part of an experience archive that could, if techniques existed, be replayed for little or no cost in future design situations.

If our topic were mathematical proof, then tools suitable for these applications have existed now for many years: they are called tactics, and they are essentially programs for doing proof. For design, which is not mathematical logic, there are tools for *recording* designs, such as lessons learned journals [RA93] and design patterns [GJVH95], for instance. For the replay of a design, there are many barriers to be overcome: for instance, even within POSE there are two elements of the problem world—the context and requirements—that could change between applications, and the solution space—consisting of a toolbox of components, for instance, or patterns—will always be mutable too. It is our aim in this section to define a system by which the translocation of designs between mutable problem and solution spaces is possible.

The tools we apply and extend are those provided for mathematical logic by Angel. The extensions are similar to those of ArcAngel [OCW03], which do not weaken the semantic basis of Angel.

## 4.2. Design tactics: Dactics

In a POSE development, the application of a transformation rule to a single problem may give rise to several sub-problems together with a (number of) justification obligations. A transformation can apply only at the open branches of the design tree. A problem that ends an open branch of a design tree will be called an *active site*:

$$\mathit{ActiveSite} == \mathit{Problem}$$

**Example 4.1.** In Figure 2 there is a single active site, $P_4$.

A design tree may have many active sites; indeed, it is sufficient to know the list of active sites to know where a design tree may be extended upwards. To be able properly to represent the *state* of a design tree, we must also know which justifications obligations have accrued from transformation applications. A *design state*, or *DState*, is therefore:

$$\mathit{DState} == [\mathit{ActiveSite}] \times \mathbb{P}\mathit{Justification}$$

Technically, to form the *list* of active sites, we must make a definite choice of ordering from where they appear in the tree. We will do this by reading from left to right across the tree to make the list. Alternatively, we could have formed the *set* of active sites. Our choice of the former is determined by the the later formulation of Angel found in Gumtree [MNU97].

To form the root of a design tree we define an injection of *Problem* into *DState* thus:

$$\iota(p) = ([p], \emptyset)$$

**Example 4.2.** To begin the design of Figure 2, we consider $P_1$ as an active site:

$$\iota(P_1) = ([P_1], \emptyset)$$

and as the design of Figure 2 progresses, its design states are:

$$([P_2], \{J_1\})$$
$$([P_3, P_4], \{J_1, J_2\})$$
$$([P_4], \{J_1, J_2, J_3\})$$

## 4.3. Manipulating design trees with dactics

Dactics change design state by transforming active sites into active sites, adding any necessary justification obligations as they do so. So that they can effect design state changes, dactic always eventually resolve down either to primitive POSE transformations or to a design state 'no-op' dactic, such as **skip** (introduced later). Although dactics may appear to be an extra layer of formalism simply to be able to apply a POSE transformation, the benefit of dactics is the programmatic flexibility with which primitives can be combined, and the powerful pattern matching features we inherit.

There are dactics for each of the basic transformation schemata introduced elsewhere in this paper: the convention we use is to shorten the schema name to name the dactic, prefix it with the Angel keyword **rule**, and to give formal parameters that indicate the variability in the rule. For instance, the SHAREM dactic applies to an active site the SHARING REMOVAL schema:

$$\textbf{rule}\ \textsc{ShaRem}(\mathcal{D}, o) \quad \text{applies} \quad \frac{\mathcal{W}, \mathcal{D}(o) : \mathcal{E}, \mathcal{S} \vdash \mathcal{R}' : \mathcal{F}'}{\mathcal{W}, \mathcal{D}_o : \mathcal{E}, \mathcal{S} \vdash \mathcal{R} : \mathcal{F}}\ [\textsc{Sharing Removal}]$$

which has formal parameters that indicate the domain, $\mathcal{D}$, which it will transform, and which shared phenomenon, $c$, should be removed.

SHARING REMOVAL requires no justification obligation to be generated; when a dactic requires a justification obligation to be discharged, we add a formal parameter which is the argument that justifies its application. The DOMINT dactic applies to an active site the DOMAIN INTERPRETATION schema:

$$\textbf{rule}\ \textsc{DomInt}(\mathcal{N}, \mathcal{E}, \mathcal{E}', \mathcal{J}) \quad \text{applies} \quad \frac{\mathcal{W}, \mathcal{N} : \mathcal{E}', \mathcal{S} \vdash \mathcal{R}}{\mathcal{W}, \mathcal{N} : \mathcal{E}, \mathcal{S} \vdash \mathcal{R}}\ \begin{array}{l}[\textsc{Domain Interpretation}]\\ \langle\langle \textsc{Claim}: \mathcal{E}' \text{ is more useful than } \mathcal{E} \rangle\rangle\end{array}$$

adding the $\mathcal{J}$ to the set of justifications.

For a dactic $d$, in the special case of applying it to an identified active site there are two possible outcomes, either:

- $d$ is applicable to the active site, i.e., the schema applied by the dactic matches the problem at that site, whence the dactic applies and the outcome is the active site that would be expected by the rule application;
- $d$ is not applicable, i.e., the schema does not match, whence the dactic application fails.

Much of the machinery Angel provides copes with the complexity of applying complex dactics within the branching context of proof trees[11]. The complexity of dactics stems from the fact that, in application to a design tree, they:

- may be non-deterministic: a dactic will, in general, generate a forest of design trees as outcomes;
- may be recursively defined: there may be an infinite number of design trees in the forest; and
- may not terminate: so the mapping to the forest is partial.

Because of this, a dactic is a partial function from a *DState* to a finite or infinite list[12] of *DState*s:

$Dactic == DState \to DState*$

**Example 4.3.** Perhaps the simplest dactics come straight from Angel: they are **skip**, which does nothing other than pass through its argument unchanged, **fail** which always fails and **abort** whose application never terminates.

$$\textbf{skip}(plist, js) = [(plist, js)] \qquad \textbf{fail}(plist, js) = [] \qquad \textbf{abort}(plist, js) = \bot_A$$

The reader will note the use of an empty list of *DState*s to represent failure. This should be compared to the list containing a (single) complete design tree, with $[([], J)]$, and without $[([], \emptyset)]$, justifications.

**Example 4.4.** The lower production in Figure 8, recorded as a dactic, is:

$$\textbf{rule}\ \textsc{ShaRem}(Barrier, \{pos\})$$

[11] In ArcAngel derivations, under the refinement calculus, do not branch, and so their job is easier.

[12] For a set $A$, $A*$ is the set of all lists whose elements are drawn from $A$, augmented with an extra element $\bot_A$ to represent an non-terminating dactic application. For a brief recap of lists, see Appendix A.

**Example 4.5.** For the last time, we return to our farmer's problem begun in Section 2.1.1. To record the initial, 'lazy designer,' transformation we applied the DOMAIN INTERPRETATION schema. As a dactic this becomes:

$$\textbf{rule}\ \textsc{DomInt}\left( \textit{Gate},\ \textit{"Installed 3 years ago"},\ \textit{"The SGVert Model"},\ \parbox{5cm}{"The \textit{SGVert} was the only model of sluice gate produced 3 years ago. It will be easier to work with this description as a manual describing this model's operation exists."} \right)$$

## 4.4. Dactic combinations: Dacticals

To be programmatic, dactics must combine. For tactics, Angel defines *tacticals*. For design tactics, we introduce *dacticals*.

### 4.4.1. Dactic arity

*DState*s contain lists; because of this it has been found ([MNU97]) that Angel-based tactics benefit from being given arities. A dactic arity tells us to which length list it applies and to which length list it returns. Arities are indicated by superscripts: a dactic $d$ of arity $\alpha$ is written $d^\alpha$.

Arities may be complex, so that a dactic can have multiple, even infinite, arity. So, even though there is a single **skip** and **fail** for each arity $n \mapsto n$, $n \in \mathbb{N}$, the single dactic **skip** of complex arity $\{0 \mapsto 0, 1 \mapsto 1, \ldots\}$ is their equal. A dactic applied outside of its arity fails. We assume that **abort** has arity $\{n \mapsto m | n, m \in \mathbb{N}\}$.

For arities $\alpha$ and $\beta$, define:

$$\begin{array}{lcl} \alpha \mid \beta & = & \alpha \cup \beta \\ \alpha \,;\, \beta & = & \{n \mapsto m \mid n \mapsto k \in \alpha \wedge k \mapsto m \in \beta\} \\ \alpha \parallel \beta & = & \{n_1 + n_2 \mapsto m_1 + m_2 \mid n_1 \mapsto m_1 \in \alpha \wedge n_2 \mapsto m_2 \in \beta\} \end{array}$$

Dactics of complex arities are generated through dacticals which allow dactics to be combined in sequence, in choice, or in parallel. Using dactic constructors, single rules and other dactics $t_1^\alpha$ and $t_2^\beta$ can be combined in:

- sequence, $t_1; t_2$, of arity $\alpha; \beta$, in which $t_1$ is applied to the active sites of a *DState* and then applies $t_2$ to each outcome of that application, finally flattening the result. If $t_1$ or $t_2$ fail then so does the whole dactic. When the dactic succeeds, the justification required are those for both $t_1$ and $t_2$.
- choice, $t_1 \mid t_2$, of arity $\alpha \mid \beta$, in which $t_1$ is applied to the active sites first. If application succeeds, then the choice succeeds. If application fails, then $t_2$ is applied to the original active sites. Again, if $t_2$ succeeds, then the choice succeeds. Otherwise the choice fails.
- parallel, $t_1 \| t_2$, of arity $\alpha \| \beta$, which applies $t_1$ to the initial active sites and $t_2$ to those remaining.

**Example 4.6.** The design tree of Figure 8, recorded as a dactic, is:

$$\textbf{rule}\ \textsc{ShaRem}(\textit{Barrier}, \{\textit{pos}\})\ ;\ \textbf{rule}\ \textsc{DomRem}(\textit{Barrier})$$

**Example 4.7.** Using dactic choice, we may combine the effect of two designs, with the complex dactic[13]

$$\begin{array}{l} (\textbf{rule}\ \textsc{SolInt}(\textit{LaTeX}[\textit{LaTeX\_Env} : \textit{LaTeX}\ \text{version}\ 3.141592](\textit{LaTeX\_Class})) \\ \quad \mid \textbf{rule}\ \textsc{SolInt}(\textit{LaTeX}[\textit{LaTeX\_Env} : \textit{LaTeX}\ \text{version}\ 3.141592, \textit{LaTeX\_Class} : \texttt{fac.cls}]()) \\ \qquad ; \textbf{rule}\ \textsc{SolExp} \end{array}$$

whose application to the journal editor's problem $\textit{User}, \textit{Doc}, \textit{EditSys} \vdash \textit{JEReqs}$ produces the *DState* list:

$$[([\textit{User}, \textit{Doc}, \textit{LaTeX\_Env}, \textit{LaTeX\_Class} \vdash \textit{JEReqs}], \emptyset), ([], \emptyset)]$$

[13] The justifications used in Example 3.2 have been omitted for brevity.

### *4.4.2. Recursion and Pattern Matching*

Angel provides the dactic user with much pre-proven machinery. The recursion operator $\mu X \bullet f(X)$ behaves as $f$ until an $X$ is encountered, whence it behaves as $\mu X \bullet f(X)$.

**Example** Recursion gives us access to the very useful dactical *exhaust* thus:

$$\mathit{exhaust}\, t = \mu X \bullet (t; X \mid \mathbf{skip})$$

It is the dactic that will apply dactic $t$ until it applies no more, and then exit with success, through choice of the **skip**. As **skip** never fails, and so continues to be applicable, we have that *exhaust* **skip** is equivalent to **abort**. Recursive dactics need not terminate; if they do not they are equivalent to **abort**. Recursion has the potential to generate dactics of infinite complex arity.

Another useful dactical is pattern matching [GPMW96]. Pattern matching provides a level of abstraction from the detail of a dactic application by making that application dependent on the active site to which it applies. The dactical

$$\pi\, \nu_1, \ldots, \nu_n \bullet p \to d$$

binds the *meta-variables* $\nu_1$, ..., $\nu_n$ within the scope of pattern $p$ and dactic $d$ (which may contain the $\nu_i$). Applied to active site $q$, that $p$ and $q$ unify on the $\nu_i$ leads the whole expression to behave like $d$. If no match is found then it behaves like **fail**. There is no requirement of a unique match, nor need the variables unify only finite times, so that the resulting dactic may be multi-, even infinitely, aritied.

**Example 4.8.** SHARING REMOVAL requires a single shared phenomenon to be identified for removal. Although one may, for every phenomenon, write a bespoke SHARING REMOVAL application, one can use pattern matching to automate the removal:

$$\pi\, D, o, \mathit{Desc}, \mathit{Context}, \mathit{Soln}, R, \mathit{Reqt} \bullet \mathit{Context}^o, D_o : \mathit{Desc}, \mathit{Soln} \vdash R : \mathit{Reqt} \to \mathbf{rule}\ \textsc{ShaRem}(D, o)$$

will match the problem

$$\mathit{Barrier}_{pos} : \mathit{Desc}, \mathit{Op}(\mathit{safe})^{pos,cmnds}, \mathit{Press}_{drop}, \mathit{Soln}^{drop}_{cmnds} \vdash \mathit{Safe\_Op}^{safe,drop}_{cmnds}$$

applying to it the dactic **rule** $\textsc{ShaRem}(\mathit{Barrier}, \mathit{pos})$. Indeed, the pattern will match all domains in the problem, applying all possible **rule** SHARREMs and producing a list of transformed problems.

**Example 4.9.** We may define PROBLEM PROGRESSION, as discussed in Section 3.3 as a derived dactic:

$$\begin{aligned}
&\mathbf{dactic}\ \textsc{ProbProg}(\mathit{Dom})\\
&= \mathit{exhaust}(\pi\, c, \mathit{Context}, \mathit{Soln}, \mathit{Reqt} \bullet \mathit{Context}_c, \mathit{Dom}^c, \mathit{Soln} \vdash \mathit{Reqt} \to \mathbf{rule}\ \textsc{ShaRem}(\mathit{Dom}, c))\\
&\qquad ; \mathit{exhaust}(\pi\, o, \mathit{Context}, \mathit{Soln}, \mathit{Reqt} \bullet \mathit{Context}^o, \mathit{Dom}_o, \mathit{Soln} \vdash \mathit{Reqt} \to \mathbf{rule}\ \textsc{ShaRem}(\mathit{Dom}, o))\\
&\qquad\quad ; \mathbf{rule}\ \textsc{DomRem}(\mathit{Dom})
\end{aligned}$$

This systematically internalises all controlled (the first pattern) then all observed phenomena (the second pattern) from parameter domain *Dom*. Finally, when no more phenomena exist to be internalised, the dactic removes domain *Dom*.

To remove all domains from a problem $P$, i.e., to arrive at a specification problem as defined in Section 2.1.1, one need simply apply the following dactic to $\iota(P)$:

$$\begin{aligned}
&\mathbf{dactic}\ \textsc{ToSpecProb}\\
&= \mathit{exhaust}(\pi\, \mathit{Dom}, \mathit{Context}, \mathit{Soln}, \mathit{Reqt} \bullet \mathit{Context}, \mathit{Dom}, \mathit{Soln} \vdash \mathit{Reqt} \to \mathbf{rule}\ \textsc{ProbProg}(\mathit{Dom}))
\end{aligned}$$

## 5. Related Work

### 5.1. Problem Orientation

POSE is both an extension and generalisation of the fundamental ideas of Problem Frames, [Jac01], a conceptual framework for Requirements Engineering based on the problem-oriented foundation laid by the third author over a number of years. POSE has generalised Problem Frames in that all forms of software solutions are considered, not just specifications; the range of allowable POSE problem transformations goes beyond Problem Frames' problem decomposition; and problem solving is transformational, with both solutions and their adequacy arguments constructed in a step-wise fashion.

As we have explained, the structural framework of POSE exhibits no dependence on any particular language for the expression of descrptions. In other work, the first and second authors have investigated *Problem Oriented Engineering* (POE) which provides a more general notion of problem than does POSE, applicable in the wider discipline of engineering design [HR08b, RH09, HR09c, HR08a, HR09b]. POE has been used to define a general problem solving process, the POE Process Pattern, in which many different problems intertwine; these include the preparation of documentation, of assurance arguments, of training, *etc*. In [RH09] we describe its application to project manage a large e-learning project with over 50 stake-holders using another POE technology, Assurance-driven Design [HR08a]. Current work shows the use of POE for business process refactoring: the study of that work involves the capture of the design of a large organisation's 'issue fixing' process for financial transactions, and its successful re-engineering under POE to improve validation and reduce cost.

## 5.2. The use of natural language in specification

Balzer *et al* ([SB82]) aim to provide automatic tool support for the production of formal specifications from partial descriptions in natural language. The motivation is that writing fully formal specification is hard, while there are many advantages in starting with informal descriptions—they are more succinct, focus on elements of a problem which are most relevant to the customer/designer, and are better suited for human communication. By constraining the syntax of such informal descriptions to some extent, tool support can then be used to generate a formal specification though text processing techniques. Some human input may be required to eliminate ambiguities of the informal descriptions.

There are many notable contributions. Following from their earlier work, Balzer ([Bal81]) proposes the derivation of implementation programs through subsequent transformations starting from a specification, which, in a sense, is a program at a high level of abstraction—a specification expresses what has to be done, while an implementation program expresses how it is done. Interactive tool support allows the developer to guide and apply transformations. A formal specification language is used to generate the initial specification. The paper also states the importance of documenting decisions at each transformation steps, although the proposed tool only kepts a record of system states. Similarly, [Wil83] proposes an automated program transformation system to transform high-level specification into code, supported by automatic tools, with some manual intervention to guide the process. To distinguish that work from POSE, we note that the focus is firmly on the solution, i.e., in the solution domain and working with solution descriptions, while POSE gives equal value to $S$, $W$ and $R$.

The scope of that work is quite narrow: transforming an informal $S$ into formal $S$, from which the generation of a programme can then be fully automated; in our perspective, this is only one of the many forms of transformation that can be applied to $S$, $W$ and $R$ to solve a problem—specifically we would view it as a solution ($S$) interpretation transformation (illustrated in Section 3.4), justified both by tool processing and manual intervention to eliminate ambiguities (justification and rationale of such a transformation is not part of that work, of course, but an interpretation of it within POSE). On the other hand, the provision of (at least prototype) tools is ahead of our work in this paper where we focus on POSE as a conceptual framework for software engineering. It should be noted, however, that the fact that POSE is amenable to automation is a consequence of its formalisation as a Gentzen-style system, and an effort towards an automatic tool is currently underway. Similar to our approach, the work emphasises the use of informal descriptions, although with different motivations: 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.

## 5.3. 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, some supported by automatic tools, again with many notable contributions.

Feather [Fea87] 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. The implicit assumption is that all of them can be described formally—a characteristic of much of the formal methods work in the 80s. The proposed style of specification is operational, i.e., one must indicate the set of acceptable histories that the system may exhibit. Through *Gist*, a combination of generation and pruning of histories is obtained: generation provides candidate histories; pruning reduces them to a required subset based on given constraints.

There are some methodological similarities between that work and POSE: the notion of problem is closed-world in nature: we ignore any aspect of the world which is not part of the problem context. In addition, behavioural descriptions are important descriptions for problems too, as they tell us how phenomena are shared or come to exist. However, we do not assume that all descriptions should be formal, behavioural or expressed in a unique description language as the case of that work, hence that a single reasoning mechanism applies throughout. In POSE, different description languages may be used in separate parts of a problem, reflecting the fact that the (formal) solution must operate in a, by and large, non-formal physical context. In POSE, formality is in the structuring of a problem and in the manipulation which is allowed within that structure, rather than in the problem descriptions themselves—which might be formal, but need not be.

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 guarded by conditions of the nature we define in POSE are sometimes found in these formal approaches to software development, such as the transformations of specifications through to program code in the refinement calculi of Morgan [Mor94] and Back [BvW94]. We have learned from the work of [OCW03] how to define a language for the recording, replay and programmatic combination of designs.

More recently, [Smi05] has proposed another formal refinement framework, this time with a categorical basis [ML98]: specifications are finite presentations of theories; their transformations are morphisms. Software development is then mechanised formal refinement which delivers code which is correct by construction. Design reuse is achieved through a library of refinements which capture design knowledge from previous successful design.

It could be argued that the nascent Model-Driven Development (MDD) [MCF03] is also an attempt to systematise and automate code generation in a transformational way. The starting point in MDD is a system model which can be transformed automatically into a software system on a target implementation platform. The intention is for the initial model to capture knowledge of the application domain, rather than programming constructs of the target implementation platform. An essential part of the approach is then the definition of model mappings, which allow for the automated transformation of models. As for the formal methods of the 80s, the approach is predicated on the existence of a description language with well-defined syntax and semantics, which allows for the expression of computationally complete models. Currently, UML [OMG] is such a candidate language. Different to POSE, MDD assumes the existence of a unified description language in which all models are expressed, and that model transformations are fully automated once mappings are defined, hence some formal notion of soundness is needed for model mappings. Notably, MDD does not make any distinction 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.

### 5.4. 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 [BBJF98]. 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 [KW04]. We have explored the relation of POSE and safety assurance in [HMR07]. Following from that study, we have further developed and generalised ideas underlying assurance and problem engineering, arriving at the concept of Assurance-driven Design (AAD) in [HR08a, HR09a], which propose assurance and stake-holder validation as a tool to drive solution design, rather than somehow follow from it.

Strunk and Knight [SK08] have points of contact with POSE and ADD with their definition of Assurance Based Development (ABD). In ABD, a safety-critical system and its assurance case are developed in parallel, using a combination of Problem Frames and goal-structuring notation (GSN) [Kel98, KW04].

## 6. Discussion and Conclusions

As stated earlier, the aim of POSE is to bring both non-formal and formal aspects of development together in a single framework. The framework provides a structure within which the results of different development activities can be combined and reconciled. The structure is the structure of the progressive solution of a system development problem; it is also the structure of the adequacy argument that must eventually justify the developed system. Each is constructed in a step-wise manner: problems form the nodes in a tree with transformations extending the tree upwards from an initial problem description.

Modelling our framework around the concept of a Gentzen's system has many advantages. Looking at the way a proof is captured in Gentzen's systems has helped us take some steps towards trying to understand a software engineering design. For instance, although the detailed nature of the justifications that guard our rule applications still needs further investigation, it was the eigenvariable condition [Kle64] that suggested to us we could prevent unsound development steps through a condition associated with a rule application (albeit, through informal interpretation). Also, the nature of the relationship between conclusion problem and premise problems has proven useful in linking the justifications of individual steps together, placing them in their proper relationship to each other, and leading to an overall 'adequacy argument' for the development. Moreover, the trees produced by identifying equal problems capture a development in a similar way as the trees produced by identifying equal formulae capture a proof.

Inevitably, the limited presentation in this paper has not considered important aspects of POSE, in particular, in the context of real-world software development. For instance, we should stress that the choice of which problem transformation to apply at any point in a software problem development application is necessarily 'tentative' as each transformation must be justified as adequate before the step can be accepted as contributing correctly to the development. Because of this, and as in mathematical proof in Gentzen-style sequent calculi, problem transformation application does not always lead to solution; step-wise transformation is prone to the discovery of 'blind alleys', entered by a poor choice of application of a particular transformation. This reflects the nature of problem solving in that, in its general form, it is exploratory of the problem and solution spaces. All that is needed, of course, is an iterative process that, given the realisation that a particular transformation (or sequence of transformations) has lead to a dead-end (or sub-optimal solution), can backtrack past the problem branches, leaving the development in a better state.

Through POSE structures are provided that allow those involved in software engineering design to record their design and the argument that will convince the problem's stake-holders, an important aspect of real-world development. Early validation of POSE in this context comes in [HMR07, MHR07b, Man09], which illustrate the use of POSE for safety critical system development of real-world avionics systems and shows how:

- a safety case (a form of adequacy argument) can be constructed concurrently with and with influence on the development of a safe product;
- one can make explicit, with the intention of managing, development risk; and
- one can reduce the tendency to over-engineer just to be sure that the product can be validated.

It is not our intention to suggest that all software engineering design be done under POSE; that would be akin to requiring all mathematics to take place within a formal system at the level of the individual rules. However, as in mathematics where it should in principle be possible to decompose any proof to some formal checkable transformation system (whether Gentzen-style or not), we postulate that any software engineering design should, in theory, be presentable as a development under POSE. In particular, [Man09] shows how a *POSE safety pattern*—a reusable pattern of repeated steps in POSE—captures standard safety-critical design practice.

However, POSE is not intended to be a framework within which all software engineering design can and should be done. The granularity of the steps that are defined in POSE is too small to meet the requirements of industrial design. Therefore we have defined a dactic language derived from Martin's *et als.* tactics. At their most basic, dactics are intended to permit the capture of design trees, this following as a consequence of the motivation for the definition of tactics in theorem provers. By extension, they allow abstraction from (using pattern matching, for instance), the programmatic combination of captured designs (using the other dacticals), modularisation (through the definition of derived dactics) all of which enables the reuse of designs reuse within different contexts.

As with ArcAngel, in which generated proof obligations require discharge for a tactic-generated refinement to be sound, our dactics too differ from dactics: dactics applications generate a justification obligation. Rather than proof, of course, within POSE justification obligations are required to be discharged to the satisfaction of all validating stakeholders for a design to be considered adequate. An unjustified (or under-justified) dactic-derived design can therefore be unsafe in the sense that an unproven refinement is unsafe; the final step of completing the justification and obtaining validation is a necessary step after dactic application. We feel that most justifies the change of name from tactics and dactic so as to avoid confusion.

This situation is complicated because we are located in POSE, not the more general POE: because justification arguments are not software, we cannot treat the discharge of the justification obligation through POSE dactics. We note that, because the design of many different forms of artefact can be achieved in POE, we do not have this restriction and, as arguments and evidence are simply designed artefacts, their design can be addressed within POE. To emphasise the difference to this paper's definitions, instead of a design tree state being a pair consisting of a list of *Problems* and a set of *Justifications*, we can see it as:

$$POEDState == [Problem] \times \mathbb{P}Problem$$

and open the design of the discharge of the justification problems to a necessarily extended POE notion of dactic. It is our intention to extend and refine the notion of dactic to POE, given this observation.

## Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Grants and SE Validation Ltd. We thank our colleagues in the Department of Computing at The Open University for their continuing support.

## References

[Ass04] AssWS. Workshop on assurance cases: Best practices, possible obstacles and future opportunities. Florence, Italy, 2004. Co-located with the International Conference on Dependable Systems and Networks.

[Bal81] Robert Balzer. Transformation implementation: An example. *IEEE Transactions on Software Engineering*, SE-7(1):3–14, January 1981.

[BBJF98] R. Bloomfield, P. Bishop, C. Jones, and P. Froome. *ASCAD - Adelard Safety Case Development Manual*, 1998.

[BCK99] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Longman, Inc., 1999.

[Bir88] R. S. Bird. Lectures on constructive functional programming. Technical Monograph PRG-69, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK, 1988.

[BvW94] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, pages 367–384, 1994.

[Fea87] Martin S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2):198–234, April 1987.

[GJVH95] Eric Gamma, Ralph Johnson, John Vlissides, and Richard Helm. *DesignPatterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GPMW96] A. Gardiner P. Martin and J. Woodcock. A tactic calculus. *Formal Aspects of Computing*, 8(E):244–285, 1996.

[HJL$^+$02] Jon G. Hall, Michael Jackson, Robin C. Laney, Bashar Nuseibeh, and Lucia Rapanotti. Relating software requirements and architectures using problem frames. In *10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002)*, pages 137–144, Essen, Germany, 2002. IEEE Computer Society.

[HMR07] 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.

[HR08a] Jon G. Hall and Lucia Rapanotti. Assurance-driven design. In *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA 2008)*. Published by the IEEE Computer Society, 2008. Also available as Open University Computing Department Technical Report #2007/15.

[HR08b] Jon G. Hall and Lucia Rapanotti. The discipline of natural design. In *Proceedings of the Design Research Society Conference 2008*. Design Research Society, 2008.

[HR09a] Jon G. Hall and Lucia Rapanotti. Assurance-driven design, 2009. Invited from ICSEA 2008.

[HR09b] Jon G. Hall and Lucia Rapanotti. Assurance-driven development in problem oriented engieering. *International Journal On Advances in Systems and Measurements*, 2(1), 2009. To appear.

[HR09c] Jon G. Hall and Lucia Rapanotti. Requirements analysis in context with poe design. In *Selected papers from the International Workshop on Requirements Analysis*. Pearson, London, 2009. To appear.

[Jac01] Michael A. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Publishing Company, 1st edition, 2001.

[Kel98] T. P. Kelly. *Arguing safety - A systematic approach*. PhD thesis, Department of Computing, University of York, 1998.

[Kle64] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, NJ., 1964.

[KW04] Tim Kelly and Rob Weaver. The Goal Structuring Notation — a safety argument notation. In *[Ass04]*. 2004.

[LHR06] Zhi Li, Jon G. Hall, and Lucia Rapanotti. From requirements to specification: a formal perspective. In Jon G. Hall, Lucia Rapanotti, Karl Cox, and Zhi Jin, editors, *Proceedings of the 2nd International Workshop on Advances and Applications of Problem Frames*. ACM, 2006.

[Man09] Derek Mannering. *Problem Oriented Engineering of Safety-Critical Software*. PhD thesis, Open University, 2009. To appear.

[MCF03] S.J. Mellor, A.N. Clark, and T. Futagami. Model-driven development - guest editor's introduction. *IEEE Software*, 20(5):14– 18, 2003.

[MGW96] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A Tactic Calculus. *Formal Aspects of Computing*, 8(4):479–489, 1996.

[MHR07a] 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.

[MHR07b] 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.

[ML98] Saunders Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer-Verlag, 1998.

[MNU97] Andrew Martin, Ray Nickson, and Mark Utting. Improving angel's parallel operator: Gumtree's approach. Technical Report 97-15, University of Queensland, Australia, 1997.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice Hall International Series in Computer Science. Prentice-Hall International, 1994.

[MV92] Carroll Morgan and Trevor Vickers, editors. *On the Refinement Calculus.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[OC00] M. V. M. Oliveira and A. L. C. Cavalcanti. Tactics of refinement. In *Proceedings of the XIV Simposio Brasileiro de Engenharia de Software*, pages 117–132, 2000.

[OCW03] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: A Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28 – 47, 2003.

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

[ORS96] E.-R. Olderog, Anders P. Ravn, and Jens Ulrik Skakkebæk. Refining system requirements to program specifications. In *Formal Methods for Real-Time Computing*, pages 107–134. Wiley, 1996.

[RA93] F. Redmill and T. Anderson. *Safety-critical systems: current issues, techniques, and standards.* Chapman & Hall, 1993.

[RH09] Lucia Rapanotti and Jon G. Hall. Designing an online part-time master of philosophy. In *Proceedings of the Fourth International Conference on Internet and Web Applications and Services*. IEEE Press, May 24-28 2009. To appear.

[RHJN04] Lucia Rapanotti, Jon G. Hall, Michael Jackson, and Bashar Nuseibeh. Architecture-driven problem decomposition. In *12th IEEE International Conference on Requirements Engineering (RE 2004)*, pages 80–89. IEEE Computer Society, 2004.

[RHL06] Lucia Rapanotti, Jon G. Hall, and Zhi Li. Problem reduction: a systematic technique for deriving specifications from requirements. *IEE Proceedings – Software*, 153(5):183–198, 2006.

[SB82] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Commun. ACM*, 25(7):438–440, 1982.

[SJ06] Robert Seater and Daniel Jackson. Problem frame transformations: Deriving specifications from requirements. In *Proceedings of 2nd International Workshop on Advances and Applications of Problem Frames*, Shanghai, China, 2006.

[SK08] Elisabeth A. Strunk and John C. Knight. The essential synthesis of problem frames and assurance cases. *Expert Systems*, 25(1):9–27, 2008.

[Smi05] D.R. Smith. Comprehension by derivation. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 3–9. IWPC, 15-16 May 2005.

[Tur86] Wladyslaw M. Turski. And no philosophers' stone, either. *Information Processing*, 86, 1986.

[Vin90] Walter G. Vincenti. *What Engineers Know and how they know it: Analytical studies from Aeronautical History.* The Johns Hopkins University Press, 1990.

[Wil83] David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983.

## A. Lists, dactics and their operators

As in Angel, [GPMW96, Appendix B] and Gumtree [MNU97], we use a threory of lists derived from [Bir88]. To create the partial and infinite lists $A*$, we add the symbol $\perp_A$ to the set of lists over $A$. 'Partial' refers to the potential for lists to be incomplete: the partial order defined over the set with $\perp_A$ the least element, the shortest partial list. The other partial lists all end in $\perp_A$. Two partial lists compare $l_1 \leq l_2$ whenever they are equal or the first is a partial list which forms an initial sub-sequence of the second. An infinite list is the limit of a directed set of partial lists.

$+\!\!+$ is the list concatenation operator, defined as for ordinary lists, but with $l_1 +\!\!+ l_2 = l_1$ whenever $l_1$ is partial. $+\!\!+/$ represents distributed concatenation (sometimes called flatten or concat) extended to partial lists. For a total function $f : A \rightarrow B$, $f* : A* \rightarrow B*$ is the function that operates on lists by applying $f$ to each of their elements. Parallel composition is defined in terms of (a simplified, from [GPMW96]) list cross product: $([x] +\!\!+ l_1) \times ([y] +\!\!+ l_2) = [x +\!\!+ y] +\!\!+ (l_1 \times l_2)$.

For dactics $d$, $d_i$ of arity $n_i \mapsto m_i$, $i = 1, 2$, and a design state $ds$, we make the following definitions, c.f., [GPMW96]:

- sequence:

$$(d_1; d_2)\ ds = (+\!\!+/ \cdot d_2 * \cdot d_1)\ ds$$

- choice:

$$(d_1 \mid d_2)\ ds = d_1\ ds\ +\!\!+\ d_2\ ds$$

- parallel composition:

$$(d_1 \parallel d_2)\ ds = \underset{ds_1 \mathbin{+\!\!+} ds_2 = ds}{\mathbin{+\!\!+}/}\ d_1\ ds_1 \times d_2\ ds_2$$

- recursion is defined as a fixed point:

$$(\mu X \bullet d(X))\ ds = \bigsqcup_{i \in \mathbb{N}} (d^i(\textbf{abort})\ ds)$$

- pattern matching:

$$(\pi\, \nu_1, \ldots, \nu_n \bullet p \rightarrow d)\ ds = \mathbin{+\!\!+}/[d[\nu_i/p_i]\ ds \mid \nu_i \text{ matches } p_i \text{ in } p]$$

## B. On Arity

We have simplified the original list-based representation of arity to be set based. Technically, this change requires us to have the following result:

**Definition B.1.** For any arity $\alpha$, define $set(\alpha) = \{a | a \in \alpha\}$. Define the relation $ar : dactic \rightarrow dactic$ such that $d_1\ ar\ d_2$ if and only if $set(arity(d_1)) = set(arity(d_2))$.

**Conjecture B.2.** *ar* is a congruence relation between dactics.

Work is currently in progress on the proof of this conjecture by structural inductaion: most of the cases are simple, only the nature of partial and infinite lists complicates the result for parallel and choice. However, even if the conjecture *does not* hold, it will be sufficient to return to Angel's definitions, at the cost of complicating the definitions of the operators between arities defined in Section 4.4.1, but otherwise leaving our adaptations unaffected.