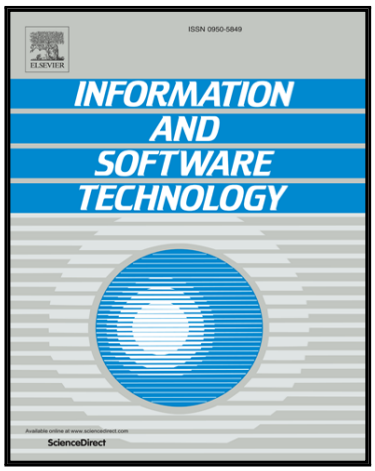


A design theory for software engineering

Jon G. Hall, Lucia Rapanotti

PII: S0950-5849(17)30069-1
DOI: 10.1016/j.infsof.2017.01.010
Reference: INFSOF 5801

To appear in: *Information and Software Technology*

Received date: 10 June 2016
Revised date: 17 January 2017
Accepted date: 24 January 2017

Please cite this article as: Jon G. Hall, Lucia Rapanotti, A design theory for software engineering, *Information and Software Technology* (2017), doi: 10.1016/j.infsof.2017.01.010

A design theory for software engineering

Jon G. Hall[a,] , Lucia Rapanotti[a]

[a]School of Computing and Communications, The Open University, Milton Keynes, MK7 6AA, UK

Abstract

Context: Software Engineering is a discipline that has been shaped by over 50 years of practice. Many have argued that its theoretical basis has been slow to develop and that, in fact, a substantial theory of Software Engineering is still lacking.
Objective: We propose a design theory for Software Engineering as a contribution to the debate. Having done this, we extend it to a design theory for socio-technical systems.
Method: We elaborate our theory based on Gregor's influential 'meta-theoretical' exploration of the structural nature of a theory in the discipline of Information Systems, with particular attention to ontological and epistemological arguments.
Results: We argue how, from an ontological perspective, our theory embodies a view of Software Engineering as the practice of framing, representing and transforming Software Engineering problems. As such, theory statements concern the characterisation of individual problems and how problems relate and transform to other problems as part of an iterative, potentially backtracking, problem solving processes, accounting for the way Software Engineering transforms the physical world to meet a recognised need, and for the problem structuring process in context. From an epistemological perspective, we argue how the theory has developed through research cycles including both theory-then-(empirical-)research and (empirical-)research-then-theory strategies spanning over a decade; both theoretical statements and related empirical evidence are included.
Conclusion: The resulting theory provides descriptions and explanations for many phenomena observed in Software Engineering and in the combination of software with other technologies, and embodies analytic, explanatory and predictive properties. There are however acknowledged limitations and current research to overcome them is outlined.

Keywords: Software Engineering, Design theory, General engineering, Problem orientation, Problem solving

*Corresponding author
Email addresses: Jon.Hall@open.ac.uk (Jon G. Hall), Lucia.Rapanotti@open.ac.uk (Lucia Rapanotti)

1. Introduction

Software Engineering (SE) is a discipline that has been shaped by over 50 years of practice. Driven primarily by the needs of industry, a theoretical basis has been slow to develop. Recently, Johnson et al. [1] have argued the need for theories which can address 'significant' questions within SE. Their main criticisms are that existing theories tend to be small, addressing limited sets of phenomena, very often implicit and only casually introduced by authors, with little academic discussion or rigorous evaluation within the community. Undoubtably, their arguments have stirred some debate in the wider SE community, and perhaps have been the catalyst for a renewed interest in the theoretical foundation of the discipline.

Our contribution is a design theory for SE which locates software as a solution within problem solving. The theory has developed from our work on Problem Oriented Engineering (shortly POE, [2]), a practical engineering framework with an accumulated body of work spanning over a decade, evaluated and validated through a number of real-world engineering case studies. In relation to previous publications, one novel contribution is to make explicit the theory implicit in the definition of our POE framework. In fact, we contribute two design theories. The first concerns problems for which software is exclusively the solution; this is obviously limited given that other engineered artefacts, for instance, end-user documentation or training materials, will usually be needed as part of a solution. The second theory remedies this limitation.

Our presentation is informed by Gregor's [3] 'meta-theoretical' exploration of Information Systems (IS), particularly her ontological, epistemological and domain questions.

The paper is organised as follows. Section 2 recalls the debate on the need for SE theories. Section 3 discusses the theoretical provenance of our theory, followed by its detailed presentation in Sections 4 and 5. An evaluation and discussion of ongoing research is given in Section 6, while Section 7 discusses related work. Section 8 concludes the paper.

2. Background

2.1. The recent debate for on theory in SE

Johnson et al. [1] constrain neither the form that a significant SE theory should have nor how it should be generated. In contrast, Adolph and Kruchten [4] argue that an SE theory 'must be useful to practitioners and explain the phenomena they are experiencing,' whence proposing an empirical approach. Similarly, Ralph [5] argues for the usefulness of *process theories* in SE, providing both a taxonomy of process theory types and examples of where such theories could be beneficial.

A complementary stance is taken by Staples [6], who proposes that, alongside process theories, what SE needs are *product theories* if engineering concerns, such as performance, are to be taken into account: the primary goal of engineering is, after all, to change the world. Accordingly, such product theories may not be

exact, but only provide some conservative approximations to support assurance about the use of artefacts, i.e., they need only be general and precise enough to reason about whether an artefact meets acceptable requirements for use. A different perspective yet is taken by Smolander and Paivarinta [7], who argue for the need of *theories of practice* focused on design and development practices, using a reflection-in-action approach to theory generation.

Wieringa [8] argues that a desire for universal theory leads to theoretical idealisations unusable in practice. Instead, he suggests that 'middle-range' theories are more usable by practitioners. The case for middle-range theories is also made by Stol and Fitzgerald [9, 10], who describe them as stepping stones towards a more general and inclusive theory.

More generally, the community is trying to come to terms with what is meant by 'theory.' There is some agreement on what is not considered a theory (e.g., [11]) and much current thinking aligns closely with [3], which constitutes the frame of reference for this paper.

Finally, Ralph [12], Johnson and Ekstedt [13] and others have considered how existing theories, such as complexity and cognition theories, might contribute to a general SE theory.

It appears that current work is speculative: we are still a long way away from an agreement on what the characteristics of a SE theory should be.

2.2. Gregor's meta-theoretical model

Gregor [3] explores IS theory. Given the strong relation between SE and IS, much of her analysis extends naturally to SE and has been adopted in that community.

Gregor's exploration asks four distinct categories of question: core *domain* questions — the phenomena of interest and the boundary of the discipline; *structural* or *ontological* questions — the nature of theory and the form that knowledge contributions make; *epistemological* questions — how the theory is constructed, the research method used and the way knowledge is accumulated; and *socio-political* questions — concerning the socio-political context in which a theory was developed.

Gregor's argues that an IS theory should be able to link natural, social and artificial worlds, drawing upon their respective design sciences, and that a wide view of theory should be taken. This also makes sense for SE and reflects the current SE community view. Gregor also sees no requirement to commit to one specific epistemological view, also mirrored in the current SE debate.

Gregor [3] concludes that some combination of the following is essential:

- description and analysis: to describe phenomena of interest and related constructs, to generalise them within an identified scope, and to analyse their relationships;
- explanation: to comment on 'how, why, and when things happened, relying on varying views of causality and methods for argumentation';

- prediction: to predict what will happen in the future under specified conditions; and
- prescription: to allow the definition of methods or 'recipes for doing' which, if followed, will make theory predictions true under specified conditions.

The last quality is notable in relating theory and method: although methods can follow theories, they are not the same thing [5]; each can exist independently of one another.

3. Research cycle and theory conceptualisation

Our design theory is based on a body of work accumulated in over a decade of research into a design-theoretic approach to problem solving in the context of software and systems engineering. The research cycle adopted has included both theory-then-(empirical-)research and (empirical-)research-then-theory strategies, starting from an initial theoretical proposition [14]. We present our theory in detail alongside a discussion of the empirical evidence accumulated in its support, and of aspects for which both theoretical and empirical work is still required.

3.1. Theoretical provenance

G.F.C Rogers defined engineering as [15]:

> the practice of organising the design and construction of any artifice[1] which transforms the physical world around us to meet some recognised need.

Rogers' definition places engineering in the problem solving domain, the problem being, given a real world environment and need, to design and construct an artifice with the requisite properties.

This definition specialises easily to *Software* Engineering simply by taking the artifice to be software, giving

> software engineering is the practice of organising the design and construction of software which transforms the physical world around us to meet a recognised need.

Unfortunately, this 'software-as-artifice' perspective is deficient:

- that software is the artifice implies that the enviroment has a particular form, so that software causes become physical e□ects, and *vice-versa*;

[1] Rogers appears to use *artifice* rather than, for instance, *artefact* to emphasise that a solution may have no physical embodiment, as is the case with software.

- we cannot, *ab initio*, know the precise combination of solution technologies that will ultimately satisfy the need, even less that software will be the sole solution technology.

Whereas the first of these can easily be discharged by careful choice of environment, the second is more difficult and motivates our second theory: it is likely that through solution technology combinations, both formal software and non-formal solution domains will need to be combined. But then, as Turski [16] observed:

> 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.

so that design involving software must cross and recross the bridge between the formal and the non-formal, what we call *the Turski disconnect*.

Moreover, in creating software to meet human needs, software engineering will encounter *ill-structured* [17] and *wicked* [18] problems, in which understanding of the problem and its stakeholders, including the designer, plays a fundamental role in the problem solving process. The stakeholders are the actual determinants of satisfaction, not any formally determined criteria that might exist.

Hence, there are three key needs that a theory for SE must meet:

- it must account for the way software is used to meet recognised needs in the real-world;
- it must bridge the formal and non-formal divide; and
- it must be sufficiently flexible to account for stakeholders' notions of satisfaction.

We therefore propose a theory that:

- takes a balanced view of problem and solution, allowing us to explore the problem without jumping to conclusions as to what the solution technology *should* be, and *vice-versa*, so as to explore ways in which the solution affects our understanding of the problem;
- makes no constraints on the language in which problems or solutions are expressed, supporting formal, informal and even intuitive reasoning [18, p. 395];
- gives focus to the construction of a stakeholder-relevant design rationale which captures their notion of satisfaction.

3.2. Phenomena basis

To support Rogers in being able to design and construct an artifice that, in its real world environment, meets a need means relating artifice, environment and need. Underpinning each of these in our theory is the notion of phenomenon:

- a *need* expresses wished-for phenomena and relations between them;
- an *environment* describes existing phenomena and their relations;
- an *artifice* is the mechanism created so that existing phenomena and their relations are organised to produce those that are wished for.

The notion of phenomenon is therefore central to our design theory.

Definition 1 (Phenomenon). A *phenomenon* is an element of the world, the occurrences of which are observable.

Some phenomena occur naturally others artificially: a protein occurs naturally through protein synthesis whereas, from a software perspective, the mechanisms for software handshake occurrences are constructed by software engineers. The meanings of natural and artificial here relate not to the phenomena themselves but to the mechanisms that underlie their occurrence: it may be that a solution to a problem includes an artificially produced protein. In essence, our theory covers how design links natural and artificial, combining, creating and relating mechanisms to meet a phenomena-expressed need in a phenomena-expressed world.

3.2.1. Phenomena models

To solve a problem, we should not *a priori* assume any particular solution technology or combination. We therefore need to be able to link across disciplines and rely on engineers in their specialist disciplines to know what they consider valid phenomena; only from these will valid phenomena models arise.

Having said this, however, that phenomena are observable reveals their dependence on time. That dependence need not be functional, however: whereas $Temperature$, for instance, is point-wise observable, a software handshake extends over an interval. Thus, other topologies of time, including the time intervals underpinning the Duration Calculus [19], may be needed to model phenomena. Our wish for inclusivity determines that we do not further constrain the semantic basis of the notion of a phenomenon.

3.3. Design theoretic

We propose a new *design theoretic* approach to characterise the elements of problem solving in terms of their effect on the design process, and only thereby the design artefact. Our inspiration is Gentzen's proof theoretic approach [20, 21]. Of course, design is much freer than proof and there are substantial differences from Gentzen's theory: for instance, our notion of design step justification (Definition 8) contrasts to Genzten's simpler proof steps; such differences are explicated as they are encountered throughout the paper.

4. Software engineering as design theoretic problem solving

Our SE theory is formal and based on the notion of, initially, software problem. The presentation below is example driven.

4.1. Software problems

Suppose a *problem owner* recognises a need in the real world and wishes that need to be satisfied by the design and construction of a software artefact: they have identified a *software problem* that they wish solved.

Definition 2. Given a problem owner PO, a *software problem* P is a pair, consisting of a real world environment E_{PO} and a need N_{PO}:

$$P = (E_{PO}, N_{PO})$$

Note that we make no assumption that PO's view as expressed in the environment is realistic or representable nor that their need is satisfiable. Thus E_{PO} and N_{PO} may have no representational instantiation; equivalently, PO's initial conceptualisation of their problem may have no solution, or even sense. Even in formal fields such as mathematics, *unrealistic problems* such as these arise often at the beginning of problem solving processes [22].

Example 1. Jon, an $iPad$ user, wants to stay dry by being alerted when the weather forecast predicts rain in his area:

'I need to know when I should take an umbrella'

Jon's problem is then:

$$P_{Jon}: \quad (iPad, \text{'I need to know when I should take an umbrella'})$$

4.2. Problem Solving

Irrespective of sense or solution existence, we will assume that a problem owner's software problem becomes a challenge to a software engineer to make sense of it and to solve it[2].

Following Rogers, given a software problem, the software engineer's challenge consists of the following steps:

SE1 gaining an understanding of the real-world environment in which the problem is located, and of the problem owner's identified need, i.e., the software engineer must form their own view, (E, N), of the problem (E_{PO}, N_{PO});

SE2 agreeing with the problem owner that (E, N) is representative, a form of validation;

[2] In the simplest case, the problem owner and software engineer will be the same person; in general they will be different.

SE3 producing the software S;

SE4 convincing the problem owner that S meets the agreed recognised need N in the agreed real-world environment E to their satisfaction, another form of validation.

Previously, we have denoted (E, N) as a set of solutions [23]. Under this semantics, we might characterise the software engineer's task as identifying elements $S \quad (E, N)$ and choosing one of them.

Here, however, we recognise that the solution process above is likely to be highly non-linear (c.f. [18, p. 392]) and so remove the asymmetry between problem and solution that could be implied by the notation $S \quad (E, N)$. Thus the software engineer's problem is written

$$E(S) \; \square_{PO} \; N$$

by which we mean 'Find S which, when installed in E, meets N to the satisfaction of PO.'

Example 2. $Lucia$, our software engineer, works with Jon to understand his need and real-world environment. She captures Jon's 'stay dry' need as $SDNeed$. $Lucia$ acknowledges that Jon has an $iPad$, and also discusses with him that they will need access to a source of a local weather forecasts, which they agree to refer to as $WStat$; this expands Jon's environment to $[WStat, iPad]$. Naming her solution $IRTA$, $Lucia$ forms her problem[3]:

$$[WStat, iPad](IRTA) \; \square_{Jon} \; SDNeed$$

4.3. The structure of a software problem

When solving, then, a software problem has four components: an environment E, a need N, a solution S and a problem owner PO. Both E and S depend on the notion of a *domain*:

Definition 3 (Domain). A *domain* D is a set of *controlled phenomena* together with a description of the domain's indicative, or known, properties, i.e., how its controlled phenomena interact with each other and those that it observes in its environment (its *observed phenomena*).

Thus, a domain names and describes a *mechanism* that relates phenomena [24, 25]. As is the case when problem solving begins, nothing may be known of a domain so that we can make no requirement of its description that it make sense; as problem solving proceeds through steps SE1–SE4, descriptions will evolve[4] before they can be validated.

[3] The square brackets delimit composite domain: see Definition 4.
[4] The mechanism for which is introduced in Section 4.5.

Example 3. Suppose $WStat$ is the domain consisting of controlled phenomena *{rl}* (the Boolean 'rain likely' phenomenon) and observed phenomena *{bar_p}*, with description relating them:

> The rain likely phenomenon (rl) holds only when the barometric pressure (bar_p) is less than 990 millibars.

When known, controlled phenomena are written superscripted, observed subscripted, thus:

$$WStat^{\{rl\}}_{\{bar_p\}}$$

We will often omit controlled and/or observed phenomena when clear by context. In an abuse of notation, we will often omit set brackets too.

4.3.1. Composite Domains

Phenomena controlled by one domain and observed by another are said to be *shared*. Domains can be brought 'within observation distance of each other' to form composite domains; domain composition places domains in parallel with synchronisation via the occurrences of shared phenomena:

Definition 4 (Composite domain). Given domains D^a_c and F^b_d we form the *composite domain*

$$[D, F]^{a\ \ b}_{c\ \ d \backslash a\ \ b}$$

the description of which is the conjunction of those of D and F.

Domain composition is associative and the composite is a domain that behaves as its composites albeit with shared phenomena occurrences synchronised ('within observation of each other'). This form of composition is similar to the ubiquitous synchronous (i.e., shared clock) parallel compositions. Examples include CSP [26], CCS [27], the Petri Box Calculus [28], and many others [29].

An *Environment* is a domain that forms the environment in which a candidate solution to a problem will be assessed as a solution:

Definition 5 (Environment). For domains D_1, ..., D_n, $n \geq 0$, define the *environment* E *on* $D_1, ..., D_n$, to be the composite domain $[D_1, ..., D_n]$.

When $n = 0$, $E = []$ is the empty composite domain; when $n = 1$, we write $E = [D_1] = D_1$.

4.3.2. Solutions

Software solutions are also domains:

Definition 6 (Software solution). A *software solution* is a domain, the description for which is a computation.

A software solution may be expressed in any programming language; its phenomena are those used by programming languages and their execution context, which might include memory, sensor data, actuators, etc.

4.3.3. Needs

Needs are similar to domains; however, a need's description expresses a wish.

Definition 7 (Need). A *need* N consists of a set of phenomena together with an optative, i.e., wished for, description of the relationship between them.

We partition a need's phenomena into two sets: those *constrained* by the need's description and those *referenced* by it. A problem's *need* states how a proposed software solution will be assessed as a solution to that problem: when installed in the problem's environment the solution should induce the desired relationship between referenced and constrained phenomena. Constrained phenomena are written superscripted; referenced subscripted.

Example 4. To interpret Jon's need, $Lucia$ creates $SDNeed^{alert}_{rl}$ with description:

> An $alert$ shall be raised whenever rl holds

4.4. Software problem transformation

All that remains to be understood of a software problem is the notion of *satisfaction*, $\square_{PO}$. This brings us to problem transformation.

Returning to our running example, first, we note that there is a difference between a candidate solution created by $Lucia$ and a problem solved to Jon's satisfaction: the difference being that Jon must agree that the candidate solution is good enough. To effect this, $Lucia$ must consider in which ways Jon would be convinced that his problem is solved by the candidate solution.

Example 5. IFTTT [30, `ifttt.com`] is a Software as a Service (SaaS, [31]) tool for creating simple trigger-action programs, or *recipes* [32].

As is a registered user of IFTTT, $Lucia$ has access to the If_Rain_Then_Alert recipe (see Figure 1). This recipe runs on IFTTT's servers, polling a local-to-$Lucia$ weather forecast, and sending an alert to $Lucia$'s $iPad$ whenever rain is forecast.

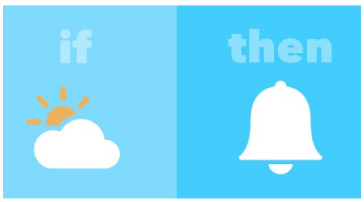
$Lucia$ believes that, if she was in Jon's place, the $IRTA$: If_Rain_Then_Alert recipe would satisfy her, i.e., she believes that she has solved the problem:

$$[WStat, iPad](IRTA) \ \square_{Lucia} \ SDNeed$$

The problem of convincing Jon remains: $Lucia$ constructs a demo (from her IFTTT account) to demonstrate how the candidate $IRTA$ solution would work for Jon. If this convinces Jon, then she has solved

$$[WStat, iPad](IRTA) \ \square_{Jon} \ SDNeed$$

In this case, the judgement as to whether one problem is solved is contingent on that of another and the existence of a justification — here, a convincing demo — that reflects a design step. This is captured in *software problem transformations*:



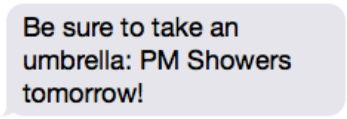


Figure 1: Left: the IFTTT If_Rain_Then_Alert recipe reminds the user to take an umbrella if tomorrow's forecast is for rain; and, right, a received alert.

Definition 8 (Problem transformation). Given problems

$$E(S) \ \square_{PO} \ N, \qquad E_i(S_i) \ \square_{PO_i} \ N_i \text{ for } i = 1, ..., n$$

and *step rationale* J, then we write:

$$\frac{E_1(S_1) \ \square_{PO_1} \ N_1 \quad ... \quad E_n(S_n) \ \square_{PO_n} \ N_n}{E(S) \ \square_{PO} \ N} \ \text{J} \tag{1}$$

to mean that:

> $E(S) \ \square_{PO} \ N$ is solved with respect to PO and with *design rationale* $AA_1 \wedge ... \wedge AA_n \wedge J$ whenever the $E_i(S_i) \ \square_{PO_i} \ N_i$ are each solved with respect to PO_i with design rationale AA_i and J satisfies PO.

Below the line is the *conclusion problem*; above are the *premise problems*. A step rationale that satisfies the conclusion problem owner is said to be *satisfactory*.

Note:

- with no premise problems remaining to be solved, i.e., $n = 0$, the conclusion problem is solved contingent only on the existence of satisfactory J.
- J need not be formal or logical (or even rational!). Thus, although the step form is purely logical, the design rationales that are constructed through it need not be.
- discovering satisfactory step rationales may be di□cult and anyway consumes resources, thus such steps a□ect process risk. We return to this in Section 4.7.

4.5. Transformation types

Smolander and Paivarinta [7] suggest that SE proceeds through a set of identifiable development practices. We observe that there are classes of problem transformations, including sensemaking [33], that recur during software problem solving which have similar step forms. To this end, we define *software problem transformation schema* which describe a general way in which a conclusion problem is related to premise problem(s) through a *step rationale*. The intention is for transformation schema to provide theoretical idealisations of common SE practices, a claim we revisit in Section 6.

4.5.1. Substitutions

Given a substitution[5] PS = $[T_i/V_i]$ of environment, need, and/or solution, and satisfactory step rationale J_{PS}, we have:

$$\frac{P[PS]}{P}\ \text{[Substitution]}_{J_{PS};\ \text{solution: } S[PS],\ \text{environment: } E[PS],\ \text{need: } N[PS]}$$

4.5.2. Interpretation Rules

The interpretation rules specialise SUBSTITUTION to a problem's elements; they are a form of sense-making. For appropriate substitutions ES, NS and SS and satisfactory step rationales J_ES, J_NS, and J_SS, we have:

$$\frac{\mathsf{E}[ES](\mathsf{S}) \ \square_{PO}\ \mathsf{N}}{\mathsf{E}(\mathsf{S}) \ \square_{PO}\ \mathsf{N}}\ \text{[Environment Interpretation]}_{J_ES}$$

$$\frac{\mathsf{E}(\mathsf{S}) \ \square_{PO}\ \mathsf{N}[NS]}{\mathsf{E}(\mathsf{S}) \ \square_{PO}\ \mathsf{N}}\ \text{[Need Interpretation]}_{J_NS} \qquad \frac{\mathsf{E}(\mathsf{S}[SS]) \ \square_{PO}\ \mathsf{N}}{\mathsf{E}(\mathsf{S}) \ \square_{PO}\ \mathsf{N}}\ \text{[Solution Interpretation]}_{J_SS}$$

Environment and Need Interpretation work in the 'problem space' to record changed problem understanding, so-called *problem exploration*; Solution Interpretation works in the 'solution space' to record improved solution understanding, so-called *solution exploration* (see also Section 4.6).

4.5.3. Solution Expansion

We define a distinguished class of *architectural structures*, or *AStruct*s (c.f. [34]) that are specific to solution interpretations. As suggested by their name, architectural structures bring a notion of architecture to our theory: an AStruct, $ASName$, combines a number of known solution components, C_i, with solution components, S_j, yet to be found:

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

[5] For any expression W, the notation $W[T_i/V_i]$ means W with each instance of T_i simultaneously replaced by V_i.

SOLUTION EXPANSION expands an AStruct as follows:

$$\frac{\begin{array}{c}[E, C_1, ..., C_m, S_2, ..., S_n](S_1) \rhd_{PO} \mathsf{N} \\ \vdots \\ [E, C_1, ..., C_m, S_1, ..., S_{j-1}, S_{j+1}, ..., S_n](S_j) \rhd_{PO} \mathsf{N} \\ \vdots \\ [E, C_1, ..., C_m, S_1, ..., S_{n-1}](S_n) \rhd_{PO} \mathsf{N}\end{array}}{E(S : ASName[C_1, ..., C_m](S_1, ..., S_n)) \rhd_{PO} \mathsf{N}} \text{ [SOLUTION EXPANSION]}$$

Note: SOLUTION EXPANSION is purely syntactic and so generates no step rationale obligation. It creates n premise problems each expressed in terms of the conclusion's elements: problem j and k ($j = k$) are distinguished only by the fact that, whereas problem j has S_j as solution domain, problem k has S_j as part of its environment.

AStructs are a simple architectural description language for our theory. They might be as small as a single component, as large as a software architecture [35] or even, in the expanded theory described later, an Enterprise Architecture [36]. As our theory is description language agnostic, AStructs do not conform to standards for such languages, for instance, [37]. However, according to that standard, as well as detailed architectural views, an architecture description is expected to include:

- identification and overview information of the architecture being expressed;
- identification of the system stakeholders and their concerns;
- architecture rationale (explanation, justification, reasoning for decisions made about the architecture being described).

all of which are available in one form or another in the various elements of a problem.

Special cases. When $n = 0$ above there are no premise problems and so problem solving is complete. Of course, a prior satisfactory step rationale would have been needed for the SOLUTION INTERPRETATION that introduced the AStruct.

When $n > 1$ above, each pair of premise problems *tangle*:

Definition 9 (Tangled Problems). Problems P and Q *tangle* whenever their constituent domains or needs share phenomena.

Constraints, such as those imposed by a solution, percolate between tangled problems through their shared phenomena. Such constraints need not be consistent and so may prevent the solution of either problem in a tangle contributing as a solution of both. Thus solutions to tangled problems generally need to be designed together, or *co-designed*.

Example 6. The familiar Model-View-Controller architecture [38] can be captured in the environment of the $iPad$ (see Figure 2) as:

$$MVC_{iPad}[iPad\_GUI](Model, View, Controller)$$

The MVC_{iPad} AStruct includes the 'already known' graphical user interface $iPad\_GUI$[6], and to-be-found components $Model$, $View$ and $Controller$.

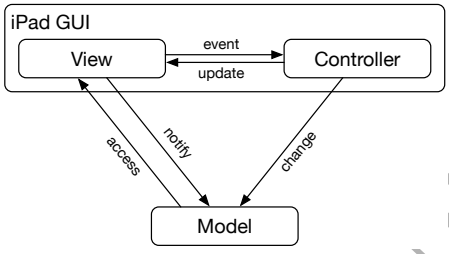


Figure 2: MVC_iPad architecture underpinning the iPad Graphical User Interface (GUI)

Should $Lucia$ have chosen to build a bespoke solution rather than rely on IFTTT, she could have used MVC_{iPad} to do so. This would have involved SOLUTION INTERPRETATION by the MVC_{iPad} AStruct above. SOLUTION EXPANSION gives these three problems:

$$P_M : [WStat, iPad, iPad\_GUI, Controller, View](Model) \ \square_{Jon}\ SDNeed$$

$$P_V : [WStat, iPad, iPad\_GUI, Model, Controller](View) \ \square_{Jon}\ SDNeed$$

$$P_C : [WStat, iPad, iPad\_GUI, Model, View](Controller) \ \square_{Jon}\ SDNeed$$

each pair of which tangle. Care must therefore be taken to ensure that solutions to these individual problems are co-designed so that they work together. Examining the model problem, P_M, it may appear that we need descriptions of the $Controller$ and $View$ to be able to solve it, whereas these will only be available when problems P_V and P_C are solved. But P_V and P_C each depend on the other as well as P_M.

The tangled interdependence mirrors what occurs in practice, and there are many practical ways of overcoming it. In the literature, for instance [39, 40], the guidance is to focus on the $Model$'s design first, which depends on the semantics of the domain of application, with a design goal of independence from its presentation (the $View$) and updating (the $Contoller$), together with a protocol for the phenomena that drive the $View$ and the $Controller$.

Thus, if M is the description of the designed model, we can write:

$$P_V : \quad [WStat, iPad, iPad\_GUI, Model^{access}_{notify} : M\ ](View^{notify}_{access}) \ \square_{Jon}\ Need$$

[6] Which we don't detail.

P_C : $[WStat, iPad, iPad\_GUI, Model_{change} : M\,](Controller^{change})$ ▫$_{Jon}$ $Need$

from which we can address the separate designs of the $View$ and the $Controller$.

Naivety risk. When problems tangle they cannot be decomposed in the naive hope that their solutions will recombine to form a solution to the tangle; indeed, the resource expended in such naive decomposition is likely to be lost. Thus we recognise the notion of *naive problem decomposition risk* (more simply, *naivety risk*) which should be considered and managed alongside other developmental risks, it being managed by careful co-design.

4.5.4. Problem progression

Problem progression, first sketched by Jackson [41], provides a way to derive specifications from requirements (or need in our context) in a systematic fashion. Problem progression removes domains from the environment whilst altering the requirement to compensate; as the complexity of the environment is reduced, the detail of the requirements increases until we are left with a specification of the solution [42].

Problem Progression has the following form:

$$\frac{E\ (S)\ \square_{PO}\ N}{[D, E](S)\ \square_{PO}\ N}\ \text{[Problem Progression]}$$

Like Solution expansion, Problem Progression is a syntactic rewriting of the conclusion judgement, with the relationship between the old need N, the new need N , E, E and D as defined by Li's rewriting rules [42].

Example 7. $Lucia$'s environment includes the complex $WStat$ domain which controls the rl phenomenon. We might wish to remove this domain from the model by progressing it. To do so we refocus the problem to depend on $IRTA$'s receipt of rl rather than $WStat$'s control of it, adjusting the $Need$ to compensate, and internalising the newly unshared specification phenomena:

$$\frac{iPad_{alert}(IRTA^{alert})\ \square_{Jon}\ SDNeed^{\ alert}_{rl}}{[WStat^{rl}, iPad_{alert}](IRTA^{alert}_{rl})\ \square_{Jon}\ SDNeed^{alert}_{rl}}$$

where $SDNeed$ has description 'If_$IRTA$_receives_rl_then_send_$alert$'.

Following Li [42], subsequent applications of problem progression allow the removal of the $iPad$ domain, leaving a high-level specification for the IFTTT solution.

4.5.5. Stakeholder replacement

Under specified conditions, one stakeholder can rely on the judgement of another:

$$\frac{E(S)\ \square_{PO'}\ N}{E(S)\ \square_{PO}\ N}\ \text{[Replace Stakeholder]}$$

PO accepts PO''s judgement, with specified conditions

This is the example we began with (Section 4.4) of $Lucia$ putting herself in Jon's place.

This concludes the description of our transformations.

4.6. Design trees

Gentzen's calculus creates proof trees; software problem transformations combine to produce *design trees*. When all branches of a design tree are complete, the problem is solved: a fit-for-purpose solution is determined to the satisfaction of the problem owner, supported by the design rationale.

Example 8. From $Lucia$'s development steps we can build the following completed design tree:

$$\dfrac{\dfrac{\dfrac{}{[WStat, iPad](IRTA) \ \square_{Lucia} \ SDNeed}}{[WStat, iPad](IRTA) \ \square_{Jon} \ SDNeed}}{E(S) \ \square_{Jon} \ N}$$

$IRTA$ performs to $Lucia$'s satisfaction

Demonstration to Jon

Apply Environment Interpretation $[E/[WStat, iPad], N/SDNeed, S/IRTA]$, found in consultation with Jon and by inspection of the environment

From the properties of the substitution rules, we can read off both the solution to $Lucia$'s developer problem and the constructed design rationale.

Note: the design rationale created in this example is extremely weak, and yet is still fit-for-purpose in this context. In general, the strength of the rationale will be determined by the various stakeholders' needs.

4.7. Process considerations

Our theory does not constrain the order of application of problem and solution exploration steps; in particular, there is no necessity for problem solving to arrive at a perfect (or any!) problem description before beginning solution exploration[7]. Problem and solution spaces interleaving is present even in the earliest SE process models (for instance, [43]) as well as being the basis of modern iterative and incremental methods, for instance, [44, 45].

Both problem and solution exploration are resource intensive – a requirements engineer will, for instance, expend resources exploring the problem with stakeholders to draft software requirements, to capture domain descriptions, *etc*; a software developer will spend their time architecting and prototyping, collaborating in scrums, writing test harnesses and documentation, consulting end-users, *etc* as they explore the solution. Moreover, the problem solving process is not 'one-way' towards the solution: not all forward steps in a development will necessarily be correct as. Environment and need descriptions may be incorrect, ambiguous or incomplete; solution descriptions and justifications likewise. Indeed, *even when previously validated by a problem owner*, later problem solving may lead only to 'dead ends' whence *backtracking* to a prior problem solving state will be required to continue from. In the worst case, this will require the complete reconsideration of the original problem, losing all expended problem solving resource.

Both problem and solution exploration, therefore, commit resources to uncertain outcomes; this constitutes developmental process risk which must be

[7] Indeed, exploration of the solution space may precede exploration of the problem space.

managed. Stakeholder validation is one way to manage such risks: problem and solution understanding are checked during development with appropriate stakeholders which lowers the risk of precipitous commitment of resources to poorly understood problems and solutions [2, 46].

The possibilities for interaction of exploration and validation steps are shown in our problem solving process pattern in Figure 3[8]. The pattern is not intended as a one-size-fits-all model for a problem solving process: it does not, for instance, mandate any specific sequence of exploration steps, nor how much exploration should be completed before validation, nor that problem validation should occur before solution exploration begins (or *vice versa*). Instead, in characterising the essential relationships among problem solving steps, it is intended to serve as a process *generator* (cf. [47]).

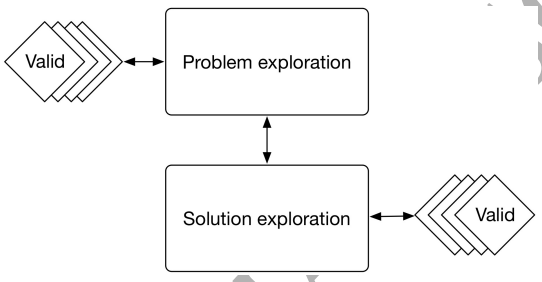


Figure 3: The macro sequencing of problem solving steps with validation checkpoints.

5. Solutions as technology combinations

We have so far discussed a theory of problems in which the solution to a problem is pure software. We have seen how it allows the expression of software problems and of step-wise processes for their solution. We have argued that its logical basis allows us to bridge between the formally described machine and the non-formally described world, an endeavour essential to SE. We have exemplified salient features of the theory and the way it accounts for key SE concepts and practices, such as sense-making, the use of architectures, the 'tangling' of software problems that necessitates co-design, and the use of validation as the means to mitigate developmental risks.

The development of our theory has not, however, relied on the nature of software as the solution domain. Given that we began with a specialisation of Rogers' definition of engineering to software, that we do not rely on the software nature of the solution suggests that we should explore the extension of our theory in the context of Rogers' full definition.

There appears very good motivation to do this. Firstly, as Rogers [48] writes:

[8] Previously called the POE Process Pattern, or PPP, in [2].

> an engineer is likely to make use of a number of different technologies in pursuit of [their] aim.

Currently, the special theory's focus on software as sole solution technology prevents us from treating other problems that involve software as part of a collection of solution technologies *within the theory*. To be clear, our special theory has been applied to real-world problems but non-software solution technologies were treated outside of the theory; see, for instance, [49]. An homogeneous theory would allow software and other solution technologies to be dealt with together.

Secondly, although it is technically feasible to conduct the engineering of software components of a system solution in isolation from that of all the other solution technologies, practically, this places more or less unsatisfiable constraints on the real-world engineering context. For instance, Royce's model [43], inflexibly predicated on a solid baseline of software requirements being isolated from system requirements before software engineering begins, is known to be deficient in volatile contexts [50]. In an homogeneous theory there would be no necessity to partition the overall need into those for separate solution technologies, enabling more flexible, iterative, interdisciplinary processes involving various solution technologies. This enables evolutionary system gains that can be achieved through iteration between various solution technologies; see, for instance, [44].

Lastly, Rogers' definition of engineering is solution technology independent and we would have a theory that supports it. The benefits include the possible discovery of commonalities between the engineering sub-disciplines and the potential for the transfer of knowledge, processes and techniques across sub-engineering discipline boundaries.

Our homogenous theory is based on *engineering problems*, i.e., problems:

$$E(S) \; \square_{PO} \; N$$

in which S ranges over all combinations of solution technologies including, but not limited to, software.

That S is not limited to software means that we must consider solutions involving both discrete and continuous phenomena (and their interplay). Other than extending the phenomenological basis of solution elements, so that they can be considered to encompass both discrete and continuous behaviours, this extension does not lead to changes in the theory as presented so far. Moreover, as software problems are a special case of engineering problems, the special theory is contained within the homogeneous theory.

In the following we discuss what more can be said in our homogenous theory.

5.1. Requirements engineering

The Reference Model of Gunter et al. [51] describes the fundamental relationships between requirements engineering artefacts, the goal being to characterise the specification of a software system as the interface between requirements

engineering and software development. Accordingly, requirements engineering relates five artefacts: domain knowledge (environment), requirements, a specification, a program, and a programming platform (system or machine) (see Figure 4). The essential characteristic of a specification is that it:

> occupies the middle ground between the system and its environment [and] provide[s] enough information for a programmer to build a system that satisfies the requirements [without reference to those requirements]. ([51, Page 38])

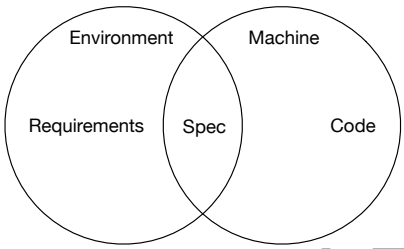


Figure 4: The two ellipse model: $Environment$, $Requirements$, specification $Spec$, forming the *Requirements problem* P_Req; with the specification $Spec$, program $Code$ and $Machine$ forming the *Development problem* P_Dev (c.f., [51])

The Gunter model can be framed thus in our theory: requirements engineering solves the problem of producing a specification from environment and need descriptions; software development solves the problem of producing a software artefact that, when embedded in a machine, satisfies the specification. That is, given problem owner PO and developer Dev there are two problems:

P_{Req} : $Env(Spec) \sqsupset_{PO} Req$ and P_{Dev} : $Machine(Code) \sqsupset_{Dev} Spec$

We note:

- these problems tangle as they are both expressed using $Spec$;
- as written in P_{Req}, $Spec$'s description is indicative, in P_{Dev} is it optative;
- as $Spec$ is not software and so is not the solution to any software problem. Rather, we must be able to work in the solution space of specifications.

We note, however, that this does not prevent a similar development within the special theory as we can write:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Machine(Code) \sqsupset_{Dev} Spec}{Machine(Code) \sqsupset_{PO} Spec}\ \text{Acceptance testing by } PO}{\ldots}\ [\textsc{Problem Progression}]}{[Env, Machine](Code) \sqsupset_{PO} Req}\ [\textsc{Problem Progression}]}{Env(2ellipse[Machine](Code)) \sqsupset_{PO} Req}\ [\textsc{Solution expansion}]}{Env(S) \sqsupset_{PO} Req}\ [\textsc{Solution interpretation}]\ \text{Two ellipse model}$$

$$\dfrac{Machine(Code) \sqsupset_{Dev} Spec}{Machine(Code) \sqsupset_{PO} Spec}\ [\textsc{Replace Stakeholder}]$$

See Section 6.5.3 for further discussion.

- the environment in which the computational device $Machine$ operates is seldom benign; a common example is a single event upset (SEU) caused by environmental phenomena ([52, 53]). In essence, an SEU induces unspecified behaviour modes in $Machine$ which can cause the $Code$ to malfunction.

  If $Env^{\lightning}$ is an aggressive environment (i.e., and environment that 'controls' SEUs) then we cannot avoid $Machine_{\lightning}$ in which, in the worst case, $chaos$ is the outcome of an observed $\lightning$ event. Thus, from a developmental perspective, a solution to

  $$Machine(Code) \ \square_{Dev} \ Spec$$

  is not necessarily a solution to

  $$Machine_{\lightning}(Code) \ \square_{Dev} \ Spec$$

  i.e., in an aggressive environment, any development of $Code$ which is decontextualised from the $Machine$ will not satisfy the $Spec$ when deployed. This has important ramifications for the application of formal methods that are decontextualised from the $Machine$.

5.2. Socio-technical problems

Extending the Reference Model for Requirements, Hall and Rapanotti [54] introduce the three ellipse model for socio-technical systems in which human and technological ellipses are solution technologies (see Figure 5).

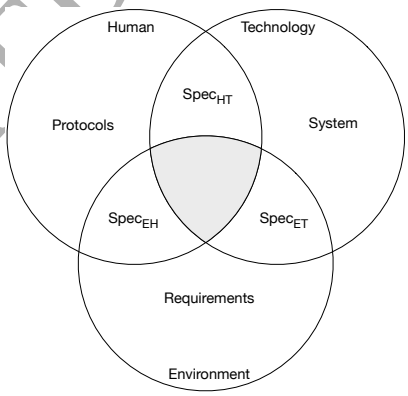


Figure 5: The three ellipse model of socio-technical systems, adapted from [54].

As in the two ellipse model, we begin with $Env(S) \ \square_G \ Req$, this time interpreting the solution with $3ellipse[Human, Tech](Proto, Sys)$, and then repeatedly applying PROBLEM PROGRESSION until only the targets — the human and

technical subsystems — remain:

$$\dfrac{\dfrac{\dfrac{Human(Proto) \;\square_{PO}\; Spec_EH}{\cdots}\,[\textsc{Prob Prog}]}{[Env, Human, Tech, Sys](Proto) \;\square_{PO}\; Req}\,[\textsc{Prob Prog}] \qquad \dfrac{\dfrac{Tech(Sys) \;\square_{PO}\; Spec_ET}{\cdots}\,[\textsc{Prob Prog}]}{[Env, Human, Tech, Proto](Sys) \;\square_{PO}\; Req}\,[\textsc{Prob Prog}]}{\dfrac{Env(3ellipse[Human, Tech](Proto, Sys)) \;\square_{PO}\; Req\;[\textsc{Soln Interp}]}{Env(S) \;\square_{PO}\; Req}\;\text{Three ellipse model } AStruct}\,[\textsc{Soln Expn}]$$

which characterises $Spec_{EH}$ and $Spec_{ET}$. The resulting problems are expressed in terms of phenomena from $Spec_{HT}$ and so tangle and so must be co-designed, i.e., the interplay of social and technological subsystems must be determined.

This area is well known in the IS literature: Fitt's early MABA-MABA lists [55], Price's Decision Matrices [56] and, more recently, Dearden's scenario method [57] all address it, suggesting that thought should first be given to the social subsystem which will constrain the $Spec_HT$ specification to bias the social aspects of the interface. The modern tendency is, however, for commercial o□-the-shelf information systems which require the social subsystem to flex to the demands made by the technical ones.

5.3. Validation problems

Validation problems [2] are a large class of problems that arise during engineering that do not have software as a solution. The general form of a validation problem is:

$$Validation\_Env(Validation\_Artefact) \;\square_{Validator}\; Validation\_Need$$

which describes the situation in which the need is for a validation artefact which, when considered in a specific validating stakeholder's *validation environment*, satisfies their validation needs.

Example 9. A safety case [58] is an example of a validation artefact, whose validating stakeholder is the regulator:

$$Regulatory\_Environment(Safety\_Case) \;\square_{Regulator}\; Regulatory\_Requirements$$

Note: validation problems exist alongside the customer's original problem which cannot be said to be solved unless we have both (a) constructed a system that solves the customer's problem *and* (b) constructed a safety case that satisfies each validating stakeholder. There will be one validation problem to solve for each validator: if there are n_p problem validators and n_s solution validators then, together with the Customer's problem, there are $n_p + n_s + 1$ problems to solve.

5.3.1. Fractal Problem Solving and Trusted Problems

Validation is a mechanism for managing developmental risk. However, preparation for validation itself — the solving of the various embedded validation

problems — can also be an expensive exercise and so there is a necessary trade-o between validated and unvalidated problem solving to be considered in any development.

That validation problems arise during problem and solution exploration indicates that problem and solution exploration incorporate problem solving processes in themselves. This leads us to define a form of problem solving process composition which is the embedding of problem solving instances within problem and solution explorations, leading to the upper part of Figure 6. This we term *fractal problem solving* in cognisance that validation problems may again have embedded validation problems associated with them, the embedding process proceeding until problems are encountered in which validation is not required. In not requiring validation, these problem solving activities are, in some sense, *trusted*: without such trusted processes there would necessarily be an infinite problem solving regress in which validation deeper and deeper within fractal problem solving instances is required[9].

One might expect trusted problem solving processes to accumulate risk; this is not the case when (a) failure has no cost associated with it; or (b) when the problem has been solved before. Even in the safety-critical context, determining the validation context for a safety case can be achieved without damaging the development process's ability to deliver a fit-for-purpose solution.

More formally, problem transformations replace a conclusion problem with a collection of premise problems and a justification. Within our homogenous theory, we may consider a problem transformation as a problem in itself. We observe the following equivalence:

$$\frac{P_1 \ldots P_n}{P}\ _J \equiv [P](PX[](P_1, \ldots, P_n, J)) \ \square_{Dev}\ \text{Each } P_i \text{ solvable} \quad J \text{ satisfies } PO$$

in which PX is the $AStruct$ for 'problem transformation' that should be read as, given conclusion problem P, find i) premise problems and ii) step rationale that, in Dev's estimation will i) be solvable and ii) are reached from P using J that will satisfy PO that the step is correct.

We note that the PX $Astruct$ expands, as expected, under SOLUTION EXPANSION to leave n + 1 premise problems: n 'P_i' problems and a single 'J' problem. Given this, the question arises whether to attack the 'P_i' problems or the 'J' problem first. The second option, determining the step rationale first, we have previously called *assurance-driven development* [2].

Through this equivalence, in terms of the process model of Figure 3, both problem and solution exploration can be seen as problem solving activities, as illustrated in Figure 6. In e□ect, problem solving is fractal with a problem owner's delegate discovering and solving problems on their behalf, and with the possibility that the delegate will, themselves, employ a delegate. This leads us to a potential theoretical di□culty with the termination of the problem solving process: when does delegated problem solving stop? In practice, the delegate

[9] This role has also been suggested for Simon's 'satisficing' ([59]).

problem solver will, at some point, have to rely on themselves to determine whether correct descriptions have been found, i.e., they will employ trusted problem solving processes.

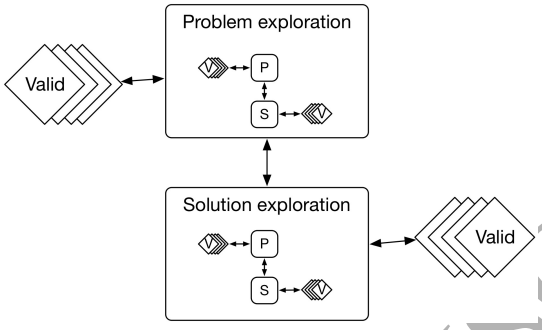


Figure 6: The Process Pattern Generator including embedded (or fractal) problem solving processes in both Problem and Solution Exploration.

In the macro sequencing of problem solving steps shown in Figure 3, backtracking due to unsuccessful validation was limited to the accumulation of resources between problem and solution exploration phases. This limited the risk in scope, if not in any practical sense. With the more complex process patterns shown in Figure 6, the potential for backtracking, and the accumulation of developmental risk, increases to include:

- all prior problem solving instances arising through sequential composition;
- all problem solving instances arising through parallel composition;
- all fractal invocations, such as validation problems.

In addition, instances of the processes pattern generator can be sequenced or run in parallel.

5.4. Problem Solving heuristics

Although Polya's problem solving context is the mathematical 'problems to find' [60, Page 77], his *Decompose and Recombining* heuristic:

> You decompose the whole [problem] into its parts, and you recombine the parts into a more or less different whole. ([60])

provides a possible interpretation for Definition 8: the upwards direction corresponds to decomposing whereas the downwards direction corresponds to recombining. Polya's assertion with this heuristic is that one should first understand the problem as a whole better to be in a position to judge which particular elements may be the most essential. This is similar to the *problem structuring* basis of the Problem Frames approach (PF, [41, Page 4]).

On decomposition, however, PF acknowledges that subproblems may interact, and introduces the *interference concern* to be considered alongside the decomposed subproblems. Definition 8 does not include an explicit interference concern problem although, if we wished to embed PF within our theory, one such could be added with the following specialised PF rule; suppose P is the original PF problem, P_i the decomposed problems, and IC the problem corresponding to the interference concern discharged by some part of J, then we have

$$\frac{P_1 \quad \ldots \quad P_n \quad IC}{P}\ {}_{\mathsf{J}}\ \text{[Problem Frames Step]}$$

6. Evaluation

In this section we discuss empirical work and other secondary evidence in support of our theory, organised according to Gregor's core qualities of a theory as recalled in Section 2.2. We conclude the section with an acknowledgment of known limitations and of current research to overcome them.

6.1. Description and analysis

In the previous sections, we have given an account of SE phenomena of interest which can be described and analysed in our theory. In particular, we have argued its ability to account for step-wise problem solving which occurs when software solutions are engineered, explicating the key relationships between artefacts and actors in the process and the classification of recurrent steps as problem transformation classes.

The empirical evaluation of the theory is ongoing. An initial comprehensive proof-of-concept application can be found in [61] where it was exercised on a complex software problem from the literature to establish the extent to which the encoding could capture common SE practices. More substantial empirical evidence from practice has been accumulated through application to a number of industrial case-studies, work which continues. It is worth noting that real-world application motivated the move from the special to the homogeneous theory: even in small scale industrial studies it was apparent that software plays only a part in the solution to real-world problems.

Work with General Dynamics UK included a series of safety-critical avionics case studies [49, 62–66] exploring the interplay of bespoke software and hardware components. The analytical and descriptive properties of the theory were put to test in the early phases of that company's integrated safety development process, and fine tuned as a result. The theory performed well, allowing the characterisation of key steps and actors in the process, with an early form of the process pattern of Section 4.6 emerging as an idealisation of the relationship between validation and design. Moreover, problem descriptions proved particularly fruitful by allowing requirements and early architectural design models to be developed which could be formally verified via existing model checking tools

[63], contributing evidence for early life-cycle assurance (captured in our theory via validation problems), something considered problematic across the industry.

In [67] the theory was applied within a financial institution. In this case the problem solved concerned the process of identification and elimination of bugs within mortgage calculation software. The system was sufficiently complex for two possible solution strategies to be identified, each with its cost and risk implications. The problem solving process was characterised and reengineered through our theory with a concomitant reduction in cost for the customer.

The prominent role of validation in problem solving was also noted in [68], which tested the theory on the live development of a socio-technical system. Of particular note was an extension of the previous observation on how assurance influences design in the context of developing socio-technical mission critical solutions: as for the safety-critical case, the resource expenditure associated with addressing validation problems was at least as high as the design effort. It was in this work that we observed that step-wise problem solving also applies to validation problems. The study combined diverse description languages, those for technical and social components providing some evidence that our theory works well.

6.2. Explanation

There are many ways in which the theory is explicative, in the sense of being able to address 'how, why, and when things happened, relying on varying views of causality and methods for argumentation' [3].

In the retrospective safety-critical avionics case studies mentioned earlier, a safety analysis performed on early problem models was able to predict safety defects that had, according to the company logs, emerged only much later, and that had required costly intervention and re-design [66]. This observation led to the addition of extra exploration and validation steps to the process for subsequent projects. Similarly, as part of the study in [67], the process pattern (see Section 4.7) was applied to an existing business process used by the organisation to deal with software defects reported by customers: the process was complex, involving a large number of stakeholders across a supply chain distributed globally. In this case too, the pattern application highlighted a lack of problem validation by relevant stakeholders early on in the process, pointing to a possible root causes for the high incidence of design rework recorded by the company. As a result, a formal validation step was introduced in the process by the company, which subsequently reported a reduction in the amount of design rework.

The process pattern can also be usefully employed to account for differences in development process models, as demonstrated in [47], which provides a theoretical characterisation and comparison of well-known software development process models from the literature.

The phenomenological basis of our theory is particularly suited to the development of arguments as to the expected behaviour of the designed solution in its environment, which goes somewhat towards satisfying Staple's [6] desire

for product theories. In fact, there is a sense in which our definition of engineering problem, coupled with the step-wise construction of the accompanying rationale (see Section 4.4), acts as a generator for practice-specific [69] product theories (more on this in Section 7). This was demonstrated empirically in our safety-critical case studies, as already mentioned, where early life cycle phenomena-based problem descriptions could be used as the basis of formal model checking and of *preliminary safety analysis*, a form of solution validation, able to: confirm any relevant hazards allocated by the system level hazard analysis; identify if further hazards need to be added to the list; and analyse the architectural description to validate that it can satisfy the safety targets associated with the identified relevant hazards.

An explanation of important relationships between key SE artefact is also embedded in the transformation classes we have defined. For instance, the AStruct in solution interpretation and expansion (see Section 4.5) can be seen as a general characterisation of problem decomposition, able to capture many types of decomposition found in practice, from design patterns [61, 70] and architectural styles [34] to viewpoints.

6.3. Prediction

Prediction refers to 'what the theory predicts will happen in the future under specified conditions, with an understanding that in IS (and SE) such predictions are often only approximate or probabilistic' [3].

We have already commented on how problem descriptions generated through application of the theory in practice can be seen as product theory, hence able to provide approximate prediction of how the solution would behave in its environment. In all the case studies reported, these were used to build assurance arguments for the purpose of validation, hence to drive subsequent design phases. The fact that successful systems were then developed on the basis of this provides some confidence that such predictions might have been sufficiently accurate, although this was not measured in our studies, so we cannot provide any empirical evidence as to their level of accuracy.

Similarly, the process pattern applications we have discussed accurately predicted process improvements, which were subsequently reported by the industrial partners: in this case too, precise measurements were not sought. However, more recent research has aimed to use the pattern as an instrument for more precise predictions, based on stochastic models which can be associated with it. For instance, Kaminsky and Hall [46] encode the process pattern as a discrete-time Markov chain model [71] in which the various probabilities correspond to problem solving team experience: it is more likely that a more experienced team's problem description will, for instance, be validated than will that of a less experienced team. Thus, the validation needs, and ultimately the risks incumbent, of different teams can be factored into the model.

Finally, there is an aspect of our theory which is of relevance to prediction. By analogy to proof-theory, where tactic languages [72] are used to generate proof semi-automatically, we have developed a design tactic, or *dactic*, language [73], by which designs can be captured for replayed in different environments.

Because of their angelic nature [72], dactics never generate 'false' solutions: the worst that can happen is that no solution will be found. Thus, the system 'learns' by cascading all dactics together. Of course, for the language to be of general use, an infeasibly large repository of dactics would be needed. However, in sufficiently restricted domains, it may be a feasible approach, although this remains a purely theoretical proposition at this point in time.

6.4. Prescription

While not prescriptive per se, our theory is suggestive of ways in which it can be applied in practice. For instance, the specific form of our problem descriptions suggests ways in which growing knowledge of problem and solution should be elicited and represented for further analysis and design, providing a problem-based approach for requirements engineering. Alongside the already mentioned case studies, this was explored in other industrial applications, such as [74, 75].

Another aspect of theory which has led to specific methods in application concerns the way the rationale is constructed. As its specific form is not mandated by our theory, it may take diverse forms depending on the context of application. For instance, a specific approach is proposed in [49] for safety-critical engineering, subsequently adopted and extended in other industrial work [76], which identifies specific safety concerns and risks that need discharging as part of the argument, all related to the type of evidence which constitute the safety case for an independent certification authority. A different approach is taken in [68], where a mix of formal and informal statements were used depending on the role and needs of the validating stakeholders in the organisation: for instance, some required stringent criteria to be met, e.g., finance or quality control stakeholders, while others were interested in technical feasibility or impact on personal workload.

6.5. Current research

6.5.1. Tangled problems

In the theory presented we have only considered the case in which a development proceeds from an initial problem, with more problems subsequently generated as part of the problem solving process. In particular, we have noted how $AStructs$ can be used to capture various forms of problem decomposition, including the case of co-design problems arising when a solution is structured via an architecture. As observed in Section 4.4, co-design problems are examples of tangled problems, that is problems whose relationships are such that their naive decomposition into nominally independent sub-problems is unlikely to lead to an overall solution. Indeed, many real-world situations can be characterised as tangled problems: [77] explores socio-economic relationships as the solution to pairs of tangled problems.

When a real-world problem presents itself as a complex tangle, a different approach is required: rather than a single abstract problem to start with, we need ways to identify and represent the various sub-problems and the way they

tangle. In [77–79] we have started to explore possible structuring of tangled problems based on the regular structure of our defined engineering problems, and the notion that two problems tangle when they share phenomena. Such tangling then constrains their solutions and the method by which solutions can be arrived at.

6.5.2. Change problems

Our problem definition assumes that a *new* solution is to be designed in a given environment, whose indicative properties are established through a knowledge elicitation process. This is a *greenfield* engineering process, in which something new is created to a□ect the environment so that the need is satisfied. As such there is no notion of 'change' as a first class element of our described theory: for instance, if meeting a new organisational need could be obtained by changing an existing information system, than the starting point would not be a blank canvas, but that information system which may be exchanged or altered to meet the need. Current research [79] builds the notion of *change problem* on top of our theory, exploring its relationship with the definition of greenfield problem above, and the implications on the process pattern and transformation classes. Initial results suggest that there exists a complex translation from a change problem to a collection of greenfield problems [80], so that the theory underlying change problems remains as defined in this paper.

6.5.3. Creativity

Hatchuel and Weil [81] argue that design is more than problem solving, saying that 'creativity cannot just be 'added' to problem solving theory.' Rather, their CK Theory locates it in propositions that have no proof value (so called 'concepts') in the knowledge base initially available to the designer. Similarly, our theory works with descriptions of solutions which have no validatable status until a stakeholder relative judgement is made of them. However, we also have a more tangible location for creativity within our theory: the invention of new AStructs that will structure the solution space in previously unknown ways; creativity is thus identified with another problem, that of finding the right AStruct. We note that the Hatchuel and Weil [81] characterisation of creativity appears to provide no vehicle for creativity in the problem space, as one might do when, for instance, seeing a problem in a new light. In contrast, in our theory, embedded problem solving activities in both the problem and solution spaces allows inventive AStructs to be used in both problem and solution spaces.

In Section 5.1, we posited a relationship between special and homogeneous developments via a specification predicated on the assumption that a sequence of Problem progressions is possible. It is an open problem as to whether this is always the case and detailed examination of Li's transformation [42] would be necessary to confirm this. If it is the case, then we have a form of *cut-elimination* ([21]), an important and hard fought property of a logical system that can be used to demonstrate THE internal consistency of a logic. Moreover, cut has been linked to creativity in proof [82, 83]. Whether cut-elimination is a property of our theory is an avenue for future investigation.

6.5.4. Practicality

In our case studies no particular difficulty was reported by practitioners in relating the underlying principles of the theory to practice. However, in devising ways to apply it in organisational and industrial settings a number of difficulties were noted, particularly in relation to the identification of appropriate phenomena and the generation of problem descriptions: both choices of level of granularity of phenomena and of their aggregation into domains are not straightforward and, in general, more guidance was needed beyond what the theory provides. Also, keeping track of the problem solving process and all generated artefacts was found too hard and time consuming to do by hand, which limits both scalability and scope of application in real-world practice.

Therefore for the theory to generate practically usable and scalable methods more work is required both in terms of guidelines and heuristics, and critically of developing some level of tool support and automation. Prototype tool support exists for a subset of POE with the prototype POElog tool [84]. We are currently investigating whether Little-JIL [85] could offer further support.

6.5.5. Kolmogorov's method-theoretic approach

As an interpretation of intuitionistic logic, Kolmogorov [86] outlines a calculus for the solution of a class of mathematical problems, including geometrical constructions. An example of this class is:

> "Using only straightedge and compasses, draw a circle passing through the vertices of a triangle."

Under this interpretation, the meaning of a problem is the method of its solution: the calculus of problems is *method-theoretic*. The meaning of, for instance, $a \quad a \quad a$ is a mapping from the method of solution of a to the method of solution of $a \quad a$.

Theoretically, it is important to reconcile our theory with Kolmogorov. An embedding is not difficult: we can build that propositional fragment including conjunction and disjunction around judgements; for instance, we have, trivially:

$$\frac{E_1(S_1) \ \square_{PO_1} \ N_1 \qquad E_2(S_2) \ \square_{PO_2} \ N_2}{E_1(S_1) \ \square_{PO_1} \ N_1 \qquad E_2(S_2) \ \square_{PO_2} \ N_2} \ \text{-Introduction}$$

For negation, the situation is less clear. For, although we can say $\neg E(S) \ \square_{PO} \ N$ *holds when the design tree for* $E(S) \ \square_{PO} \ N$ *ends in a known unsolvable problem*, we would also need the result that no problem can be both solved and reduced to known unsolvable problems, which looks unlikely given the unconstrained nature of the step rationale: a stakeholder that always validates causes problems. More work is required here.

We cannot expect, however, that our theory is contained in Kolmogorov's. In his critique of existential propositions [86], Kolmogorov identifies two elements of a [mathematical] problem: an *objective element* (the problem) and a subjective element (the solution). The problem is objective as it is 'independent of our knowledge'; the solution is subjective as it is dependent on our knowledge.

Within our theory, however, we observe that environment and needs can change during problem solving so we should not expect knowledge independence. With respect to Rittel's wicked problems [18], an interesting conclusion is that we should not expect that a problem, once solved, need remain solved.

7. Related Work

7.1. Design theories for software engineering

Wieringa et al. [69] make the case for the utility of design theories in SE. They discuss 'the engineering cycle' as a logical model including tasks such as *problem investigation* (akin to our problem exploration), which may or many not include an evaluation of a current implementation (for us, this would be a form of problem validation), *treatment design* and *treatment implementation* (akin to our solution explorations), and *design validation* (in our theory, a form of solution validation). In contrast to our process pattern, Wieringa *et al*'s engineering cycle is very specific as to what each of the tasks should entail: this is because the cycle is used as a reference model, i.e. a 'rational reconstruction' of the problem solving process [87], rather than a process analyser and generator as is the case of our pattern. Moreover, with no explicit consideration of risk and resources, it may be argued that the cycle is purely descriptive. Within the cycle, Wieringa [87] stresses the different problem-solving nature of tasks: for instance, problem investigation is seen as addressing *knowledge problems*, whose aim is to change our knowledge of the world, while treatment design addresses *practical problems*, that is how to change the world so that it better meets stakeholder goals. The key observation is that both practical and knowledge problems are nested within the problem solving process, something we also acknowledge in our theory. Also, in design validation, an argument is needed to justify that a solution both causes the desired effects in its environment, and that it satisfies other criteria specified by stakeholders [87]. Indeed, both elements are also present in our theory: problem models refer to the mechanisms by which solution and environment phenomena interact to satisfy a recognised need, with further validation criteria elicited and checked though validation activities.

A form of design theory which Wieringa et al. [69] see as particularly fruitful to SE is one of an *architectural* nature (as opposed to logical theories, such as deductive axiomatic theories): this type of theory is one which provides a model of mechanisms that produce phenomena. For instance, an architectural theory of SE might be one providing a model of the mechanisms within a problem environment. By following this line of thought, our definition of problem as the juxtaposition of problem and solution phenomena sets (our domains), with their controlled and observed relationships, embodies the concept of architectural design theory.

Related to this is Weiringa et al.'s consideration of different levels of generalisation and idealisation a theory may provide [69]. At the lowest level, a theory might just be the expression of a practitioner's specific problem and treatment design theories (in our terms, a specific solved problem model): no generalisation or idealisation is present here. At the highest level, a theory might be

some universal truth, as might be the case of some natural science laws. For SE, [69] supports the idea that we should aim for mid-range generalisations, to attain some level of generality, which is nevertheless applicable to particular cases in practice. Indeed our theories are of such nature: they generalise (and idealise) some common features of software problem solving, while still allowing the practitioner to define specific theories in their own particular context.

Finally, based on in-depth study of the SE literature, [69] addresses Gregor's epistemological questions by arguing in support of case-study research for the development of architectural design theories able not just to explain, but also predict: this is motivated by their observation that most SE theories were statistical, and not backed up by any theoretical or phenomenological law, hence hard to generalise beyond the observed phenomena. Indeed, striving for predictive, as well as descriptive, theories is also a goal of our research, as discussed previously.

Wieringa et al. [69] introduce a notion of *problem theory* as a theory for a particular problem environment or classes of problem, which is characterised by the identification of stakeholders, goals and and criteria, some symptoms, their diagnoses and possible negative implications, as well as some stakeholder evaluation. Although there are some obvious overlaps with our work, the lack of precise definitions makes any formal mapping a matter of pure interpretation, therefore we will not attempt one. Instead, we will stress the common philosophy of the two approaches in the need for an understanding of phenomena in context, which are strongly dependent on needs and criteria expressed by stakeholders: in our terms, the assurance-driven nature of SE problem solving. Similarly, Wieringa et al. [69] introduce a notion of *design theory* as a theory to explain in which way a designed artefact will contribute to meeting stakeholders' goal. Here the similarity with our approach is even stronger, mapping to the rationale accompanying any instance of a design tree.

Expanding on over four decades of research in SE development processes and practices, [44] details a principles-based process meta-model which allows practitioners to generate their own bespoke software development processes based on characteristics and constraints of their own development, organisation and market contexts, while still adhering to the key principles embodied in the meta-model. From an ontological perspective, the process meta-model is close to our process pattern, although intended primarily as a software life-cycle process generator, rather than an analyser: it is a 'spiral' model, recognising, like our process pattern, the iterative and incremental nature of successful development processes, and it includes explicit consideration of risk and risk-based stakeholder validation. Di□erent from our process pattern, which is highly abstract and generic, the meta-model provides a specific mapping of activities to traditional phases of software development, from initial problem exploration, through software development to operation.

The four principles underlying the meta-model were distilled though the accumulation of knowledge a□orded by decades of case study research, which is epistemologically similar to our research. They also align closely to our principled approach, which is theoretically pleasing as the two were developed in-

dependently and from diverse case studies across the industry. For instance, the 'stakeholder value-based guidance' principle encourages the inclusion and consideration of value propositions for all success-critical stakeholders, which is equivalent to the assurance-driven nature of our solving processes. Similarly, the 'evidence- and risk-based decisions' principles support the idea that decisions should be made based on evidence, to prevent the building up of project risk: in our process pattern this equates to tackling the validation problems associated with problem and solution explorations.

7.2. On mathematical approaches to design

The second part of this paper extends our theory into the general design context, presenting a mathematical approach to design. There are many mathematical approaches to design.

Specifically, our problem 'triple' notation is reminiscent of Zave and Jackson [88] (and others) who argue that the role of domain knowledge K is to fill the gap between requirements R and specification S, written as

$$S, K \quad R,$$

interpreted as a predicate in [51]. [89] argues that this formulation cannot be used as the basis of a design process as design is not always performed with complete information: specifically K and R, for instance, cannot be written down definitively at the beginning of design. Our theory adopts the unitary sequent-like triple [21]

$$E(S) \ \square_{PO} \ N$$

in which elements and their descriptions are completed during design through iteration and backtracking between problem and solution spaces, irrespective of the completeness of the descriptions therein.

More generally, General Design Theory (GDT, [90]) gives a mathematical formulation of the design process derived from an understanding of how design is conceptually performed. GDT is entity based: its *entity concept set* corresponds to the 'a designer's abstract mental impressions of the concrete entities that form a design solution'. Above this set is built GDT's function and attribute spaces characterising, in our terms, optative and indicative descriptions, respectively. A design specification in GDT is a point in the function space; a design solution a point in the attribute space; the process of design is the construction of a mapping between them. As in our theory, the construction is stepwise, in GDT via the *evolution* (or gradual refinement) of *metamodels* (finite sets of attributes). The design process prescription is *abduction* of design candidates, *deduction* of their properties and *circumscription* [91] on failure to solve the problem, followed by iteration. In our theory, such design cycles would be embedded within problem solving processes that are themselves embedded within problem explorations.

Axiomatic Design Theory (ADT, [92]) defines design in terms of vectors of functional requirements (FRs) in the functional domain and vectors of design parameters (DPs) in the physical domain, the design process being the linear

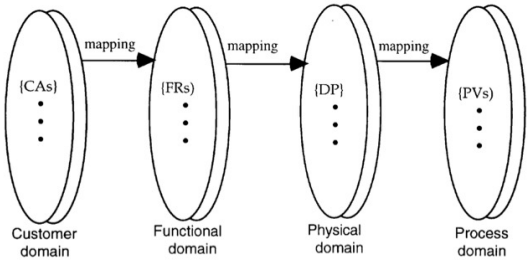


Figure 7: The four domains of the Axiomatic Design Theory design world with the characteristic vectors of each indicated [92].

algebraic construction of transformation matrices between those two. *Axiomatic* refers to the two foundational goals — the Independence axiom, that states that functional requirements should be kept independent; and the Information axiom, that states that we should minimise the information content of the design — that underpin ADT. Together, they ensure that the functional and physical domains are related by (at worst) triangular matrices meaning that one can solve for the optimal design by linear algebra. We have no analogue of these axioms in our theory; however, although the notions are not precisely defined, they appear to be related to phenomena:

- independence of functional requirements being that two such functional requirements should not refer to the same phenomena, i.e., they should not tangle;
- the minimisation of information being that elements of the design should constrain the fewest phenomena possible.

The process of design in ADT consists of moving from domain to domain, decomposing the characteristic vectors until the design can be implemented. Except in trivial cases, this decomposition cannot be completed by remaining in a single domain; rather, ADT introduces the essential notion of *zigzagging* [92] by which the characteristic vectors and their decompositions are navigated between; in essence, zigzagging connects higher and lower design 'levels' together. The backtracking described in Section 4.6 appears a similar mechanism.

In general terms, GDT and ADT are to *Hilbert-style deduction systems* [21] as our theory is to Gentzen-style deductive systems: Gentzen's[10] goal in [93] was an analysis of mathematical proofs as they occur in practice. The result was a logical system in which *proof actions* were primary, each represented by a deduction rule, similar to those we have presented. The alternative is the Hilbert-style in which properties of the interrelation of *objects* are characterised as axioms, with few deductive rules, typically only *modus ponens* and *generalisation*: for

[10] English translation in Szabo [20].

instance, the propositional axiom

$$A \quad (B \quad A)$$

relates propositions A and B. In particular, the Independence and Information axioms of ADT are of this form.

Whereas the two forms of system are equipotent, Gentzen-style is known better to support a constructive approach, a property we feel will benefit the description of the process of design.

8. Conclusion

This article has discussed a substantial design theory of SE, which embodies a view of SE as the practice of framing, representing and transforming SE problems, where our notion of problem is derived from the Rogers' definition of engineering. As such, the theory accounts for the way SE transforms the physical world to meet a recognised need, and for the problem structuring process in context. Moreover, the theory bridges Turski's formal and non-formal divide through its base in the propositional calculus, adopting and adapting the contextualisation tools that proof-theoretic-like semantics provide.

The theory is contextualised in the ongoing academic debate on the need and form of a substantial theory for SE. Both ontological and epistemological views of the theory were explored, and appropriate empirical evidence, from a body of work spanning over a decade, was brought to bear.

We have argued that the theory has both analytic, explicative and predictive properties, and is amenable to the definition of methods for practical application. We have also acknowledged current deficiencies and briefly outlined ongoing research towards addressing them both from a theoretical and practical perspective.

9. Acknowledgements

We are deeply indebted to the reviewers for their constructive comments that greatly assisted in the revision of this paper.

References

[1] P. Johnson, M. Ekstedt, I. Jacobson, Where's the theory for software engineering?, IEEE software (5) (2012) 96.

[2] J. G. Hall, L. Rapanotti, Assurance-driven design in Problem Oriented Engineering, International Journal On Advances in Systems and Measurements 2 (1) (2009) 119–130.

[3] S. Gregor, The nature of theory in information systems, MIS quarterly (2006) 611–642.

[4] S. Adolph, P. Kruchten, Generating a useful theory of software engineering, in: 2013 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE), IEEE, 47–50, 2013.

[5] P. Ralph, Developing and Evaluating Software Engineering Process Theories, in: Proceedings of the 37th IEEE International Conference on Software Engineering, IEEE/ACM, 2015.

[6] M. Staples, The Unending Quest for Valid, Useful Software Engineering Theories, in: 4th SEMAT Workshop on a General Theory of Software Engineering, IEEE/ACM, 2015.

[7] K. Smolander, T. Paivarinta, Forming theories of practices for software engineering, in: Software Engineering (GTSE), 2013 2nd SEMAT Workshop on a General Theory of, IEEE, 27–34, 2013.

[8] R. J. Wieringa, Towards middle-range usable design theories for software engineering, in: Proceedings of the 3rd SEMAT Workshop on General Theories of Software Engineering, ACM, 1–4, 2014.

[9] K.-J. Stol, B. Fitzgerald, Uncovering theories in software engineering, in: Software Engineering (GTSE), 2013 2nd SEMAT Workshop on a General Theory of, IEEE, 5–14, 2013.

[10] K.-J. Stol, B. Fitzgerald, Theory-oriented software engineering, Science of Computer Programming 101 (2015) 79–98.

[11] R. I. Sutton, B. M. Staw, What theory is not, Administrative science quarterly (1995) 371–384.

[12] P. Ralph, Possible core theories for software engineering, in: 2013 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE), IEEE, 35–38, 2013.

[13] P. Johnson, M. Ekstedt, Exploring Theory of Cognition for General Theory of Software Engineering, in: 4th SEMAT Workshop on a General Theory of Software Engineering (GTSE), IEEE/ACM, 15–24, 2015.

[14] J. G. Hall, L. Rapanotti, A framework for software problem analysis, Tech. Rep. 2005/05, Department of Computing, The Open University, 2005.

[15] G. F. C. Rogers, The nature of engineering: a philosophy of technology, Macmillan Press, 1983.

[16] W. M. Turski, And no philosophers' stone, either, Information processing 86 (1986) 1077–1080.

[17] H. A. Simon, The sciences of the artificial, MIT press, 3rd edn., 1996.

[18] H. W. Rittel, On the Planning Crisis: Systems Analysis of the" First and Second Generations", Institute of Urban and Regional Development, 1972.

[19] Z. Chaochen, C. A. R. Hoare, A. P. Ravn, A calculus of durations, Information processing letters 40 (5) (1991) 269–276.

[20] M. E. Szabo (Ed.), Gentzen, G.: The Collected Papers of Gerhard Gentzen, Amsterdam, Netherlands: North-Holland, 1969.

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

[22] V. A. Kruteťsïÿąkiĭ, I. Wirszup, J. Kilpatrick, The psychology of mathematical abilities in schoolchildren, University of Chicago Press, 1976.

[23] J. G. Hall, L. Rapanotti, M. Jackson, Problem Frame semantics for software development, Journal of Software and Systems Modeling 4 (2) (2005) 189 – 198, URL http://dx.doi.org/10.1007/s10270-004-0062-1.

[24] S. A. Kau□man, Articulation of parts explanation in biology and the rational search for them, in: Topics in the Philosophy of Biology, Springer, 245–263, 1976.

[25] S. Glennan, Rethinking mechanistic explanation, Philosophy of science 69 (S3) (2002) S342–S353.

[26] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall International, 1985.

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

[28] E. Best, R. Devillers, J. G. Hall, The box calculus: a new causal algebra with multi-label communication., in: G. Rozenberg (Ed.), Advances in Petri Nets, vol. 609 of *Lecture Notes in Computer Science*, Springer-Verlag, ISBN 3-540-55610-9, 21–69, 1992.

[29] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone, The synchronous languages 12 years later, Proceedings of the IEEE 91 (1) (2003) 64–83.

[30] IFTTT, ifttt.com, Last Accessed, 26 Feb, URL https://ifttt.com/myrecipes/personal, 2015.

[31] M. Turner, D. Budgen, P. Brereton, Turning software into a service., Computer. 36 (10) (2003) 38–44.

[32] B. Ur, E. McManus, M. Pak Yong Ho, M. L. Littman, Practical trigger-action programming in the smart home, in: Proceedings of the 32nd annual ACM conference on Human factors in computing systems, ACM, 803–812, 2014.

[33] K. E. Weick, Sensemaking in Organisations, Thousand Oaks, Calif. Sage Publications, 1995.

[34] L. Rapanotti, J. G. Hall, M. Jackson, B. Nuseibeh, Architecture-driven Problem Decomposition, in: 12th IEEE International Conference on Requirements Engineering (RE 2004), IEEE Computer Society, 80–89, 2004.

[35] M. Shaw, D. Garlan, Software architecture: perspectives on an emerging discipline, Prentice Hall, 1996.

[36] J. W. Ross, P. Weill, D. Robertson, Enterprise architecture as strategy: Creating a foundation for business execution, Harvard Business Press, 2006.

[37] ISO, ISO/IEC/IEEE 42010:2011; Systems and software engineering – Architecture description, Web-page, URL http://www.iso.org/iso/catalogue_detail.htm?csnumber=50508, 2011.

[38] G. E. Krasner, S. T. Pope, A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, Journal of Object Oriented Programming 1 (3) (1988) 26–49, ISSN 0896-8438.

[39] P. Kruchten, The Rational Unified Process: An Introduction (2nd Edition), Addison-Wesley Professional, 2 edn., ISBN 0201707101, 2000.

[40] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e, Pearson Education India, 2012.

[41] M. Jackson, Problem Frames: Analyzing and Structuring Software Development Problems, Addison-Wesley Publishing Company, 2001.

[42] Z. Li, Progressing Problems from Requirements to Specifications in Problem Frames, Ph.D. thesis, Department of Computing, The Open University, Walton Hall, Milton Keynes, UK, URL http://www.scm.keele.ac.uk/staff/z_li/PhD_Thesis.pdf, 2007.

[43] W. Royce, Managing the Development of Large Software Systems, in: Proceedings of IEEE WESCON, vol. 26, 1 – 9, 1970.

[44] B. Boehm, J. A. Lane, S. Koolmanojwong, R. Turner, The incremental commitment spiral model: Principles and practices for successful systems and software, Addison-Wesley Professional, 2014.

[45] Agility, Agile Alliance. http://www.agilealliance.org/home, last accessed November 2004, 2004.

[46] D. Kaminsky, J. G. Hall, Towards Process Design for E□cient Organisational Problem Solving, in: Proceedings of BUSTECH 2015, 2015.

[47] J. G. Hall, L. Rapanotti, Towards a design theoretic characterisation of software development process models, in: Proceedings of the 4th SEMAT Workshop on General Theories of Software Engineering, 2015.

[48] G. F. C. Rogers, The Nature of Engineering: A Philosophy of Technology, Palgrave Macmillan, 1983.

[49] J. G. Hall, D. Mannering, L. Rapanotti, Arguing safety with Problem Oriented Software Engineering, in: 10th IEEE International Symposium on High Assurance System Engineering (HASE), Dallas, Texas, 2007.

[50] V. Rajlich, Changing the paradigm of software engineering, Communications of the ACM 49 (8) (2006) 67–70.

[51] C. Gunter, E. L. Gunter, M. Jackson, P. Zave, A reference model for requirements and specifications, in: Requirements Engineering, 2000. Proceedings. 4th International Conference on, IEEE, 189, 2000.

[52] E. Normand, Single event upset at ground level, IEEE transactions on Nuclear Science 43 (6) (1996) 2742–2750.

[53] B. W. Johnson, Fault-tolerant microprocessor-based systems., IEEE Micro 4 (6) (1984) 6–21.

[54] J. G. Hall, L. Rapanotti, Problem Frames for Socio-technical Systems, in: P. Zaphiris, C. S. Ang (Eds.), Human Computer Interaction: Concepts, Methodologies, Tools, and Applications, chap. 2.22, Information Science Reference, 713–731, 2009.

[55] P. M. Fitts, Functions of man in complex systems, Aerospace Engineering 21 (1) (1962) 34–39.

[56] H. E. Price, The allocation of functions in systems, Human Factors: The Journal of the Human Factors and Ergonomics Society 27 (1) (1985) 33–45.

[57] A. Dearden, M. Harrison, P. Wright, Allocation of function: scenarios, context and the economics of e□ort, International Journal of Human-Computer Studies 52 (2) (2000) 289–318.

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

[59] A. Hatchuel, B. Weil, A new approach of innovative Design: an introduction to CK theory., in: DS 31: Proceedings of ICED 03, the 14th International Conference on Engineering Design, Stockholm, 2003.

[60] G. Polya, How to solve it, Princeton University Press, second edn., 1957.

[61] J. G. Hall, L. Rapanotti, M. Jackson, Problem Oriented Software Engineering: solving the Package Router Control problem, IEEE Trans. Software Eng. 34 (2) (2008) 226–241.

[62] D. Mannering, J. G. Hall, L. Rapanotti, Safety Process Improvement: Early Analysis and Justification, in: Proceedings of the 2nd Institution of Engineering and Technology Conference on System Safety 2007, 2007.

[63] D. Mannering, J. G. Hall, L. Rapanotti, Safety process improvement with POSE & Alloy, in: F. Saglietti, N. Oster (Eds.), Computer Safety, Reliability and Security (SAFECOMP 2007), vol. 4680 of *Lecture Notes in Computer Science*, Springer-Verlag, Nuremberg, Germany, 252–257, 2007.

[64] D. Mannering, J. G. Hall, L. Rapanotti, Towards Normal Design for Safety-Critical Systems, in: M. B. Dwyer, A. Lopes (Eds.), Proceedings of ETAPS Fundamental Approaches to Software Engineering (FASE) '07, vol. 4422 of *Lecture Notes in Computer Science*, Springer Verlag Berlin Heidelberg, 398–411, 2007.

[65] D. Mannering, J. G. Hall, L. Rapanotti, Problem oriented safety process improvement, in: Proceedings of the Safety-critical Systems Symposium 2008, Bristol, UK, 2008.

[66] D. Mannering, Problem Oriented Engineering for Software Safety, Ph.D. thesis, Open University, 2009.

[67] A. Nkwocha, J. G. Hall, L. Rapanotti, Design rationale capture for process improvement in the globalised enterprise: an industrial study, Journal of Software and Systems Modeling 12 (4) (2013) 825–845, online first (`http://www.springerlink.com/content/d45x17g438833069/`).

[68] L. Rapanotti, J. 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, 2009.

[69] R. Wieringa, M. Daneva, N. Condori-Fernandez, The structure of design theories, and an analysis of their use in software engineering experiments, in: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on, IEEE, 295–304, 2011.

[70] J. Overton, J. G. Hall, L. Rapanotti, A Problem-Oriented Theory of Pattern-Oriented Analysis and Design, in: Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, IEEE Computer Society, 208–213, 2009.

[71] H. C. Tijms, A first course in stochastic models, John Wiley and Sons, 2003.

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

[73] J. G. Hall, L. Rapanotti, Software engineering as the design theoretic transformation of software problems, Innovations in Systems and Software Engineering 8 (3) (2012) 175–193, DOI 10.1007s11334-011-0171-2.

[74] N. Eve, Using Problem Oriented Engineering as a generic process pattern for product design and development and comparing it against the traditional V-Model of software development, Master's thesis, Masters research thesis, The Open University, 2011.

[75] T. K. Mwangi, Counteracting IT Silo Culture in Public Organisations through Problem Oriented Engineering in the Requirements Phase, Master's thesis, Masters research thesis, The Open University, 2013.

[76] M. O'Halloran, An Integrated Approach to Software Development for Safety-Critical Systems using Problem Oriented Engineering, Master's thesis, The Open University, 2012.

[77] J. G. Hall, The Global Knot: How Problems Tangle in the World's Economy, in: M. D. Lytras, P. O. de Pablos, W. Lee, W. Karwowski (Eds.), Electronic Globalized Business and Sustainable Development through IT Management: Strategies and Perspectives, IGI-Global, URL http://oro.open.ac.uk/19439, 2011.

[78] J. G. Hall, L. Rapanotti, The discipline of natural design, in: Proceedings of the Design Research Society Conference 2008, Design Research Society, 2008.

[79] G. Markov, J. G. Hall, L. Rapanotti, An engineering framework for dealing with change problems: theoretical underpinnings and initial evaluation, in: Proceedings of the 15th International Conference on Knowledge, Culture and Change in Organizations and the Organization Knowledge Community, Berkeley, California, 2015.

[80] G. Markov, J. G. Hall, L. Rapanotti, POE-Δ: Towards an engineering framework for solving change problems, 2016.

[81] A. Hatchuel, B. Weil, CK design theory: an advanced formulation, Research in engineering design 19 (4) (2009) 181–192.

[82] R. McDowell, D. Miller, Cut-elimination for a logic with definitions and induction, Theoretical Computer Science 232 (1) (2000) 91–119.

[83] J. Hintikka, On creativity in reasoning, in: The complexity of creativity, Springer, 67–78, 1997.

[84] J. G. Hall, L. Rapanotti, POELog: a Prolog-based engine for Problem Oriented Engineering, Tech. Rep. TR2008/07, The Open University, 2008.

[85] A. G. Cass, A. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, A. Wise, Little-JIL/Juliette: a process definition language and interpreter, in: Software Engineering, 2000. Proceedings of the 2000 International Conference on, IEEE, 754–757, 2000.

[86] P. Mancosu, From Brouwer to Hilbert: The debate on the foundations of mathematics in the 1920s .

[87] R. Wieringa, Design science as nested problem solving, in: Proceedings of the 4th international conference on design science research in information systems and technology, ACM, 8, 2009.

[88] P. Zave, M. A. Jackson, Four Dark Corners of Requirements Engineering, ACM Transactions on Software Engineering and Methodology 6 (1) (1997) 1–30.

[89] H. Takeda, P. Veerkamp, H. Yoshikawa, Modeling design process, AI magazine 11 (4) (1990) 37.

[90] H. Yoshikawa, General design theory and a CAD system, Man-Machine communication in CAD/CAM .

[91] J. McCarthy, Circumscription—A form of non-monotonic reasoning, Artificial Intelligence 13 (1) (1980) 27 – 39, ISSN 0004-3702, URL http://www.sciencedirect.com/science/article/pii/0004370280900119.

[92] N. P. Suh, Axiomatic design theory for systems, Research in engineering design 10 (4) (1998) 189–209.

[93] G. Gentzen, Untersuchungen über das logische Schließen (Investigations into Logical Inference), Ph.D. thesis, Universität Göttingen (Translation in [20, 68–131]), 1935.