A design theory for software engineering

Jon G. Hall and Lucia Rapanotti

3 January, 2016

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

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk
A design theory for software engineering

Jon G. Hall, The Open University, UK
Lucia Rapanotti, The Open University, UK

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. This article contributes to the ongoing debate by proposing a design theory for Software Engineering.

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 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, 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 in the discussion.

Analytic, explanatory and predictive properties of the theory are also discussed, alongside acknowledged limitations and current research to overcome them, and how the theory relates to other work in the literature.

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. [2012] have argued the need for “significant” theories of SE, that is theories which can address significant questions within the discipline. While recognising that SE is not entirely lacking theory, 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.

Undoubtedly, 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 developed from our work on Problem Oriented Engineering (shortly POE, [Hall and Rapanotti 2009a]), a practical engineering framework with an accumulated body of work spanning over a decade, which includes practical application and evaluation through a number of real-world engineering case studies. In fact, we contribute two design theories. The first is a theory of pure software artefacts that solve problems. This first theory is, by its nature, limited in its possible applications as there are very few problems for which software provides all of the solution; even when software is the primary solution technology, other engineered artefacts, such as a safety case or even end-user training materials, will be needed. The second design theory remedies this defect so that software is one of many possible solution technologies that can form complex solutions.

By design theory, we mean a theory that characterises the elements of software problem solving in terms of the effect they have on the process of design. In this sense, we take a design theoretic approach that follows Gentzen’s proof theoretic approach to mathematical proof [Szabo 1969; Kleene 1964]. Of course, our universe of discourse is much less formal than proof, and so we must take care in representing stakeholder views within design.

Our presentation is primarily based on the influential “meta-theoretical” exploration of the structural nature of a theory in the discipline of Information Systems [Gregor 2006], which we briefly recall in Section 2.2. In relation to previous publications, the novel contribution of this paper is to: make explicit the theory implicit in the definition of our engineering framework, with reference to Gregor’s ontological, epistemological and domain questions;
argue, with reference to Gregor’s classification, the key qualities of such a theory, bringing to bear any empirical evidence already accumulated and compare our approach to other emerging theories in SE which offer similar ontological and epistemological views. We will also discuss some ongoing research and yet unpublished related results.

The paper is organised as follows. Section 2 recalls the ongoing debate on the need for SE theories and briefly outlines [Gregor 2006]. 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 some related work. Finally, Section 8 concludes the paper.

2. BACKGROUND

2.1. The ongoing debate

In their opinion piece, Johnson et al. [2012] fall short of proposing any specific form a significant SE theory should have or what the process of generating a theory might be. Other authors have subsequently offered their opinion on these matters. For instance, Adolph and Kruchten [2013] argue that an SE theory “must be useful to practitioners and explain the phenomena they are experiencing” and propose an empirical approach based on grounded theory for theory generation. Along similar lines, Ralph [2015] makes the case 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 to SE. Likewise Adolph and Kruchten [2013], Ralph [2015] argues that such theories should be grounded in SE practice, advocating empirical approaches for their development and comparative studies for their evaluation. A complementary stance is taken by Staples [2015], who proposes that alongside process theories, what SE needs are product theories: as the primary goal of engineering is to change the world rather than merely understanding it as is the case of pure sciences, then engineering theories should be about the performance of artefacts, and their main quality should be their consistency with actual behaviour of software-based systems. Such product theories may not be exact, but only provide some conservative approximations to support assurances about the use of artefacts, that is they need only be general and precise enough to reason about whether an artefact meets acceptable requirements for use. A different angle yet is taken by Smolander and Paivarinta [2013], who argue for the need of theories of practice focused on design and development practices, using a reflection-in-action approach to theory generation.

While Johnson et al. [2012] argue in favour of “significant” or “general” theories for SE, Wieringa [2014] argues that this could be interpreted as the need for universality, which in his opinion would lead to theoretical idealisations unusable in practice. Instead, he argues for “middle-range” theories, which are partial in scope and are usable by practitioners. In particular, also based on previous work [Wieringa 2009; Wieringa et al. 2011], he argues for the need of middle-range design theories, intended as “scientific theories about the interaction of artifacts with their context.” We will return on Wiringa et al.’s work in Section 7.

The case for middle-range theories is also made by Stol and Fitzgerald [2013; 2015], as stepping stones towards a more general and inclusive theory, something that in their opinion is beyond what currently achievable given the relative immaturity of SE. They also discuss the complementary role of theory and empiricism in SE, citing seminal work in behavioural science [Kaplan 1973], and how both (empirical)research-then-theory and theory-then-(empirical)research are viable strategies for theory development, within a general research cycle following a pattern of alternate activities of theorising and gathering empirical evidence.

Beside the type or form of SE theories, the community is also trying to come to terms with what is meant by “theory.” There is some agreement on what is not considered a theory (e.g., [Sutton and Staw 1995]) and much current thinking aligns closely with [Gregor 2006].
A design theory for software engineering which we consider in the next section in some detail as it also constitutes the frame of reference we adopt in this paper.

Finally work has also considered how existing theories might contribute to a general SE theory, including Ralph [2013], who explores five existing theories, including complexity theory and a theory of cognition, and argues how they could be used to analyse SE phenomena, and Johnson and Ekstedt [2015] who also explore the utility of a theory of cognition as part of a theoretical foundation useful to SE.

It is probably fair to say that much work at the moment is speculative and we are still long way away from an agreement on many of the issues we have mentioned.

2.2. Gregor’s meta-theoretical model

Gregor [2006] offers an exploration of theory in the context of the Information Systems (IS) discipline. Given the strong relation between the two disciplines, many of the arguments extend naturally to SE, which explains why this work has been particularly influential within the nascent SE theory community.

Gregor’s exploration of theories is in reference to the following three distinct categories of questions: ontological questions, which pertain to the elements of the theory, how the theory is expressed, and the form of knowledge contribution the theory makes; epistemological questions, which pertain to the way the theory is constructed, the research method and the way knowledge is accumulated; and domain questions, which pertain to the core problems, phenomena of interest and boundary of the discipline. The epistemological dimension is what makes a theory practically useful: it allows knowledge to be accumulated in a systematic manner, guiding research towards crucial questions and enlightening professional practice, something concisely expressed by Lewin [1945] as “nothing is so practical as a good theory.”

When it comes to domain questions, Gregor’s position is that a relevant theory for IS should be able to link the natural, social world and artificial world of human constructions, drawing upon a vast body of knowledge from natural, social and design sciences, and that a wide, rather than narrow view of theory should be taken. It is difficult to argue against this statement for SE also. Indeed, the discussion in the previous section has demonstrated that authors are already considering a wide range of theories from those domains as contributing to a theoretical foundation for SE. Also, with regard to epistemological questions, Gregor’s position is one of inclusiveness, with no requirement to commit to one specific epistemological view: this too is mirrored in the current SE debate as discussed in the previous section, where various epistemological strategies are recognised as equally valuable within the research cycle. Ontological questions concerns the structural nature of a theory and allow one to characterise and compare different theories: indeed they are the basis for the classification of IS theories provided in [Gregor 2006].

With respect to the core qualities of a theory, Gregor’s exploration of the literature leads her to conclude that some combination of the following is essential:

—description and analysis, which refers to an ability to describe phenomena of interest and related constructs, to generalise them within an identified scope, and to analyse their relationships;
—explanation, which refers to the ability of a theory to comment on “how, why, and when things happened, relying on varying views of causality and methods for argumentation” [Gregor 2006];
—prediction, which 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; and

1A fourth category, socio-political questions, concerns the socio-political context in which a theory is developed, something we do not consider in this paper.
— prescription, which allows 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 that it establishes a relationship between theory and method: methods can follow theories, but they are not the same thing [Ralph 2015] and each can exist independently of one another.

In the remainder of this paper we will often refer to Gregor’s work in presenting our theory.

3. RESEARCH CYCLE AND THEORY CONCEPTUALISATION

The design theory defined in this paper is based on a body of work accumulated over a decade of research into a design-theoretic approach to problem solving in the context of software and systems engineering. Given the long-lasting nature of the research, the research cycle adopted has included both theory-then-(empirical-)research and (empirical-)research-then-theory strategies over the years, starting from an initial theoretical proposition [Hall and Rapanotti 2005]. In the following sections, we will present our theory in detail alongside a discussion of any empirical evidence already accumulated in its support, and of aspects for which both theoretical and empirical work is still required. Firstly, we briefly recall influential work which has shaped our thinking and the ontological view of our theory.

3.1. Theoretical provenance

Our initial conceptualisation of the theory was inspired by works in SE and beyond, which we recall briefly in this section.

G.F.C Rogers defined engineering as Rogers [1983a]:

the practice of organizing the design and construction of any artifice\(^2\) which transforms the physical world around us to meet some recognized need

Ostensibly, this definition can be specialised to Software Engineering, simply by taking the artifice to be software. For embedded software, for instance, the software will transform the physical world through precisely described sensors and actuators, perhaps those inside an F1 car engine controller, to drive the wheels in contact with a wet race track [Lyon et al. 1994].

Unfortunately, this “software-as-solution-centric” perspective is deficient: we cannot \textit{a priori} know the precise combination of solution technologies that will ultimately satisfy the need, even less that software will be the sole solution technology.

Moreover, it is likely that through technology combinations, both formal software and non-formal solution domains will need to be combined. But then, as Turski [1986] observed:

“\textit{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.

Finally, in creating software to meet human needs, SE must align with Design. As such an important part of software engineering will be in addressing ill-structured problems [Simon 1996], in which important understanding of the problem and its stakeholders, including the desinger,

\(^2\)Rogers appear to use “artifice” rather than, for instance, “artefact” to emphasise the possibility of a solution with no physical embodiment, such as a business process or, for that matter, software.
plays a fundamental role. These are the actual determinants of satisfaction, not any formally
determined criteria that might exist.

Three key needs that a theory for SE must meet are:

— it must account for the way software is used to meet recognised needs in their real-world
   context;
— it must account for stakeholders’ notions of satisfaction; and
— it must bridge the formal and non-formal divide.

Therefore, we 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 to
   explore ways in which the solution affects our understanding of the problem;
— makes no assumption of the language in which problems or solutions are expressed, nor
   their needs for formal or non-formal reasoning;
— gives focus to the construction of a stakeholders relevant design rationale which captures
   their notion of satisfaction.

3.2. Ontological view
For the first two theory needs: our theory derives its notion of software problem from Rogers’
view of engineering, i.e., as the organisation of the design and construction of (software)
artefacts that meet a recognised need. That the need is recognised introduces the notion of
stakeholder and so their metrics for satisfaction.

For the third, we develop a constructive theory which asks that each formal property
expressed is associated with a constructed witness, i.e., we cannot use a postulated design
to discharge that a problem has been solved, a solved problem must be accompanied by a
constructed solution. Moreover, like the propositional calculus, below the meta-level we do
not constrain the language by which stakeholders, including the designer, communicate.

Moreover, we follow Gentzen [Kleene 1964] in separating context from content in his proof-
theoretic characterisation of the propositional calculus. Briefly, Gentzen assigns meanings
to the propositional connectives through their role in proofs, rather than as Tarski’s manip-
ulations of truth values [Tarski 1944]. Under Gentzen’s interpretation, the contextualised
notion of truth in the propositional calculus is invariant under proof manipulations. For us
the subject of the theory is design, defined so that the contextualised notion of solution is
invariant under design manipulations, hence design theoretic.

A key question is which SE phenomena are amenable to such an encoding. In [Hall and
Rapanotti 2013] we briefly explore this question by sketching a theoretical formulation to
express meaningful relationships between some of the most fundamental entities of SE: in
that position paper, we reason by analogy to mathematics, where Euler’s famous identity
\[ e^{i\pi} + 1 = 0 \]
brings together in a beautifully simple and powerful expression the most fundamental math-
ematical quantities. While recognising the profoundly different nature of mathematics and
SE, we argue for the virtues of equivalent formulations for SE.

In summary, the intent of our theory is to provide theoretical abstractions for the descrip-
tion, analysis, explanation, and, to some extent, prediction of key SE phenomena. Moreover,
while our theory is not prescriptive per se, methods based on such a theory are also possible.

4. SOFTWARE ENGINEERING AS DESIGN THEORETIC PROBLEM SOLVING
Our SE theory is formal and based on the notion of, initially, software problem.
4.1. Software problems
Suppose a problem owner, G, recognises a need in the real world and wishes that need to be satisfied by the design and construction of a software artefact: G has identified a problem P that they wish solved. From G’s perspective, then, a problem P is a pair, consisting of a real world context EG and a need NG giving P as:

\((E_G; N_G)\)

Note that we make no assumption that G’s view of the real world is realistic or representable nor that their need is satisfiable so that \(E_G\) and \(N_G\) must be understood only as placeholders having, perhaps, no representational instantiation: G’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 [Kruteľský et al. 1976].

Example 4.1. Jon, an iPad user, wants to be 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:

\(P_{Jon} : (iPad; ‘I need to know when I should take an umbrella’)\)

4.2. Problem Solving
Irrespective of sense or solution existence, we will assume that G’s wish becomes a challenge to a software engineer D to make sense of P and to solve it. In the very simplest case, G and D will be the same person; in general they will be different.

Following Rogers, given what we know about \(E_G\) and \(N_G\), D’s challenge consists of:

— gaining an understanding of the real-world context in which the problem is located, and of G’s identified need, i.e., they must form their own view, \((E, N)\), of the pair \((E_G, N_G)\);
— agreeing with G that \((E, N)\) is representative, a form of validation;
— producing the software S;
— convincing G that S meets the agreed recognised need N in the agreed physical context E to their satisfaction, another form of validation.

Previously, we have characterised \((E, N)\) as a collection of solutions [Hall et al. 2005]. Under this interpretation, we might state D’s task as identifying elements \(s \in (E, D)\) and choosing one of them.

Here we recognise that, even if expressed linearly as bullet points, the challenge facing D is likely to be highly non-linear; for instance, if D follows some agile school of development, then N might be captured and discharged iteratively as S was built.

For this reason, we wish to remove the asymmetry between problem and solution that could be implied by the notation \(s \in (E, D)\) writing, instead, D’s problem as

\(E(S) \parallel_G N\)

by which we mean “Find S which, when installed in E, meets N to the satisfaction of G.”

Example 4.2. Lucia, our developer, works with Jon to understand their need and context. She captures Jon’s “stay dry” need as SDNeed. Lucia acknowledges Jon’s 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 context to [WStat, iPad]. Naming her solution-to-be IRTA, Lucia forms the developer problem

\(DevP_{Lucia} : [WStat, iPad](IRTA) \parallel_Jon SDNeed\)

(in which the complex environment has been delimited).
4.3. The structure of a software problem

Our encoding of a software problem so far makes no demands on the particular form its elements takes. Indeed, like Jackson [2000], we make no demands of the language of expression other that it can refer to phenomena, which form the mechanisms by which elements of a problem interact.

According to Jackson’s model, a phenomenon is:

An element of what we can observe in the world. Phenomena may be individuals or relations. Individuals are entities, events, or values. Relations are roles, states, or truths. ([Jackson 2001, Page 273])

Even if not precisely equivalent, other notions of phenomena, such those used in hybrid systems (for instance, [Henzinger 2000; de Lemos and Hall 1996], for instance, would be equally valid.

4.3.1. Problem elements.

Environment. A software problem’s environment is characterised by the domains located therein, a domain being a set of related phenomena that are usefully treated as a unit for the purpose of problem solving (c.f., [Jackson 2001, Page 270]).

As a structure, an environment Env is a collection of domains [D₁, ..., Dₙ] each a named description, Name : Description, the description being written in terms of its known, or indicative, properties [Jackson 2001]. Domains interact through their sharing of phenomena. Behaviourally, a domain maps a collection of phenomena to a timeline of their occurrences and interactions [Hall and Rapanotti 2003].

Associated with each domain D are three alphabets of phenomena:

— the controlled alphabet: the phenomena visible to, and that can be shared by, other domains, but that are controlled by D. For the WStat domain above, these might include a likelihood of rain phenomenon rl; for the solution IRTA, they include the alert phenomenon. Controlled phenomena are written superscripted:

\[
\text{WStat}^{rl} \text{ and IRTA}^{alert}
\]

— the observed alphabet: the phenomena made visible by other domains, that are shared by, and whose occurrence is observed by, D. In our example, for the IRTA domain these include rain likely phenomenon, rl; for the iPad, the alert phenomenon. Observed phenomena are written subscripted:

\[
\text{IRTAn}, \text{ and iPad}_{alert}
\]

— the unshared alphabet: all phenomena of D that are not controlled or observed with other domains. For the iPad, these include internal state, myriad internal communications and processing phenomena. Actually, what is shared or unshared depends on the problem under consideration so that the decision as to what is shared is contextually defined. Unshared phenomena are, when presented, written parenthesised:

\[
\text{WStat}(\text{barometric}_{pressure})
\]

The barometric_pressure phenomenon is continuous.

Phenomena composition. Phenomena composition consists of bringing two phenomena within hearing distance of each other. This could be their physical colocation or some other mechanism. Composing phenomena allows their event occurrences to effect one another; for instance, should an iPad be able to observe directly the rain likely phenomenon rl, we could write

\[
\text{[WStat}^{rl}, \text{iPad}_{rl}]
\]
to model the resulting relationship between it and the weather station.

**Example 4.3.** Of course, the iPad cannot observe the rl phenomenon directly, that is the role of the solution. Therefore, in our example, considering phenomena, we have domains:

\[
\text{WStat}^{rl} \quad \text{iPad}_{\text{alert}} \quad \text{IRTA}_{\text{rl}}^{\text{alert}}
\]

which, when composed, will allow Lucia’s IRTA solution to connect the rl and alert phenomena together.

As already mentioned, as described, software problem element descriptions can be in any relevant description language in which there is some notion of phenomena. Moreover, different elements can be described in different languages as they are related only through the phenomena they share.

**Need.** A software problem need states how a proposed solution description will eventually be assessed as the solution to that problem. Like a domain, a need is a named description, **Name : Description.** A need description should always be interpreted in the *optative* mood [Jackson 2001], i.e., as expressing a wish. There are two alphabets associated with a **Need:**

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

**Example 4.4.** Lucia’s need references the rl phenomenon and constrains the alert phenomenon:

\[
\text{SDNeed}^{rl}_{\text{alert}}
\]

**Software Solution.** A software solution is simply a domain, \( S = \text{Name : Description} \), of which the description constitutes a description of some computation which might simply be program code.

As a domain, a solution has controlled, observed and unshared phenomena; the union of the controlled and observed sets is the set of *specification phenomena* for the problem. If a need’s **refs** or **cons** refer to phenomena of the solution domain \( S \), they must be specification phenomena.

### 4.4. Judgements

Someway through problem solving we will encounter D’s variously detailed problem elements and might feel ready to ask whether we have done sufficient work to satisfy the stakeholder. To express whether a problem \( P \) has been solved to the satisfaction of its owner, we form a **judgement:**

\[
P \checkmark
\]

by postfixing the solved interrogative, \( \checkmark \). We explore the means by which this judgement can be made in the sequel.

**Example 4.5.** [IFTTT 2015, ifttt.com] is an Internet Software as a Service (Saas, [Turner et al. 2003]) tool for creating simple trigger-action programs, or **recipes** [Ur et al. 2014].

Lucia is a registered user of [IFTTT] and has shared her location with it. As a user, Lucia has access to the If Rain Then Alert recipe, shown on the left of Figure 1. This recipe runs in the background 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 this IRTA : If Rain Then Alert recipe forms a solution to her problem and so forms the judgement:

\[ \text{WStat, iPad} (\text{IRTA}) \vdash \text{Jon SDNeed} \checkmark \]

Lucia demonstrates her recipe to Jon a number of times and Jon is content that it will alert him as hoped and he accepts Lucia’s solution: the above judgement holds true.

4.5. Software problem transformation

Under the proof theoretic semantics, we establish the truth of a proposition by exploring possible proofs (or refutations), a proof being a sequence of truth preserving transformations that move a proposition to known true propositional judgements, axioms, say; a proposition is true if a proof exists.

By analogy, in our design theory we establish that a problem is solved (and create a solution for it) through a sequence of judgement-preserving transformations that the relevant stakeholder agrees preserve solvability, and that move a problem to known solved problems.

Suppose we have problems \[ E_i (S_i) \vdash_G N_i \], \( i = 1, \ldots, n \), \( n \geq 0 \), and \( \text{step rationale} J \), then we will write:

\[
E_1(S_1) \vdash_G N_1 \checkmark \quad \ldots \quad E_n(S_n) \vdash_G N_n \checkmark \quad \langle J \rangle
\]

(1)

to mean that:

\[ E(S) \vdash_G N \checkmark \]

\[ E(S) \vdash_G N \checkmark \quad \langle J \rangle \]

Below the line is the conclusion problem; above the line are the premise problems.

The reader will note that:

— When there are no premise problems remaining to be solved, i.e., \( n = 0 \), the conclusion problem is solved, with \( J \) justifying that the solution is adequate.

— When problems are expressed in terms of the same phenomena, we say that they tangle. So, when \( n > 0 \), conclusion and premise problems tangle and, when \( n > 1 \), the premise problems may also tangle. Since the solution to a problem may constrain phenomena, the solutions to tangled problems generally need to be designed together, or co-designed. (See Hall et al. 2008 for conditions sufficient to ensure a tangled problems are separable.)
This rule can, perhaps, be seen as an analogue of Polya’s *Decompose and Recombining* heuristic [Polya 1957] with decomposition being from conclusion problem to premise problems, recomposition being from premise solutions to conclusion solutions. Tangles in the premise problems raise the risk that naive decomposition and recomposition will not be adequate, i.e., that the resource used in solving the premise problems in isolation in the hope that their solutions will recompose to be a solution to the conclusion problem, will be wasted. The risk of naive decomposition we term *naivety risk*.

### 4.6. Transformation types

Smolander and Paivarinta [2013] suggest that SE proceeds through a set of identifiable development practices. We observe that there are classes of problem transformations, including sensemaking [Weick 1995], that recur during software problem solving (and have similar step rationale forms). To this end, we define a *software problem transformation schema* as named classes of problem transformations, each describing a general way in which the conclusion problem is related to premise problem(s) through *step rationale*. The intention is for transformation schema to provide theoretical idealisations of common SE practices.

#### 4.6.1. Substitutions.

Given a substitution $PS = [A_i/E_i]$ of environment, need, and/or solution, and step rationale $J_{PS}$, we immediately have:

$$P[PS] \sqrt{[\text{SUBSTITUTION}] \langle\langle J_{PS}; \text{solution: } S[PS], \text{environment: } E[PS], \text{need: } N[PS]\rangle\rangle P}$$

i.e., if a substitution satisfies the appropriate stakeholder, then the original problem will satisfy the stakeholder when the substitute is made.

#### 4.6.2. Interpretation Rules.

The interpretation rules specialise substitution to an element; they are a form of sense-making as applied to the environment, need and solution. For appropriate substitutions $ES, NS$ and $SS$ and step rationales $J_{ES}, J_{NS}$, and $J_{SS}$:

$$E[ES](S) \vdash_c N \sqrt{[\text{ENVIRONMENT INTERPRETATION}] \langle\langle J_{ES} \rangle\rangle}$$

$$E(S) \vdash_c N \sqrt{[\text{NEED INTERPRETATION}] \langle\langle J_{NS} \rangle\rangle}$$

$$E(S)[SS] \vdash_c N \sqrt{[\text{SOLUTION INTERPRETATION}] \langle\langle J_{SS} \rangle\rangle}$$

We note that, because these are specialised substitutions, the conclusion problem’s elements are determined via the substitution applied.

The obligation to provide a step rationale that will convince $G$ is the condition that must be discharged for these interpretations to preserve the solvability of the problem with respect to $G$; the rationale should explain why the new environment, need or solution is preferred to the original; the meaning of “preferred” will, in general, be defined by $G$ and the context.

*Environment* and *Need Interpretation* work in the “problem space” to record improved understanding, the result of *problem exploration*. *Solution Interpretation*, on the other hand, works in the “solution space” to record improved solution understanding, the result of *solution exploration*; see Section 4.7 for more details.

Determining substitutions and step rationales during explorations consumes resources, so that exploration steps generates process risk. We return to this in Section 4.8.

#### 4.6.3. Solution Expansion.

Subject to stakeholder agreement, although for each interpretation rule there is a free choice of substitution, we define a distinguished class of *architectural structures*, or *AStructs* (c.f. [Rapanotti et al. 2004]) that are specific to solution interpretations. As suggested by their name, *AStructs* bring a notion of architecture to our theory: in
A design theory for software engineering

A named AStruct are combined a number of known solution components, $C_i$, with software solution components, $S_j$, yet to be found:

$$\text{AStruct}_{\text{Name}}[C_1, \ldots, C_m](S_1, \ldots, S_n)$$

**Solution expansion** expands an AStruct as follows:

$$[E, C_1, \ldots, C_m, S_2, \ldots, S_n](S_1) \models_G N \sqrt{\ldots} [E, C_1, \ldots, C_m, S_1, \ldots, S_{j-1}, S_{j+1}, \ldots, S_n](S_j) \models_G N \sqrt{\ldots} [E, C_1, \ldots, C_m, S_1, \ldots, S_{n-1}](S_n) \models_G N \sqrt{\ldots}$$

but generates no step rationale obligation in doing so: it is a syntactic transformation that is always applicable to an AStruct interpreted solution. It creates $n$ premise problems each expressed in terms of the conclusion’s elements: problem $j$ and $k$ ($j \neq k$) 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.

When $n = 0$ there are no premise problems and so (subject to the existence of an adequate step rationale for the AStruct interpretation) problem solving is complete.

When $n \geq 2$, in sharing all phenomena, the premise problems tangle so that solutions need to be co-designed.

*Example 4.6.* The MVCiPad AStruct encodes the Model/View/Controller ([Krasner and Pope 1988]) in the context of the iPad:

$$\text{MVCiPad}[[\text{iPad}_{\text{GUI}}](\text{Model}, \text{View}, \text{Controller})]$$

This architecture may already be familiar to the reader; it is illustrated in Figure 2. The MVC AStruct includes the “already known” graphical user interface iPad_{GUI}$^3$, and the “co-designed” components Model, View and Controller.

![Fig. 2. MVC architecture underpinning the iPad Graphical User Interface (GUI)](image)

Should Lucia have chosen to build a bespoke solution rather than rely on MVCiPad, she could have used MVCiPad to do so. This would have involved **solution interpretation** by the MVCiPad AStruct above. As the MVCiPad involves three to-be-found components, a subsequent solution expansion leaves three problems:

$$P_M : [WStat, \text{iPad}, \text{iPad}_{\text{GUI}}, \text{Controller}, \text{View}](\text{Model}) \models_{\text{Jon SDNeed}}$$

$^3$Which we don’t detail.
\( PV : [WStat, iPad, iPad\_GUI, Model, Controller](View) \vdash_{\text{Jon}} SDNeed \)

\( PC : [WStat, iPad, iPad\_GUI, Model, View](Controller) \vdash_{\text{Jon}} SDNeed \)

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 \( PV \) and \( PC \) are solved. But \( PV \) and \( PC \) 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 [Kruchten 2000; Larman 2012], 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, together with a protocol for the phenomena that drive its presentation by the View and updating by the Controller.

If \( M_{\text{Design}} \) is the description of the designed model, then we can write:

\( PV : \ [WStat, iPad, iPad\_GUI, Model : M_{\text{design}}, Controller](View) \vdash_{\text{Lucia}} SDNeed \)

\( PC : \ [WStat, iPad, iPad\_GUI, Model : M_{\text{design}}, View](Controller) \vdash_{\text{Lucia}} SDNeed \)

from which we can then address the design of the View and the Controller which are more loosely coupled.

**4.6.4. Problem progression.** Problem progression, first sketched by Jackson [2001], provides a way of deriving specifications from requirements in a systematic fashion [Li 2007]. The idea of problem progression is to remove domains from the environment model whilst altering the requirement model to compensate; as the complexity of the modelled environment is reduced, the detail of the modelled requirements increases until we are left with a specification of the solution [Li 2007].

Li [2007] defines three classes of Problem Progression depending on the nature of the domain, but each has the following form:

\[
E'(S) \vdash_G N' \quad \text{[Problem Progression]}
\]

Like Solution expansion, Problem Progression is a syntactic rewriting of the conclusion judgement, with the relationship between the old need \( N \), the new need \( N' \), \( E' \) and \( D \) as defined by Li’s rewriting rules [Li 2007, Page 121ff].

**Example 4.7.** 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 will 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:

\[
[\text{iPad}_{\text{alert}}[(\text{IRTA}_{rl})_{\text{alert}}]] \vdash_{\text{Jon}} SDNeed_{rl}^{\text{alert}} \quad \text{[Problem Progression]}
\]

where SDNeed’ has description ‘If IRTA receives rl then send alert’.

Following Li [2007], a subsequent applications of problem progression would allow the removal of the iPad domain, leaving a specification for the solution.

**4.6.5. Stakeholder replacement.** One stakeholder that relies on the judgement of another can be replaced. The replacement will often be associated with criteria for acceptance of a proffered solution, such as acceptance testing:

\[
E(S) \vdash_G' N \quad \text{[Replace Stakeholder]}
\]

\[
E(S) \vdash_G N \quad \langle G \text{ accepts G’s judgement, with specified conditions} \rangle
\]
A design theory for software engineering 13

**Example 4.8.** Jon’s father, George would also like to know whether to take an umbrella. George isn’t a sophisticated user and doesn’t have the requisite skills to be able to come up with a solution to the problem herself. However, as George trusts Jon’s judgement, he accepts Lucia’s solution:

\[
\begin{align*}
{}_{\text{WStat}^t, \text{iPad}_{\text{alert}}} & \vdash_{\text{Jon}} \text{SDNeed}_{\text{alert}} \checkmark \\
{}_{\text{WStat}^t, \text{iPad}_{\text{alert}}} & \vdash_{\text{George}} \text{SDNeed}_{\text{alert}} \checkmark
\end{align*}
\]

〈George defers to Jon〉

In the case when George is more discerning, the acceptance of Jon’s judgement might involve other criteria, such as acceptance testing, for instance.

### 4.7. Design trees

Again inspired by the propositional calculus, software problem transformations combine to produce *design trees*. Moreover, as we saw above, any software problem transformation that leaves an empty set of problems above the line, i.e., \( n = 0 \) in Equation 1, completes a branch of the tree. When all branches are complete, the problem is solved: a fit-for-purpose solution is determined together with a complete design rationale to the satisfaction of the stakeholder.

**Example 4.9.** From Lucia’s development steps we can build the following completed design tree:

\[
\begin{align*}
{}_{\text{WStat}, \text{iPad}} & \vdash_{\text{Jon}} \text{SDNeed} \checkmark \\
{}_{\text{WStat}, \text{iPad}} & \vdash_{\text{Jon}} \text{N} \checkmark
\end{align*}
\]

〈Apply Need Interpretation [N/SDNeed], found in consultation with Jon〉

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

The reader will note that the design rationale created in this example is extremely weak, indeed it has no *formal* existence at all, and yet is still fit-for-purpose in this stakeholder context. For other stakeholder contexts, mission- or safety-critical systems for instance, much stronger step rationales would most likely be needed to justify individual transformations leading to another fit-for-purpose overall design rationale.

Not all forward steps in a development are correct. Indeed, even in the case when the stakeholder is convinced by the step rationale, it could be that a substitution, chosen as part of the interpretation rules, is inadequate or incorrect. In this case, we will need to backtrack by unwinding steps taken subsequently, to recover a prior design tree state to continue our exploration from. Thus, the commitment of resources to problem and solution explorations raises a process risk. This is discussed in the next section.

### 4.8. Process considerations

Our theory does not constrain the order of application of problem and solution exploration steps; in particular, there is no necessity for an SE project to arrive at a “perfect” (or even any!) problem description before beginning solution exploration. Rather, problem and solution explorations will likely interleave until the required level of understanding is gained of both problem and solution. Iteration between problem and solution domains is typical even in the earliest SE process models (for instance, [Royce 1970]). Larman and Basili [2003] seek to avoid a “single-pass sequential, document-driven, gated-step approach,” and iteration is the basis of modern agile methods [Agility 2004], for which the incremental advancement of need (‘requirements’ and software solution is key.
Both problem and solution exploration are resource intensive – problem explorers will, for instance, expend resources exploring the problem, perhaps with stakeholders, to draft software requirements, capture domain descriptions, etc in the hope of understanding the problem; solution explorers will, in contrast, spend their time with end-users, for instance, architecting and prototyping, collaborating in scrums, writing test harnesses and documentation, etc in the hope of finding the solution. Neither phase is guaranteed to be successful.

Both problem and solution exploration, therefore, commit resources to uncertain outcomes which thus constitutes developmental process risk:

— in the solution space: that a proffered solution is rejected by a stakeholder — it fails agreed acceptance tests, for instance — leading to rework;
— in the problem space: that understanding of a problem is not adequate leading to solution exploration with little chance of success, incurring loss of resource for both.

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 ([Hall and Rapanotti 2009a; Kaminsky and Hall 2015]).

The possibilities for interaction of exploration and validation steps is characterised in the problem solving process pattern shown in Figure 3. 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 development practices it serves more as a problem solving process generator.

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 systematic solution, and 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 through co-design, and the use of validation as the means to mitigate developmental risks.

We now observe that, throughout the development of our theory, we have placed little, if any, reliance on the nature of the solution domain as software. Given that we began with
A design theory for software engineering

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 special theory in the context of Rogers’ full definition.

There appears very good motivation for such an endeavour:

— Firstly, as Rogers [1983b] writes:
  “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, [Hall et al. 2007]. 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 [1970], 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 [Rajlich 2006], i.e., contexts in which software requirements change rapidly. 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, [Boehm et al. 2014].

— 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.

Of course, the consideration of multiple solution technologies in a theory is not new; there is already much extant work in the modelling of hybrid systems, in which discrete and continuous modes of behaviour combine to describe a system. Example of hybrid modelling theories include hybrid logics [Chaochen et al. 1993; Hall and de Lemos 1996], hybrid automata [Henzinger 2000], hybrid Petri nets [David and Alla 2005] amongst others. Hybrid systems work generally focuses on the modelling and analysis of, what we would call, the solution domain; our step-wise design-theoretic treatment of both problem and solution is novel.

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

\[ E(S) \models_G N \]

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 the implications for engineering problems that may be of interest to the software engineer.
5.1. Requirements engineering

The Reference Model of Gunter et al. [2000] 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. According to Gunter et al., requirements engineering relates five artefacts: domain knowledge (environment), requirements, a specification, a program, and a programming platform (system or machine). 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].’

((Gunter et al. 2000, Page 38))

These elements are arranged as shown in Figure 4.

![Diagram of the two ellipse model](image)

Here we interpret the Gunter model in our problem model. The problem that requirements engineering solves is to produce a specification from environment and need descriptions; the problem that software development solves is to produce a software artefact that, when embedded in a machine, satisfies the specification. That is, given problem owner \( G \) and developer \( D \):

\[ P_{\text{Req}} : \text{Env}(\text{Spec}) \vdash_G \text{Req} \quad \text{and} \quad P_{\text{Dev}} : \text{Machine}(\text{Code}) \vdash_D \text{Spec} \]

We note:

— these problems tangle as they are both expressed using specification phenomena;
— 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 that include very high-energy radiation, electromagnetic interference, battle damage, and environmental extremes ([Johnson 1984]). In essence, an SEU induces unspecified behaviour modes in Machine which can cause the Code to malfunction. If Env is an aggressive environment (i.e., and environment that ‘controls’ SEUs) then we cannot avoid Machine\(_i\) in which, in the worst case, chaos is the outcome of an observed \( i \) event. Thus, from a developmental perspective, a solution to

\[ \text{Machine}(\text{Code}) \vdash_D \text{Spec} \]

is not necessarily a solution to

\[ \text{Machine}_i(\text{Code}) \vdash_D \text{Spec} \]
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. Spec is not a software artefact and so not representable as the solution to a (pure) software problem. So, to be able to partition the engineering process this way, then, we must be able to work in the solution space of specifications. This does not prevent a similar development within the special theory: we can write:

\[
\begin{align*}
\text{Machine}(\text{Code}) & \vdash \circ \text{Spec} \quad \text{(Replace Stakeholder; acceptance testing by } G) \\
\text{Machine}(\text{Code}) & \vdash \circ \text{Spec} \quad \text{(Problem Progression)} \\
\text{\ldots} & \\
\text{Env} \text{, Machine}(\text{Code}) & \vdash G \text{ Req} \quad \text{(Problem Progression)} \\
\text{Env}(2\text{ellipse}\text{Machine})(\text{Code}) & \vdash G \text{ Req} \quad \text{(Solution expansion)} \\
\text{Env}(S) & \vdash G \text{ Req} \quad \text{(Solution interpretation inspired by the two ellipse model)}
\end{align*}
\]

That special and homogeneous developments are equivalent is predicated on the assumption that the sequence of Problem progressions is possible. It is an open problem as to whether this is always the case. Detailed examination of Li’s transformation [Li 2007] would be necessary to confirm this. If it is the case, then we have a form of Cut-elimination ([Kleene 1964, Page 443]). In Gentzen’s system, Cut elimination is an important and hard fought for property which can be used to demonstrate that, amongst other things, the system is internally consistent. We might, therefore, find it desirable that its equivalent would holds between the special and homogenous theories. This is an avenue for future investigation.

5.2. Socio-technical problems

As an extension to the above Reference Model for Requirements, Hall and Rapanotti [2009b] introduce the three ellipse model for socio-technical systems in which social and technological ellipses are added as potential solution ‘technologies’. An adapted three ellipse model is shown in Figure 5 (cf., Figure 4).
As in the two ellipse model, we begin with $\text{Env}(S) \vdash G \text{ Req}$, this time interpreting the solution with $3\text{ellipse} \{\text{Soc}, \text{Tech} \} \{\text{Proto, Sys} \}$, and then repeatedly applying Problem Progression until only the target — the social and technical subsystems — remain:

$$
\begin{align*}
\text{Soc(Proto)} & \vdash G \text{ Spec}_{\text{ES}} \checkmark \\
\text{Tech(Sys)} & \vdash G \text{ Spec}_{\text{ET}} \checkmark \\
\text{Env, Soc, Tech, Sys}(\text{Proto}) & \vdash G \text{ Req} \checkmark \\
\text{Env, Soc, Tech, Proto}(\text{Sys}) & \vdash G \text{ Req} \checkmark \\
\text{Env}(\text{Soc, Tech}(\text{Proto, Sys})) & \vdash G \text{ Req} \checkmark
\end{align*}
$$

and by which the two environment specifications $\text{Spec}_{\text{ES}}$ and $\text{Spec}_{\text{ET}}$ are characterised. The resulting problems will be expressed in term of phenomena from $\text{Spec}_{\text{ST}}$ and so tangle. The order in which these problems should be attacked is therefore an issue.

The interplay between human and information system is an area well explored in the IS literature, including Fitt’s early MABA-MABA lists [Fitts 1962], Price’s Decision Matrices [Price 1985] and, more recently, Dearden’s scenario method [Dearden et al. 2000], so that thought may first be given to the social subsystem which will constrain the $\text{Spec}_{\text{ST}}$ specification to bias the social aspects of the interface. However, more modern trends see the development of the commercial off-the-shelf technical sub-system. If the organisation requires a enterprise resource planning (ERP) system, for example, then SAP ERP might fit the bill. Such systems require the social subsystem to flex to the demands of the technical system which is determined first.

5.3. Validation problems

Validation problems [Hall and Rapanotti 2009a] are a class of problems that arise during software engineering (indeed, they arise during all forms of engineering) but do not have software as a solution component. The general form of a validation problem is:

$$
\text{Validation}_{\text{Env}}(\text{Validation}_{\text{Artefact}}) \vdash \text{Validator}_{\text{Validation}_{\text{Need}}}
$$

i.e., the need is for a validation artefact which, when considered in the validation environment, satisfies the needs of the validator. Validation problems arise from the relationship between, for instance, a validating stakeholder — the problem owner, for instance — and a developer. A validation environment is the environment in which the validation artefact ‘works’ to satisfy the validator.

Example 5.1. A safety case, as required by a safety regulator [Bloomfield et al. 1998], is an example of a validation artefact, to be validated by the regulator:

$$
\text{Regulatory}_{\text{Environment}}(\text{Safety}_{\text{Case}}) \vdash \text{Regulator}_{\text{Regulatory}_{\text{Requirements}}}
$$

The reader will note that this validation problem exists 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 the regulator. The criteria for satisfaction of the regulator is captured in the Regulatory Requirements which will include any required form that the safety case must have. The Regulatory Environment consists of the relevant regulations and standards, but may have other elements, such as an executive summary, house-style, associated presentation, etc.

---

4The rather dated ‘Men Are Better At-Machines Are Better At’
5https://en.wikipedia.org/wiki/SAP_ERP
As is suggested by the example, there will be one validation problem to solve for each validator. In the general case, 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.

The customer’s and validators’ problems provide another instance of tangled problems. Here, rather than being generated as part of the expansion of an AStruct, they arise through the interactions of problem solving processes.

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-off between validated and unvalidated problem solving to be considered in any development.

5.3.1. Trusted and Fractal Problem Solving. Depending on the developmental context, of course, it may be that developmental activities occur without accumulating risk at all when, for instance, failure has no cost associated with it; development of an recipe is, to some extent, a throw away activity needing no serious validation. Even in an organisational setting, an end-user created spreadsheet may not need validation if nothing mission-critical rests on it. These are examples of trusted problem solving activities. It is natural to consider that they should be avoided in, say, a safety-critical context, with the expectation that all activities should be validated. We note, however, that trusted processes need not uncover unreasonable levels of risk: in the safety-critical context, for instance, whereas the lack of validation of an assurance case will critically damage the development process’s ability to deliver, the lack of validation in the identification of a solution validation problem environment and need need not incur untenable levels of risk.

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 composition 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.

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

\[
P_1 \quad \ldots \quad P_n \quad \langle\langle J \rangle\rangle \equiv [P](\text{ProbXform}[[P_1, \ldots, P_n, J]]) \vdash_D \text{Each } P_i \text{ is solvable} \land J \text{ will satisfy } G
\]

in which ProbXform is the AStruct for ‘problem transformation’ and which should be read as, given conclusion problem \( P \), find i) premise problems and ii) step rationale that, in \( D \)'s estimation will i) be solvable and ii) are reached from \( P \) using \( J \) that will satisfy \( G \) that the step is correct.

We note that the ProbXform AStruct expands, as expected, under solution expansion to leave \( n + 1 \) premise problems: \( n + 1 \) 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, solving the ‘\( J \)’ problem first, we have previously called Assurance-driven development [Hall and Rapanotti 2009a].

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 effect, 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 difficulty with the termination of the problem solving process: when does delegated problem solving stop? In practice, the delegate 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.

Fig. 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.

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 section, 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 the step-wise problem solving which takes place when software solutions are engineered, including the explication of key relationships between artefacts, such as requirements, context assumptions and architectural design, between actors in the process, specifically the various stakeholders who take part as explorers or validators, and the classification of recurrent steps as idealised problem transformation classes.

The empirical evaluation of the theory is an ongoing process. An initial comprehensive proof-of-concept application can be found in [Hall et al. 2008], where the logical encoding was exercised on a complex software problem from the literature: the aim of that evaluation was primarily to test the extent the encoding was sufficiently expressive to capture common SE practices. More substantial empirical evidence from practice was subsequently accumulated.
through the repeated application of the theory in the context of a number of industrial case-studies and indeed it carries on within ongoing industrial studies. It is worth noticing that it was its real-world application which motivated the move from the special software theory towards the more general homogeneous theory: even in small scale industrial studies, it is apparent that software is most often only one of many technologies and social elements which participate in the solution to real-world problems, and whose interplay needs addressing.

A three-year research collaboration with General Dynamics UK included a series of case studies [Mannering et al. 2007a; 2007b; 2007c; Hall et al. 2007; Mannering et al. 2008; Mannering 2009] in the context of safety-critical avionics application development, where the interplay between bespoke software and hardware components is of particular relevance. 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 fined 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 4.7 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 tool [Mannering et al. 2007b], contributing evidence for early life-cycle assurance (captured in our theory via validation problems), something considered problematic across the industry. The relation between validation and design was one of particular relevance in these case studies, where high assurance is needed for product certification by an independent authority, a pre-requisite for a system to become operational. Although not discussed in this paper, assurance-driven design, where assurance guides the design rather than being retrofitted to it, is accounted for in our process pattern, as discussed in [Hall and Rapanotti 2009a].

In [Nkwocha et al. 2013] the theory was tested on a software problem in the context of a financial institution. In this case the problem concerned the identification and elimination of the source of recurrent incorrect mortgage calculations made by the 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 through our theory, and a notable result was that the step rationale which accompanies problem solving steps was the vehicle for expressing the design rationale for solution architecture alternatives, something that could be used as the basis for validation and decision making by the customer, who was paying for the software to be fixed.

The prominent role of validation in problem solving was also noted in [Rapanotti and Hall 2009], which tested the theory on the live development of a socio-technical system. Of particular note in that study was an extension of the previous observation on how assurance influences design to 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. An interesting result from the study was the observation that step-wise problem solving also applies to validation problems, which reinforces the idea of treating engineering and validation problems uniformly. Also the study put to test the ability of the theory to combine multi-description languages, due to the need to describe and analyse both the behaviour of technical and social components within their environment: this provides some evidence that the logical framework works well as a meta-layer above that of specific domain descriptions, adding to the evidence from the previous studies where formal description languages were used.

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” [Gregor 2006].
In some of the retrospective safety-critical avionics case studies mentioned earlier, a safety analysis performed on early problem models was able to predict safety defects that, according to the company logs, in the live projects had only emerged in much later design phases, requiring more costly intervention and re-design [Mannerering 2009]. This led to the addition of extra exploration and validation steps to the process for application in subsequent projects. Similarly, as part of the study in [Nkwocha et al. 2013], the process pattern (see Section 4.8) 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 [Hall and Rapanotti 2015], which provides a theoretical characterisation and comparison of well-known software development process models from the literature. Although a thought experiment, it does highlight the explanatory capability of the process pattern, based on its view of development risk and resource expenditure, and provides examples of how the pattern may be used to analyse development processes in practice.

The phenomenological basis of our problem description is particularly suited to the development of arguments as to the expected behaviour of the designed solution in its context, which goes somewhat towards satisfying Staple’s [2015] desire for product theories. In fact, there is a sense in which our definition of engineering problem, coupled with the step-wise logical construction of the accompanying rationale (see Section 4.5), acts as a generator for practice-specific [Wieringa et al. 2011] 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.

As we share such a phenomenological basis of problem descriptions with the work on Problem Frames [Jackson 2001], further evidence can be found in the related literature. For instance, a demonstration of the powerful relation between those descriptions, causality and argumentation based therein is given in [Li et al. 2012], where causal chains among phenomena are exploited through PROBLEM PROGRESSION to transform systematically requirements expressed in terms of real-world phenomena into corresponding software specification.

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 4.6) can be seen as a general characterisation of problem decomposition, able to capture many types of decomposition found in practice, from decomposition based on design patterns [Hall et al. 2008; Overton et al. 2009] and architectural styles [Rapanotti et al. 2004] to that based on viewpoints, like Jackson’s problem projection [Jackson 2001].

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” [Gregor 2006].
We have already commented on how problem descriptions generated through application of the theory in practices can be seen as product theory, hence able to provide approximate predication 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 (2015) encodes our process pattern as a discrete-time Markov chain model [Tijms 2003] in which the various probabilities correspond to problem solving team experience: a more experienced team will, for instance, understand a problem with higher probability than 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 logical encoding which is of relevance to prediction. By analogy to proof-theory, where tactic languages [Martin et al. 1996] are used to generate proof semi-automatically, we have developed a design tactic, or dactic, language [Hall and Rapanotti 2012], by which designs can be captured for replayed in different contexts. Because of their angelic nature [Martin et al. 1996], 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 areas, 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 [Eve 2011; Mwangi 2013].

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 [Hall et al. 2007] for safety-critical engineering, subsequently adopted and extended in other industrial work [O’Halloran 2012], 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 [Rapanotti and Hall 2009], 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 A\textsc{ structs} 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 4.5, co-design problems are examples of tangled problems, that is problems whose relationships are such that their naive decomposition into separate sub-problems is unlikely to lead to an overall solution. Indeed, many real-world situations can be characterised as tangled problems: for instance, [Hall 2011] 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 [Hall and Rapanotti 2008; Hall 2011; Markov et al. 2015] 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. Another important topic not considered in our theory so far is change. 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 assumes a type of “greenfield” engineering, in which something new is created to affect the environment so that the need is satisfied. As such there is no notion of “change” as a first class element of the theory: for instance, if meeting a new organisational need could be obtained by changing an existing information system, then the starting point would not be a blank canvas, but an existing software system which may be tweaked to meet the need. As our theory stands, the process of analysis of a current situation where this type of change may be needed is not accounted for, which is a clear limitation. Therefore, current research is introducing a notion of change problem into our theory, [Markov et al. 2015] exploring its relationship with the definition of engineering problem above, and the implications on process pattern and transformation classes.

6.5.3. 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 tracks 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.

6.5.4. Kolmogorov’s method-theoretic approach. As an interpretation of intuitionistic logic, Kolmogorov [Mancosu 1998, Page 328–334] 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, $a \Rightarrow a \land a$ is a mapping from the method of solution of $a$ to the method of solution of $a \land a$ which, in turn, is the method of solution of $a$. 
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:

$$E_1(S_1) \models G_N \sqrt{E_2(S_2) \models G_N} \sqrt{\wedge - \text{Introduction}}$$

For negation, the situation is less clear. For, although we can say $$\neg E(S) \models G_N$$ holds when the design tree for $$E(S) \models G_N$$ ends in a known unsolvable problem, we need the result that no problem can be both solved and reduced to known unsolvable problems. Indeed, a stakeholder that always validates will cause problems in this regard. More work is required here.

We cannot expect, however, that our theory is contained in Kolmogorov’s. In his critique of existential propositions [Mancosu 1998, Page 333], Kolmogorov identifies two elements of a 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 are subjective as their expression depends on the knowledge of the developer, i.e., subjectivity extends throughout the whole problem. See Figure 7 for an illustration.

During problem solving, then, we will encounter various degrees of objectivity and subjectivity within a problem statement. The fractal nature of problem solving is integral to the resolution of subjectivity: more objective problem or solution understanding can be the outcome of a fractal problem solving invocation.

As a final comment, the substitution rule allows a problem solver to explore their beliefs of a problem’s elements. This becomes (stakeholder-)relative knowledge after their validation has been given, based on the justification that step and design rationale provide. Thus, knowledge in our theory is stakeholder validated justified belief.

7. RELATED WORK

Wieringa et al. [2011] 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 may 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). Different from our process pattern, Wiringa 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 [Wieringa 2009], rather than a process analyser and generator as is the case of our pattern, with the acknowledgment that not all specified tasks will be present in all engineering processes in practice. Also, as there is no explicit consideration of risk and resources, it may be argued that the cycle has a purely descriptive function. Within the cycle, Wieringa [2009] 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 observations 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 context, and that it satisfies other criteria specified by stakeholders [Wieringa 2009]. Indeed, both elements are also present in our theory: problem models refers to the mechanisms by which solution and context phenomena interact to satisfy a recognised need, with further validation criteria elicited and checked though validation activities.

Wieringa et al. [2011] introduce a notion of problem theory as a theory for a particular problem context 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 stakeholders’ 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. [2011] 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.

A form of design theory which Wieringa et al. [2011] 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 context. 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 Wieringa et al.’s [2011] consideration of different levels of generalisation and idealisation a theory may provide. 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, [Wieringa et al. 2011] 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, [Wieringa et al. 2011] 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.

Expanding on over four decades of research in SE development processes and practices, [Boehm et al. 2014] 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. Different from our process pattern, which is a 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 afforded 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 independently 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.

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.

REFERENCES


A design theory for software engineering


