ISSN 1744-1986

Technical Report N^{o} 2005/05

A framework for software problem analysis

Jon G. Hall
Lucia Rapanotti

7th April 2005

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

http://computing.open.ac.uk

A framework for software problem analysis

Jon G. Hall Lucia Rapanotti
Computing Research Centre
The Open University

ABSTRACT

The paper introduces a software problem calculus based on a view of requirements engineering proposed by Zave and Jackson, that we hope can underpin techniques and processes in requirements engineering and early software design. The approach is centred around the notion of problem and problem transformation. It is propositional in nature in that it allows for heterogeneous problem descriptions and correctness argumentation for the validation and verification of solutions. We present a number of fundamental rules to populate the calculus including 'divide-and-conquer' problem decomposition, description reification, prior-solution reuse, the use of architectures for solution structuring. In addition, and central to the foundational nature of the calculus is the interpretation and subsequent incorporation into our framework of others' work as problem transformations. This is an on-going task: examples presented in this paper include interpretations of simple goal-oriented decompositions and viewpoints. The calculus also provides rich traceability of the requirements of an original problem through transformation to those of its solution. We use the design of a sluice gate as a simple running example throughout the paper.

1. INTRODUCTION

The purpose of this paper is to define a calculus for requirements engineering based on the notion of software problem and software problem transformation. Software problems take many forms. For us a software problem is a contextualised need that has a software solution. This takes its inspiration from Jackson's work on Problem Frames (PF) [21], although its tradition goes back to, at least, Polya [29].

Our software problem calculus embodies the standards for requirements engineering established in The Four Dark Corners paper of Zave and Jackson [42]. These standards, quoted or paraphrased in italics below, are based on the separation of the machine , i.e., the computational artefact, and the environment for which the machine is to be built, expressed as:

- *The terminology used in requirements engineering should be grounded in the reality of the environment for which a machine is to be built.* Our calculus allows for description grounded in the language of the domain — in particular, it does not prescribe a single description language in which designations are expressed. In this sense our calculus is propositional in nature: it operates above the language of the domain.

- *It is not necessary or desirable to describe (however abstractly) the machine to be built. Rather, the environment is described in two ways: as it would be without or in spite of the machine (the indicative mood), and as we hope it will become because of the machine (the optative mood).* Our calculus similarly separates indicative and optative descriptions, so that we focus on the requirements and domains rather than the solution.

- *It is essential to identify which phenomena are controlled by the environment, which phenomena are controlled by the machine, which phenomena of the environment are shared with the machine, and which phenomena are constrained or referred to by the requirement.* This separation is an important part of our calculus: a solution machine must correctly control and observe the phenomena that its environment exposes; our calculus assumes that each phenomena has a single controller (either in the environment or the machine); the aim of the machine is to satisfy the constraints expressed in the requirements.

- *The primary role of domain knowledge in requirements engineering is in supporting refinement of requirements to implementable specifications. Correct specifications, in conjunction with appropriate domain knowledge, imply the satisfaction of the requirements.* Our calculus provides problem transformations for requirements refinement in order to bridge the gap to implementable specifications.

Our calculus takes a proof-theoretic form, as is found in the work of Gentzen [22, 28], in using semantics preserving transformations on sequents, a sequent being a problem statement, i.e., a syntactic characterisation of a problem. Manipulation of problems, i.e., the relation of problems to problems, occurs through the application of rules in the calculus. Rules are defined to relate problems-with-solutions to problems-with-solutions by relating their components, i.e., their context, requirement, and solution form. In addition, in parallel with the development of a solution of a problem, a correctness argument is created that justifies its solutionhood.

The paper is structured as follows...

2. RELATED WORK

A calculus for requirements engineering to the best of our knowledge is missing from the literature. However, a reference model for requirements and specifications which shares the Four Dark Corners' foundations with this work was defined in [14], and extended by Hall and Rapanotti in [15]. The focus of the reference model is the formal characterisation of the meaning of requirements and specifications and their relationship as well as what it means for requirements engineering to be complete. Instead our calculus provides a framework for the refinement of requirements into specifications. The notion of software problem is not included in the reference model, while it is the focus of our calculus. Based on the reference model, REVEAL [17] is a process for requirements engineering based on the disciplined use and separation of domain, requirements and specifications descriptions and the discharge of correctness arguments for validation and rich traceability of requirements. In being based on the reference model, REVEAL has no explicit notion of problem and problem transformation, although it is possible to give an interpretation of some of its process steps as problem transformations within our framework. Being rooted in practice, REVEAL has provided extensive validation of the reference model for practical industrial requirements engineering.

Problems-oriented requirements engineering was proposed by Jackson in his Problem Frames approach, [20, 21]. (A roadmap of Problem Frame-related research appears in [9].) Problem Frames appear similarly motivated by [42] and provide a graphical notation for representing and classifying software problems. They also identify two basic types of problem transformatons, projection and progression. They do not, however, formally characterise problems not do they formally relate problems one to another as we do in our calculus. In the paper, we will discuss how problem frames are properly captured in our calculus.

Formal characterisation of problems have appeared as proposed semantics of Problem Frames. Hall *et al.* forge a formal relationship between Problem Frames and the reference model in [16]. Work by Bjorner ([3]) characterises the properties of some early problem frames in terms of typical types of domains and requirements, and for two of them, the design of the solution machine. [5] takes this approach further, providing a formalisation of the Translation frame and Information System frame using CASL (the Common Algebraic Specification Language, [7]). [10] proposes a semantic approach which uses UML as a means of describing a meta-model as a semantics for problem frames that might be used to insert a notion of problem into the UML. The meta-model describes context diagrams, problem diagrams and problem frames, but does not capture the meaning of other elements, such as correctness of a machine specification with respect to its environment and requirements. None of these approaches to a semantics of problem frames address the formal relation between problems or the formal refinement of requirements into specifications.

Parallel to problem-based approaches, a range of goal-oriented approaches to requirements engineering have been proposed which are based on the notions of goal, task and agent. Transformations are then goal refinements and decompositions as well as the assignment of a task to agents to satisfy goals. Goal-based and problem-based approaches can be seen as complementary, as demonstrated, for instance, by the work of Bleistein et al, [4] on combining i* [41, 40] and Problem Frames. As we will see, in our calculus goal decomposition can be captured as a type of requirement refinement, under the interpretation of goals as requirements.

Parallel to problem-based approaches are the scenario-based approach to requirements engineering, such as Volere [31] and Use-cases [6]. These are usually predicated on the identification of business processes within an oganisation, which need some form of automation. Relevant business rules that governs such processes need be identified together with their boundary, the external business events to which they react, and the effect of such events. Scenarios are then used to identify the technical requirements that the automated system should satisfy. In such approached the notion of problem is missing, and they tend to be informal to a large extent. However, by interpreting business processes and their boundary as a problem context, scenario-based techniques for refining requirements can be seen as a form of problem progression - one of the identified types of problem transformation in our framework.

3. PROBLEMS, SOLUTIONS

For us, a software problem is a requirement in a real-world context. The context is a collection of interacting domains, described by their indicative properties; the requirement is an optative property that we would like to be true of the context.

For the requirement R and real-world context $W = \{D1, ..., Dn\}$ each domain in the context will expose various phenomena — events, commands, states, entities, *etc.* — that it controls and which can be observed by another domain in the context; the domain will also be able to observe exposed phenomena from other domains in the context. In addition, the domain may have internal phenomena which, although unobservable to other domains, can be constrained or referred to in the requirement. That D controls the phenomena set c, observes the phenomena set o, and has internal state d will be written[1] $D(d)^c_o$. There are simple correctness conditions on these sets, *viz.*: they should be pairwise disjoint. That the requirement R constrains phenomena set c and refers to phenomena set r is written R^c_r, again c and r should be disjoint. A phenomenon has precisely one controller. A phenomenon is said to be *machine shareable* when it can be either controlled or observed by a machine; for instance, the temperature of a room is not machine shareable, the output of a temperature sensor could be machine shareable.

The control-observes links between domains provide a software problem with a topology: domains D^c and E_o are linked if $c \cap o \neq \emptyset$. Similarly, the requirement can be incorporated into the topology if, with domain $D(d)^c_o$, it constrains or refers to a phenomenon in $d \cup c \cup o$. The linking of domains imbues the context with controls and observes sets: for the $W = \{D1, ..., Dn\}$ above, W^c_o when $Di^{c_i}_{o_i}$, and c (respectively o) are the machine shareable phenomena in $\bigcup c_i$ (respectively $\bigcup o_i$).

A software problem challenges us to find a solution that, in the context, brings about the requirement. Typically, a machine will be able to observe or control a proper subset of those phenomena exposed by a problem's context, i.e., the machine shareable phenomena. The machine must observe and control the context through these phenomena, so to have it satisfy the requirement, i.e., solve the software problem.

Formally, for $W^c_o = \{D1, ..., Dn\}$, a *pre-solution* $\mathcal{S}$ to a software problem must fit within the problem context, i.e., $\mathcal{S}^o_c$. In our calculus, that $\mathcal{S}$ is a pre-solution to a software problem with context W and requirement R is indicated by writing (the sequent):

$$W, \mathcal{S} \vdash R$$

(To be able to distinguish the pre-solution, it will always be written closest to the $\vdash$ symbol, on the left.) As already mentioned, sequents are syntactic expressions, manipulated by rule application.

A pre-solution is a *solution* to a software problem if it is accompanied by a correctness argument $\mathcal{CA}$ which justifies, to an appropriate level of conviction, its solution status. There are many levels of conviction to which a correctness argument may be justified; we turn to the literature and practice for some examples: if (almost) ab-

[1]We will often omit a superscript or subscripted if these are clear by context, or if we are not concerned with them.

solute conviction is required we could require a fully formal proof of solutionhood; many projects work to a level of conviction provided by testing; another is the development of safety cases that satisfy a safety authority; perhaps the weakest is simple trust. It is important to note, therefore, that the correctness argument is not necessarily a statement requiring formal proof of correctness: its role is to justify a solution, not to remove doubt.

A correctness argument distinguishes a pre-solution — i.e., something that interacts with a problem context in the correct way, but perhaps not able to achieve the requirement — from a solution. As such it is a very important component of our framework. Later in this paper we will show how a correctness argument for a particular solution can be defined incrementally through problem transformation.

Although a slight abuse of notation, it will be convenient to be able to refer to the triple $W, \mathcal{S} \vdash R$ as a problem.

3.1 Example

As an example, we consider the software problem of designing a sluice gate controller, based on [21, 23]. Finding a solution to this problem will be a goal of this paper; it will also illustrate some of the aspects and properties of our software problem calculus. The problem is simple, but not so simple as to be trivial as we shall see. The reader should imagine a customer in the market for a sluice gate controller, from whom the requirements engineer is gathering requirements.

From the literature, we know that the relationship between customer and requirements engineer is based on communication but that this communication is not always smooth — it will sometimes be the case that misunderstandings can occur because of no shared language, because of unwritten assumptions by either side, *etc*, so that iteration between the parties is needed to gain clarification. To us, this is all part of the problem solving process, and so should not be ignored by the calculus; the example will give a flavour of how this is handled.

The given domain descriptions for this problem, those that together form W are:

$\mathbf{G}$ The Gate, when stopped, reacts to (i.e., observes) a $\langle \mathit{Clockw}, \mathit{On} \rangle$ event sequence by moving upwards, unless already fully open. When stopped, it reacts to (observes) an $\langle \mathit{AntiClW}, \mathit{On} \rangle$ event sequence by moving downwards, unless already fully closed. When it receives (observes) an *Off* event it turns its motor off, stopping any motion. When it detects that it is fully open it generates a Top event (that it controls). When it detects that it is fully closed it generates a Bottom event (that it similarly controls).

$\mathbf{Op}$ The operator generates (controls) Raise commands to request that the gate moves upwards, generates (controls) Lower commands to request that the gate move downwards, and generates (controls) Stop commands to request that the gate stops.

From the description, the phenomena are[2]

$$\begin{array}{llll} a: & \{\mathit{Raise}, \mathit{Lower}, \mathit{Stop}\} & b: & \{\mathit{Top}, \mathit{Bottom}\} \\ c: & \{\mathit{ClockW}, \mathit{AntiClW}, \mathit{On}, \mathit{Off}\} & d: & \{\mathit{Open}, \mathit{Shut}\} \end{array}$$

with a controlled by the operator, b (respectively, c, d) controlled by (respectively, observed by, an internal state of) the gate.

We will assume that the initial requirement description for gate operation as gathered from our customer consists of two statements $\mathbf{TGO}$ and $\mathbf{MGO}$, apparently intended as a conjunction:

[2]The phenomena are presented as named sets, which facilitates their repeated reference.

$\mathbf{TGO}$ The gate should be Open only for the first ten minutes of each three-hour period.

$\mathbf{MGO}$ The gate should be either Open or Closed consistent with the Operator's commands to Raise, Stop or Lower.

Taken as a simple conjunction, $\mathbf{TGO}$ and $\mathbf{MGO}$ sometimes conflict and so are not always satisfiable: consider what should happen during the first ten minutes of a three-hour period, if the operator commands the gate to Lower. Our first task in the calculus will be to clarify what the customer means.

With pre-solution $\mathbf{C}$, even if unsatisfiable, we may represent this software problem as:

$$\mathbf{G}(d)^b_c, \mathbf{Op}^a_\emptyset, \mathbf{C}^c_{a,b} \vdash (\mathbf{TGO} \textit{ and } \mathbf{MGO})^d_a$$

4. PROBLEM TRANSFORMATION

It is often the case that by solving one problem we are led to the solution of another. There are many examples: from mathematics, a solution to Fermat's equation — that of the famous last theorem — leads to a contraction with the Taniyama-Shimura conjecture [33]; from software engineering, software problems are split up under functional decomposition, object-orientation, *etc* to be able to solve them; in addition, solving a software problem in a test harness often leads (perhaps not directly) to a solution in the real-world.

There are many benefits of transforming one problem into another:

- one may be easier to solve; it may be stated in a way that makes it more amenable to solution;
- one may already have been solved;
- one may be more detailed, specific and/or correct;
- there may be specialists who know how to solve the transformed problem;
- *etc*.

A problem transformation should, to be useful, be accompanied by two related artefacts:

- a transformation that relates the solutions of the transformed problem (if and when found) to that of the original — in the case of a test harness, it may be that the same solution may be used in the final environment, most likely with extra testing;
- an argument that justifies the transformation as correct — we should be convinced that correctness of the solution in the transformed problem leads to a correctness of the transformed solution in the original problem.

Rules in our calculus transform problem to problems, and include each of these artefacts. Formally, a rule in our calculus is an instance and/or specialisation of the following rule. Suppose we have problems $W, S \vdash R$, $Wi, Si \vdash Ri$, $i = 1, ..., n$, then we will write

$$\frac{W1, S1 \vdash R1 \qquad ... \qquad Wn, Sn \vdash Rn}{W, S \vdash R} \mathcal{CA}$$

to mean that, if $\mathcal{CA}$ holds then S is a solution of $W, S \vdash R$ whenever $S1, ..., Sn$ are solutions of $W1, S1 \vdash R1, ..., Wn, Sn \vdash Rn$, respectively. Typically, $\mathcal{CA}$ will contain a clause that relates the form of the solution S to the original problem to those of the transformed problems $S1, ..., Sn$ — this is the relation of solution of the transformed problems to the solution of the original.

Similarly, there will be similar clauses relating context and requirements.

The remainder of $\mathcal{CA}$ is used to justify the transformation is correct; i.e., $\mathcal{CA}$ is a step in a correctness argument for the solution, specifically, $\mathcal{CA}$ extends the transformed problems' correctness arguments to be that of the original problem.

5. CLASSES OF PROBLEM TRANSFORMATION

In this section, we propose a number of classes of problem transformation. These are, variously, encodings, extensions and generalisations of problem transformations found in the literature, as well as simpler and more complex transformations that we identify and introduce in this paper.

In our example, the first step we will take is to clarify how the conflicting statements $\mathbf{MGO}$ and $\mathbf{TGO}$ should form a satisfiable customer requirement. We do this by applying the Domain and Requirement description reification rule:

5.1 Domain and Requirement description reification

A rich source of transformation is the reification of domain descriptions, i.e., taking an initial less precise description — natural language say — as captured from the customer, and re-expressing it more precisely — as a statechart description of a real-world domain, say. This is the domain modelling step. As the needs of development change, as we approach a solution through problem transformation, natural language changes into other, more precise, descriptions.

Formally, domain and requirements description reification is captured by the following rule:

$$\frac{W, D', S \vdash R'}{W, D, S \vdash R} \quad \textit{D' more precise than D and R' more precise than R}$$

in which W stands for the parts of the context other than the reified domain

The reader will note that the correctness argument is not expressed formally. Although a more formal expression might be appropriate in a particular development environment — if Z [**?**] was being used as the development language, for instance — it would be a mistake to be too prescriptive about the level of conviction needed in the rules expression as we want to be able to allow both formal and informal forms of description reification.

5.1.1 Example

In consultation with the customer, we have been able to clarify that that operator should be able to override the timed gate operation, i.e., the customer intended $\mathbf{TGO}$ *unless* $\mathbf{MGO}$. As a requirement reification, this can be recorded as:

$$\frac{\mathbf{G}(d)^b_c, \mathbf{Op}^a_\emptyset, \mathbf{C}^c_{a,b} \vdash (\mathbf{TGO}\ \textit{unless}\ \mathbf{MGO})^d_a}{\mathbf{G}(d)^b_c, \mathbf{Op}^a_\emptyset, \mathbf{C}^c_{a,b} \vdash (\mathbf{TGO}\ \textit{and}\ \mathbf{MGO})^d_a} \quad \textit{Customer intent}$$

(where *Customer intent* indicates that *unless* is the customer's intended meaning for *and*). In the sequel, we will refer to $\mathbf{TGO}$ *unless* $\mathbf{MGO}$ as $\mathbf{GO}$.

5.2 Problem Projection

Under problem projection, a number of sub-problems are formed, each of which is intended to solve some small part of the original problem. Problem projection is described as:

$$\frac{W1, S1 \vdash R1 \quad \ldots \quad Wn, Sn \vdash Rn}{W, \mathbf{C}(S1, \ldots, Sn) \vdash R1 \wedge \ldots \wedge Rn} \quad \textit{$R1, \ldots, Rn$ not in conflict and $\mathbf{C}$ a linearizable combinator}$$

where $\bigcup Wi = W$. That $\mathbf{C}$ is a linearizable combinator means that it places the Si in parallel and ensures that their combined effect is linearizable [18]; specifically, it ensures there is no destructive interference between the sub-problem solutions; see for instance [21] for a discussion. When the Si do not interfere, $\mathbf{C}$ can be simple parallel composition. Problem projection includes — indeed, was inspired by — the notion of problem projection of Problem Frames as defined in [21].

It is worth restating that, although the correctness argument may be capable of formal discharge — that $R1 \wedge \ldots \wedge Rn$ is never false can be proven by considering its truth value at each time point — for instance, there is no requirement in our calculus to do so. Of course, without a formal proof, there will be no certainty that the rule produces a correct transformation, but this matches the case in software engineering: without proof, one cannot know a piece of software is correct, but most software cannot be proven correct, and yet is still written.

5.2.1 Example

Consider again the gate problem: as $\mathbf{TGO}$ and $\mathbf{MGO}$ sometimes conflict, they do not allow problem projection to be applied directly. However, writing $R1$ as "$\mathbf{TGO}$ when no operator command" and $R2$ as "$\mathbf{MGO}$" then $\mathbf{GO} \iff R1 \wedge R2$ so that the problem projection rule may be applied leading to

$$\frac{\mathbf{G}(d)^b_c, \mathbf{Op}^a_\emptyset, \mathbf{C1}^c_{a,b} \vdash R1^d_a \qquad \mathbf{G}(d)^b_c, \mathbf{Op}^a_\emptyset, \mathbf{C2}^c_b \vdash R2^d_a}{\mathbf{G}(d)^b_c, \mathbf{Op}^a_\emptyset, \mathbf{C}(\mathbf{C1}, \mathbf{C2})^c_{a,b} \vdash \mathbf{GO}^d_a} \quad \textit{R_1, R_2 not in conflict and $\mathbf{C}$ a linearizable combinator}$$

for some linearizable combinator $\mathbf{C}$, not specifically determined by this rule. Indeed, there are many forms that $\mathbf{C}$ can take; we will use one of them to illustrate the use of architectures in the calculus.

5.3 Architectural expansion

The approach upon which our calculus is based emphasise the distinction between environment and system spaces, i.e., between the problem and solution domains. Other approaches, such as the Problem Frames approach, intend their user to focus their attention on the problem space, practically ignoring the solution space until the problem is wholly understood. While undeniably a useful separation of concerns for requirements engineering, we feel that this separation should not be interpreted to mean that a software development is a linear process from problem to solution: first requirements engineering, then software development as there are many characteristics of modern software engineering practice that would contradict this. Firstly, even modestly complex problems can force problem owner and solution engineer into negotiation over trade-offs and consideration of details of the solution [1]. Secondly, in practice, the development of new systems is very rarely green-field: new software is usually developed from existing components [19] or within existing frameworks [11] and architectures [32]. Finally, expert workers in well-known application domains express their expertise through development, even for bespoke software. Therefore our calculus includes problem transformations which allow solution space knowledge to be used to inform the problem analysis for new software developments.

An architecture is assumed to combine a number of extant components and components yet to be found in a topology. An architecture may also be named. This is not unlike approaches to architecture in the literature: a mature example being [2]. The representation of an architecture in our calculus is as follows:

$$Name[C1, ..., Cm](D1, ..., Dn)^c_o$$

where $Name$: is the name of the architecture ($Arch$ if it is anonymous), Ci are existing components, and the Di remain to be designed.

For correctness, for components $Ci^{c_i}_{o_i}$ and $Dj^{d_j}_{p_j}$, the architecture must satisfy the following:

- $o_i \cap c_i = \emptyset = d_j \cap p_j$, for all i, j;
- the c_i and d_j are pairwise disjoint;
- $c \subseteq (\bigcup c_i \cup \bigcup d_j)$;
- $o \subseteq (\bigcup o_i \cup \bigcup p_j)$.

These ensure that the architecture contributes correctly to the solution's interaction with the environment.

The rule assumes that a software solution will be structured according to a particular architecture. The *architectural expansion* rule therefore exposes the extant components Ci to the environment, and focusses the problem on the domains Dj that remain to be designed as follows:

$$\frac{\begin{array}{r} W, C1, ..., Cm, D2, ..., Dn, D1 \vdash R \\ ...\ W, C1, ..., Cm, D1, ..., Dj-1, Dj+1, .., Dn, Dj \vdash R\ ... \\ W, C1, ..., Cm, D1, ..., Dn-1, Dn \vdash R \end{array}}{W, Name[C1, ..., Cm](D1, ..., Dn) \vdash R}$$

To clarify the rule's form, focus on $D1$, the domain that forms the subject of design in the first sub-problem in the conclusion of the rule. The new context for $D1$ consists of all designed components Ci, together with all of the other designable components, *each of which is considered as an already designed domain*[3]. Clearly, there are assumptions being made about the other Di. The correctness argument, other than requiring the solution to be of the form of the architecture ($S^c_o = Name[C1, ..., Cm](D1, ..., Dn)^c_o$), is identically true: one can always apply an architecture. It is therefore omitted for brevity.

The complexity of the rule is due by the very general transformation that is implied by an architecture. We will see ways on managing this complexity in the sequel.

It is also notable that only one of the Di is the subject of design in any sub-problem, whereas we are supposed to be determining the design of the whole collection of Dis to solve the problem. Thus the solution components need to be designed together by growing the trees of the individual sub-problems upwards together. We illustrate a simple form of co-design in this example.

5.3.1 Example

We now use a simple architecture to structure the solution for the gate controller. It has one extant component — a mediator — and two to-be-designed components, corresponding to $\mathbf{MGO}$ and $\mathbf{TGO}$ respectively.

The architectural artefact is defined as

$$Arch[\mathbf{Med}(a')^{a_1,c}_{a,b,c_1,c_2}](\mathbf{C1}^{c_1}_{\emptyset}, \mathbf{C2}^{c_2}_{a_1})^c_{a,b}$$

in which the mediator has description:

$\mathbf{Med}$ *commandIssued* is initially false. On receiving a command from the operator, set *commandIssued* to true and relay the command to $\mathbf{C}2$. On receiving an event sequence from $\mathbf{C}1$, if *commandIssued* is false, then issue that event sequence, otherwise ignore it. On receiving an event sequence from $\mathbf{C}2$, if *commandIssued* is true, then issue that event sequence and set *commandIssued* to false, otherwise ignore it.

Note that the mediator gives priority to the operator's commands rather than always imposing the regime that the gate should be open the first ten minutes in every three hours.

Application of the architectural expansion rule to the gate problem leads to two sub-problems:

$$\frac{\begin{array}{r} \mathbf{G}(d)^b_c, \mathbf{Op}^a_{\emptyset}, \mathbf{Med}(a')^{a_1,c}_{a,b,c_1,c_2}, \mathbf{C2}^{c_2}_{a_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{GO}^d_a \\ \mathbf{G}(d)^b_c, \mathbf{Op}^a_{\emptyset}, \mathbf{Med}(a')^{a_1,c}_{a,b,c_1,c_2}, \mathbf{C1}^{c_1}_{\emptyset}, \mathbf{C2}^{c_2}_{a_1} \vdash \mathbf{GO}^d_a \end{array}}{\mathbf{G}(d)^b_c, \mathbf{Op}^a_{\emptyset}, Arch[\mathbf{Med}(a')^{a_1,c}_{a,b,c_1,c_2}](\mathbf{C1}^{c_1}_{\emptyset}, C2^{c_2}_{a_1})^c_{a,b} \vdash \mathbf{GO}^d_a}$$

where

$$\begin{array}{llll} a: & \{Raise, Lower, Stop\} & a': & \{commandIssued\} \\ b: & \{Top, Bottom\} & d: & \{Open, Shut\} \end{array}$$

$$\begin{array}{ll} c: & \{ClockW, AntiClW, On, Off\} \\ a1: & \{\mathbf{Med}.Raise, \mathbf{Med}.Lower, \mathbf{Med}.Stop\} \\ c1: & \{\mathbf{C}1.ClockW, \mathbf{C}1.AntiClW, \mathbf{C}1.On, \mathbf{C}1.Off\} \\ c2: & \{\mathbf{C}2.ClockW, \mathbf{C}2.AntiClW, \mathbf{C}2.On, \mathbf{C}2.Off\} \end{array}$$

From the architectural expansion rule application, we have derived two sub-problems: these represent the problems of co-designing $\mathbf{C}_1$ and $\mathbf{C}_2$ in the presence of the given mediator.

Architectural expansion uncovers solution structure, making it part of the problem context. However, the transformation is very regular: each sub-problem is very similar. To be able to leverage the new solution structure to solve the problem, we need to focus on the detail of the newly introduced structure. We may do this by problem progression.

5.4 Problem progression

The progression of problems is inspired by a figure in Jackson (2001). The use of progression is to bring requirements 'far into the world' closer to the machine. The idea behind problem progression is to remove a domain, compensating for its effect by changing the requirement. This transformation is arrived at iteratively, so to cope with the many situations in which the removed domain can influence the behaviour of other domains. Moreover, the transformations transform the requirement, and so in most cases there will be no precise form of the transformation — recognising the transformations is left to the expertise of the developer[4].

There are a number of progression rules, some which transform the requirement, others that transform the problem's context.

5.4.1 Requirements progression

The first form of the progression rule applies when the requirement constrains an internal phenomenon of D, brought about by domain $D1$ issuing a:

$$\frac{W, D1^a, D(b)_a, S \vdash R1^a}{W, D1^a, D(b)_a, S \vdash R^b}\ a \rightsquigarrow b$$

Here, the requirement R which constrains an internal phenomenon b of domain D is re-expressed in terms of a requirement on another domain's phenomena (a) that makes b come about; statements in

[3]Remember that, by convention, the subject of design is the single domain closest on the left to the $\vdash$ symbol

[4]A fully formal version of problem progression exists, when the description language is CSP, based on Lai's quotient operator, [24].

R of the form 'bring about b in D' are replaced in R with the statement '$D1$ issues a'. The correctness argument for this transformation requires a causal relationship between a and b, $a \rightsquigarrow b$, to be established.

The second rule applies when a domain $D1$ observes a phenomenon from controlling domain D. If the observed phenomenon leads casually to an internal phenomenon of $D1$, then reference to the observed phenomenon can be removed from R to become a constrained or referenced internal phenomenon:

$$\frac{W, D1(a)_b, D^b, S \vdash R1^a}{W, D1(a)_b, D^b, S \vdash R^b}\ b \rightsquigarrow a$$

Here, the requirement R is re-expressed in terms of the requirements on an internal state of domain $D1$ rather than on $D1$'s observation of b.

5.4.2 Domain removal

Problem progression reduces a domain's involvement in a problem. Having progressed a requirement R using the above rules, it may be that mention of domain D is no longer made in the requirement, although it may still be connected in the problem's context. Unfortunately, such a domain cannot trivially be removed from the problem: it may still influence the problem. Consider the following simple, slightly contrived, example; there are three domains:

- *Press*, a massive metal panel press (as used, perhaps, in a car factory for shaping car bodywork panels);
- an operator *Operator* who controls the press
- a safety barrier, SB, that prevents the operator being in the vicinity of the press during operation.

The requirement is *Safe Press Operation* (which, for the sake of argument, we assume does not mention the safety barrier). The problem is to build a controller *Controller* for the press. The presence of the safety barrier means that the controller does not need to ensure that the operator is sufficiently far from the press to meet the requirement. Removing the safety barrier domain from the problem, even though it is not mentioned in the requirement, changes the problem to one in which the position of the operator must be monitored.

Rather, to be able to remove the domain whilst preserving solutions, we must re-express the problem's requirement so that an assumption — equivalent to the operation of the domain we wish to remove — is built in. In this case, this would lead to the requirement 'Assuming the operator is sufficiently distant from the *Press* during operation, *Safe Press Operation*'.

This finds expression as:

$$\frac{W, D1, S \vdash R1^e_f}{W, D1_c, D(d)^c, S \vdash R^e_f}\ (e \cup f) \cap (d \cup c) = \emptyset$$

where $R1$ is 'Assuming D's behaviour, R.' Note, that this does not require D to be named in the requirement, only that the behaviour of D, i.e., what it does and what it prevents from happening, are assumed in the requirement.

As the motivation behind progression is to remove a domain from the problem context, it will often be the case that requirement progression and domain removal are combined in a single transformation, as is the case in our running example.

5.4.3 Example

From the sub-problems that we derive from the application of the architecture expansion rule, we want to derive specific technical requirements for the specification of components $\mathbf{C}1$ and $\mathbf{C}2$. We will focus on the first sub-problem, that in which we design $\mathbf{C}1$.

$\mathbf{GO}$ refers to commands issued by the operator $\mathbf{Op}$; applying the second progression rule we may instead rewrite $\mathbf{GO}$ in terms of the commands that the mediator $\mathbf{Med}$ receives, and follow it with the remove domain rule to progress away the operator. Similarly, the gate $\mathbf{G}$ can be progressed away leaving the mediator to choose between the components. Finally, the second component can be removed, by moving its behaviour into the requirement. Assuming the same phenomena sets as before, formally we have:

$$\cfrac{\cfrac{\cfrac{\mathbf{Med}(a')^{a_1,c}_{c_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{MedTC2}^{c}_{a'}}{\mathbf{Med}(a')^{a_1,c}_{c_1,c_2}\mathbf{C2}^{c_2}_{a_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{MedTC}^{c}_{a'}}\ \mathcal{CA}(3)}{\mathbf{G}(d)^b_c, \mathbf{Med}(a')^{a_1,c}_{b,c_1,c_2}, \mathbf{C2}^{c_2}_{a_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{MedGO}^{d}_{a'}}\ \mathcal{CA}(2)}{\mathbf{G}(d)^b_c, \mathbf{Op}^a_{\emptyset}, \mathbf{Med}(a')^{a_1,c}_{a,b,c_1,c_2}, \mathbf{C2}^{c_2}_{a_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{GO}^d_a}\ \mathcal{CA}(1)$$

in which

$\mathbf{MedGO}$ (Mediated gate operation) The gate should be *Open* for the first ten minutes of each three-hour period, unless the mediator's *commandIssued* is *true*, in which case the gate should be *Open* or *Shut* consistent with the mediator's commands (one of $\langle$*Off*, *ClockW*, *On*$\rangle$, $\langle$*Off*, *AntiClW*, *On*$\rangle$ and $\langle$*Off*$\rangle$).

$\mathcal{CA}(1)$ an operator's commands causes *commandIssued* to become true, a causal relationship. In addition, $(d \cup a') \cap a = \emptyset$.

$\mathbf{MedTC}$ (Mediated timed control) When *commandIssued* is *false* the mediator should issue $\langle$*Off*, *ClockW*, *On*$\rangle$ at the beginning of a three-hour period, followed by $\langle$*Off*, *AntiClW*, *On*$\rangle$ ten minutes later;

$\mathcal{CA}(2)$ the mediator's command sequence $\langle$*Off*, *ClockW*, *On*$\rangle$ (respectively, $\langle$*Off*, *AntiClW*, *On*$\rangle$) causes the gate to become *Open* (respectively, *Shut*), a causal relationship. In addition, $(c \cup a') \cap (d \cup b) = \emptyset$.

$\mathbf{MedTC2}$ (Mediated timed control, without $\mathbf{C}2$) Assuming $\mathbf{C}2$'s behaviour, when *commandIssued* is *false* the mediator should issue $\langle$*Off*, *ClockW*, *On*$\rangle$ at the beginning of a three-hour period, followed by $\langle$*Off*, *AntiClW*, *On*$\rangle$ ten minutes later;

$\mathcal{CA}(3)$ $(c \cup a') \cap c_2 = \emptyset$.

The behaviour of $\mathbf{C}2$, as assumed in $\mathbf{MedTC2}$, is not accessed in the requirement as we work under the assumption that *command Issued* $=$ *false*. Hence, $\mathbf{MedTC2}$ and $\mathbf{MedTC}$ are equivalent, so there is a requirement reification that can be made:

$$\frac{\mathbf{Med}(a')^{a_1,c}_{c_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{MedTC}^{c}_{a'}}{\mathbf{Med}(a')^{a_1,c}_{c_1}, \mathbf{C1}^{c_1}_{\emptyset} \vdash \mathbf{MedTC2}^{c}_{a'}}$$

5.5 Problem solving

The topmost problem above has a particularly simple form; we are ready to solve it. Looking again at the general rule in Section 4, when $n = 0$ we have a special case, the *problem solved rule*:

$$\frac{}{W, S \vdash R}\ \mathcal{CA}$$

where $\mathcal{CA}$ is the argument that S is a solution of $W, S \vdash R$, meaning that the problem's solution has been found.

One can imagine many situations in which the problem solved rule will be applicable: the first is when a problem is sufficiently simple to be solved by inspection. Another is when it is complex, but sufficiently well-defined: perhaps a previous development of a similar problem has led to knowledge of a solution that is well circumscribed and easy to deal with, so that the solution to that problem can be reused as is, or — more likely — after some form of customisation. Customisation for reuse is not a routine task: it still, for instance, requires the development of a correctness argument but, again, this correctness argument might be adapted from one that already exists.

5.5.1 Example

In the gate example, after progression, we are left with a problem of the simple form:

$$\mathbf{Med}(a')^{a_1,c}_{c_1}, \mathbf{C}1^{c_1}_{\emptyset} \vdash \mathbf{MedTC}^{c}_{a'}$$

By inspection, we can write the solution $\mathbf{C}1$ as

$\mathbf{C}1$ At (time mod 180) = 0, start to open the gate (issue a $\langle \mathbf{C}1.\mathit{Off}, \mathbf{C}1.\mathit{Clockw}, \mathbf{C}1.\mathit{On}\rangle$ event sequence). At (time mod 180) = 10, start to close the gate (issue an $\langle \mathbf{C}1.\mathit{Off}, \mathbf{C}1.\mathit{AntiClW}, \mathbf{C}1.\mathit{On}\rangle$ event sequence).

Applying the problem solved rule, we have:

$$\frac{}{\mathbf{Med}(a')^{a_1,c}_{c_1}, \mathbf{C}1^{c_1}_{\emptyset} \vdash \mathbf{MedTC}^{c}_{a'}}\ \mathcal{CA}$$

where

$\mathcal{CA}$ $\mathbf{C}1$, at (time mod 180) = 0, issues the event sequence $\langle \mathbf{C}1.\mathit{Off}, \mathbf{C}1.\mathit{ClockW}, \mathbf{C}1.\mathit{On}\rangle$ (specification). The mediator, when in state $\mathit{commandIssued} = \mathit{true}$, will issue the same sequence (domain description). Ten minutes later, at (time mod 180) = 10, $\mathbf{C}1$ issues the event sequence $\langle \mathbf{C}1.\mathit{Off}, \mathbf{C}1.\mathit{Anti ClW}, \mathbf{C}1.\mathit{On}\rangle$ (specification). The mediator in state $\mathit{command Issued} = \mathit{true}$ will issue the same sequence (domain description). Therefore, the combination of $\mathbf{C}1$ and mediator behaviour will satisfy the requirement that, when $\mathit{command Issued}$ is false, the mediator should issue $\langle \mathit{Off}, \mathit{ClockW}, \mathit{On}\rangle$ at the beginning of a three-hour period, followed by $\langle \mathit{Off}, \mathit{AntiClW}, \mathit{On}\rangle$ ten minutes later.

Although we will not present the development of $\mathbf{C}2$, its form is similar. This completes the example, and the definition of the main rules in the software problem calculus. We now turn to consider how other approaches to software problem transformation[5] may be represented in our calculus.

6. SOFTWARE PROBLEMS AND OTHER APPROACHES

There are a number of validated approaches to requirements engineering in the literature. It is a goal of the authors to map those parts of other approaches that most closely represent problem transformations into our framework. Here we provide some initial interpretations both from other work of the authors — Architectural Frames [30, 16] — and from Goal-oriented Requirements Engineering [37, 27], and Viewpoints [34].

6.1 Architectural Frames

Architectural Frames (or *AFrames* for short) were introduced in [30, 16] as a new element of the Problem Frames framework. The intention of AFrames was to provide a practical tool for sub-problem decomposition and recomposition that allows the Problem Frames practitioner to separate and address, in a systematic fashion, the concerns arising from the intertwining of problems and solutions. The rationale behind AFrames is the recognition that solution structures can be usefully employed to inform problem analysis. For this document we interpret AFrames as rule applications.

AFrame application amounts to a combination of architectural expansion and progression, which provides guidance to the developer as to how to decompose a problem given an architecture fitting a particular context (with its topology) and requirements. As such, AFrames capture development expertise: it is a particular combination of architectural expansion, including choice of solution architecture, and progression *that has already proven useful* for a particular class of problems. AFrames work with problem classes, characterised in [30, 16] by problem frames [21].

The example development of Section 5.3 introduced a mediator because there were requirement parts that needed to be prioritised to avoid conflict. A similar architecture was proposed in [23] as a generic solution architecture for conflicting requirements, in the class of Controlled and Required Behaviour (multi-)Frames [21]. To capture such an architecture as an AFrame in our calculus, we express the accumulated rule application consisting of architectural expansion and progression in the following derived single-step problem transformation:

$$\frac{\begin{array}{c}\mathbf{Med}(a')^{c}_{c_1}, \mathbf{C}1^{c_1}_{\emptyset} \vdash \mathbf{R1}^{c}_{a'} \\ \mathbf{Med}(a')^{a_1,c}_{c_2}, \mathbf{C}2^{c_2}_{a_1} \vdash \mathbf{R2}^{c}_{a',a_1}\end{array}}{\mathcal{CD}(d)^{b}_{c}, \mathcal{O}^{a}_{\emptyset}, \mathbf{Med}[\mathbf{Med}(a')^{a_1,c}_{a,b,c_1,c_2}](\mathbf{C}1^{c_1}_{\emptyset}, \mathbf{C}2^{c_2}_{a_1})^{c}_{a,b}) \vdash \mathbf{R}^{d}_{a}}\ \mathcal{CA}$$

where $\mathcal{CA}$ is $R \iff R1$ *unless* $R2$, $\mathcal{CD}$ is a domain to be controlled, and $\mathcal{O}$ an operator that controls the domain through the machine. The $\mathbf{Med}$ AFrame is a generalisation of the example solution architecture in that the requirements no longer need to be of the specific form found there.

6.2 Goal decomposition

Goal-based requirements engineering can be used to derive requirements for a computer-based system from the goals of that system within its wider world context, for example, within an organisation. A goal is an optative description [25, 37], and so can be considered as forming the requirement of a problem. The notion of derivation used in goal-oriented approaches maps into our framework if we reify a goal-as-requirement as goal-as-requirements.

In the simplest form of goal decomposition, high level goals are decomposed into sub-goals with decomposition being complete when the goals are those of the information system (IS). An IS goal is then decomposed into functional and non-functional requirements for the computer-based system, perhaps expressed through real-world domains.

There are two main goal decomposition forms, *conjunctive* and *disjunctive* decomposition. For a conjunctive decomposition each sub-goal must be attained to attain the original goal. In the disjunctive case, at least one of the sub-goals must be attained for the original goal to be attained.

In our calculus, that a goal G can be decomposed as $G1 \wedge ... \wedge Gn$, i.e., as independent, non-conflicting sub-goals, will mean that there are n sub-problems, one corresponding to each conjunct, each of which share the context of the original goal, but each having its own solution — that there are no conflicts introduced by the conjunctive sub-goals means that the recomposition of the solution to the sub-problems is simple parallel composition (meaning that conjunctive decomposition is a special case of the projection rule):

$$\frac{W, S1 \vdash G1 \qquad ... \qquad W, Sn \vdash Gn}{W, S1 \| ... \| Sn \vdash G1 \wedge ... \wedge Gn}$$

[5]Although, of course, software problem transformation is only a retrospectively applied term.

In contrast to conjunctive decomposition, disjunctive decomposition of a goal G decomposed as $G1 \vee ... \vee Gn$ generates n possible trees

$$\frac{W, Si \vdash Gi}{W, Si \vdash G1 \vee ... \vee Gn}$$

the solution to any providing a solution to the original.

In real life, complications arise in goal decomposition: one such is when two (or more) conjunctive sub-goals are in conflict, meaning that they cannot be satisfied (either partially or completely) together [38]. We have already seen an example of conflict in the definition of the sluice gate in which the goal of automatic control was in conflict with that for manual control. In that situation the conflict was solved in the solution space. In other situations, particularly for high-level goals, the conflict will need to be resolved by, for instance, choosing a consistent subset of sub-goals.

In each of these situations, sub-goal conflict manifests itself in our problem calculus, typically being apparent in the difficulty of developing the $\mathcal{CA}$ for an upwards step in a problem development (that sub-goals cannot be satisfied together indicates the impossibility of developing the correctness argument). Of course, even though goals may be in conflict, if desired, such conflict need not delay a problem development; it is only the choice of $\mathcal{CA}$ that might be delayed while the sub-problems are explored more fully.

6.3 Viewpoints

Another possible complication is that a goal or domain description can mean different things to different stake-holders. In the case that stake-holders disagree, they may be encouraged to discuss and negotiate away their disagreements, as is suggested in the View-Points process [34]. Our modelling of goal or domain conflict is to accept that two (or more) stake-holders' conflicting views on a problem mean, essentially, that there are two (or more) problems to be solved, under the constraint that a single solution *must satisfy both*. Assume that stake-holders $T1$ and $T2$ see goal G (respectively, problem context W) in different ways (whether they disagree on all of it or just some part of it is immaterial at this point), as $G1$ and $G2$, say (respectively, $W1$ and $W2$). This leads us to the rule:

$$\frac{W1, S \vdash G1 \qquad W2, S \vdash G2}{W, S \vdash G} \quad \begin{array}{l} Wi = W \textit{ interpreted by } Ti \\ Gi = G \textit{ interpreted by } Ti \end{array}$$

7. DISCUSSION AND CONCLUSIONS

We have presented a software problem calculus that allows software problems to be described, and software specifications and correctness arguments to be incrementally built. It is not claimed that the calculus has a practical application in its current form; rather some basic rule forms have been determined and it has been shown how the solution to a simple problem can be found by application of the problem transformation rules. We have also made interpretations of other forms of 'software problem transformation', including some simple ones from Goal-oriented requirements engineering and Viewpoints, in the calculus. The interpretation was relatively simple; for instance, the basic conjunctive goal decomposition for instance, being a case of problem projection. This gives us hope that other approaches will map into our calculus, meaning that the more or less formal relationships between RE approaches can be explored.

Although the system for transforming problems is formal, it does not require that problems are described formally (although more information may be extracted from a more formal problem description), nor that the correctness argument is formal. Rather the user of the rules can work with the description they have, refining them as necessary as features of a problem are recognised.

7.1 Design processes; Normal and radical design

Vincenti, [39], describes the design of aeroplanes — 'typical of devices that constitute complex systems' — as being multilevel and hierarchical. His process starts with project definition — including the *translation of some usually ill-defined [...] requirement into a concrete technical problem* — proceeding through *overall design*, *major component design*, *subdivision of areas of component design according to engineering discipline required*, and *further subdivision into highly specific problems*. The process produces, at each lower level, smaller more manageable sub-problems, each of which can be 'attacked in semi-isolation'. Whereas we do not claim to have a process, the details of Vincenti's partitioning of the design process into levels has some resonance within our calculus. Indeed, the intention and the actual steps are the same: from an ill-defined problem (recall the initial requirement for the sluice gate) we transformed to a soluble technical problem; then, via an architecture, we identified specific components that were designable in (semi-)isolation. Of course, in such a simple example, many of the lower problem levels would be coalesced. However, there appears no *a priori* reason why Vincenti's design hierarchy should not usefully manifest itself for larger problems and, indeed, why this wouldn't be representable in our calculus.

This brings us to normal vs. radical design, one of Vincenti's most powerful ideas. Radical design engages a designer in the design of a device s/he has never seen before and for which there is no presumption of success. Normal design, that Vincenti claims as 'the bulk of day-to-day engineering', is design from 'combinations of off-the-shelf technologies that are then tested, adjusted, and refined until they work satisfactorily'. Normal design may still entail novelty, but is not characterised by it — there is no need for technical innovation, for instance. Some have argued that, in software engineering, much design is still radical. There are normal design tendencies in software engineering, currently mainly in the solution domain (architectures [32], components [19, 11], design pattern [13], *etc.*) but also in the problem domain (analysis patterns [12], problem frames [20, 21], the domain theory [36, 35]). We expect that our calculus can offer something to both radical and normal design camps. However, we intend that, with its interpretations of extant approaches, its foundational nature, its acceptance of reuse — both experience-based (AFrames, for instance) and artefact-based (architectures and components) — that it offers something of unique value to normal software design. There is still much to be done, before this claim can be substantiated.

7.2 Gentzen's calculus

The notion of problem transformation is central to our software problem calculus; rules in the calculus relate an original problem (written, in the Gentzen style [22, 28], below the line) to (a set of) transformed problems (written above the line).

Gentzen's sequent calculus provides a firm basis for our software problem calculus. There are extensions we have added: the notion of a separately developed correctness argument is missing from Gentzen's calculus (for which the proof is, literally, the correctness argument itself). Our needs are different: although problem manipulation is formally driven in the Gentzen sense — i.e., adhering to the form of syntactic manipulation — a correctness argument is typically not a proof of correctness: it may not even be rigourous; it is most usefully that that can be shared between developer and customer (or other stake-holders). It may therefore be testing based, for instance, or even have components taken from many different argumentation languages as would be the case written to satisfy a safety authority. It may even be based on trust of an individual developer's expertise.

7.3 Future work

Use cases [6] focus on interactions at the boundary of the software system, allowing system operations to be specified from identified goals of actors external to the system. As such, we wish them to have a place in our calculus — we have some evidence that use-cases and their embedded scenarios, can be modelled as forms of problem progression. The modelling of a problem domain and the related structuring of the solution, such as occurs in OOA&D ([8]), is also an important technique. Again, we have early evidence that this can be incorporated within our calculus. We will, at some point, be required to show that solution description languages, such as architectural description languages (ADLs.), can be mapped into our notation for architectures. Initially, we intend to look to PADL [2] as it appears to form a good fit, given a domain description language as CCS [26].

This calculus forms part of our larger work on a general problem calculus, one in which general engineering problems can be represented together with the various domains in which they can be solved. For that work a much richer concept of solution is needed to include, at least, a social part — humans, their skills, competences and training needs — so that socio-technical systems can be reasoned about.

We are also building a tool for problem representation and transformation based on the ideas of this paper.

Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Grants, and the EPSRC, Grant number EP/C007719/1. Thanks also go to our colleagues in the Computing Research Centre in The Open University, particularly Anne De Roeck, Bashar Nuseibeh and Michael Jackson.

8. REFERENCES

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

[2] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.

[3] D. Bjorner, S. Koussobe, R. Noussi, and G. Satchok. Michael Jackson's Problem Frames: Towards methodological principles of selecting and applying formal software development techniques and tools. In L. ShaoQi and M. Hinchley, editors, *Proceedings of the International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE Computer Society Press, 1997.

[4] S. Bleistein, K. Cox, and J. Verner. Requirements Engineering for e-Business Systems: Intergrating Jackson Context Diagrams with Goal Modelling and BPM. In *Proceedings of the 11th International Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 410–417, Busan, Korea, 2004. IEEE. 30th November–3rd December 2004.

[5] C. Choppy and G. Reggio. A UML-Based method for the Commanded Behaviour frame. pages 27–34, Edinburgh, 2004. IEE. 24 May 2004.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[7] CoFI. CASL the common algebraic specification language summary, version 1. Technical report, The Common Framework Initiative for Algebraic Specification and Development, 1999. http://www.brics.dk/Projects/CoFI/.

[8] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice-Hall., 1994.

[9] K. Cox, J. G. Hall, and L. Rapanotti. Editorial: A roadmap of problem frames research. *Information Science and Technology*, 2005.

[10] G. Delannay. A meta-model of Jackson's Problem Frames. Technical report, University of Namur, 2002.

[11] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

[12] M. Fowler. *Analysis Patterns: Reusable Object Models*. Object Technology Series. Addison-Wesley, 1997.

[13] E. Gamma, R. Johnson, J. Vlissides, and R. Helm. *DesignPatterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[15] J. G. Hall and L. Rapanotti. A reference model for requirements *engineering*. In *11th IEEE International Conference on Requirements Engineering (RE 2003)*, pages 181–187. IEEE CS Press, 2003.

[16] J. G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Journal of Software and Systems Modeling*, 2005. Electronic version available at Springer Online First (29 July 2004). Printed version in press.

[17] J. Hammond, R. Rawlings, and A. Hall. Will it work? In *The proceedings of the 5th IEEE International Symposium on Requirements Engineering.*, 2001.

[18] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Language and System*, 12(3):463–492, July 1992.

[19] J. D. J. Cheesman. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[20] M. Jackson. *Software Requirements and Specifications*. Addison-Wesley, Harlow, 1995.

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

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

[23] R. Laney, L. Barroca, M. A. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proceedings of the 12th Int. Conf. on Requirements Engineering (RE'04)*, pages 113–122, Kyoto, Japan, 2004. IEEE Computer Society Press.

[24] Z. Li, J. G. Hall, and L. Rapanotti. Reasoning about decomposing and recomposing problem frame developments: a case study. In *Proceedings of 1st International Workshop on Applications and Advances of Problem Frames*. IEEE CS Press, 2004.

[25] P. Loucopoulos and V. Karakostas. *System Requirements Engineering*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[26] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[27] J. Mylopoulos, M. Kolp, and J. Castro. Uml for agent-oriented software development: The tropos proposal,. In *Proceedings of the Fourth International Conference on the Unified Modeling Language UML 01 - Toronto, Canada,,* 2001.

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

[29] G. Polya. *How to solve it*. Princeton University Press, second edition, 1957.

[30] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *12th IEEE International Conference on Requirements Engineering (RE 2004)*, pages 80–89. IEEE Computer Society, 2004.

[31] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison Wesley, Harlow, England., 1999.

[32] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an Emerging Discipline*. Prentice Hall, 1996.

[33] S. Singh. *Fermat's Last Theorem*. Fourth Estate, 1998.

[34] G. Spanoudakis, A. Finkelstein, and W. Emmerich. Viewpoints 96: international workshop on multiple perspectives in software development (sigsoft 96) workshop report. *SIGSOFT Softw. Eng. Notes*, 22(1):39–41, 1997.

[35] A. Sutcliffe. *The Domain Theory: Patterns for Knowledge and Software Reuse*. Lawrence Erlbaum Associates, 2002.

[36] A. Sutcliffe and N. Maiden. The domain theory for requirements engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, 1998.

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

[38] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. on Sofware. Engineering, Special Issue on Inconsistency Management in Software Development*, 1998.

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

[40] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of RE97: 3rd International Symposium on Requirements Engineering*, pages 226–235, 1997.

[41] E. S. Yu. Modeling organizations for information systems requirements engineering. In *Proceedings 1st IEEE International Symposium on Requirements Engineering*, pages 34–41, 1993.

[42] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.