A General Theory of Engineering: Thinking Bigger than Software

Jon G. Hall and Lucia Rapanotti

28 January, 2015

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 General Theory of Engineering: Thinking Bigger than Software

Jon G. Hall\textsuperscript{a,*}, Lucia Rapanotti\textsuperscript{a}

\textsuperscript{a}Dept. of Computing and Communications, Walton Hall, Milton Keynes, MK7 6AA, UK

Abstract

Context:. Software engineering 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.

Objective:. A cogent theory of software engineering improves the maturity of our discipline, placing it alongside other engineering disciplines whose theories are apparent. Moreover, given that software engineering produces some of the most complex and versatile objects that have ever been designed, we might also like to reflect whether what has been learned in software engineering might not be usefully reflected in other engineering areas.

Method:. The theory was developed through empirical observation of practice together with philosophical argument from which a principled basis was developed. The paper brings together and explicates over 10 years of research in this area.

Results:. We describe two theories of software engineering. The first special theory brings together phenomena of specific interest to software engineering, systematising concepts and practices in a way that attempts to capture how software engineers go about addressing real-world problems. The second general theory embeds software engineering in a general theory of engineering, in passing showing that the phenomenological and process bases of software engineering usefully extends to the more general setting.

Conclusions:. The theories we have proposed capture and generalise relationships between some of the most important elements that are found in Software Engineering, by introducing a systematisation of concepts and
practices. In addition, we have demonstrated their analytic and explanatory features. The general theory is predictive in suggesting interventions in engineering processes that reduce process risk, both for software and when software is a component of a multi-technology system.

*Keywords:* Problem orientation; Software Engineering; Design theory; Engineering Theory

1. Introduction

The paper discusses the conceptual foundation of Problem Oriented Engineering (POE) [1], our proposed theory of engineering, and its development from and embedding of our preceding theory of Software Engineering, Problem Oriented Software Engineering (POSE) [2].

Our proposal is design theoretic on two accounts. On the one hand, it sees Software Engineering as a socially situated design problem solving process [3, 4], within which creativity and expertise come together to solve real-world problems so by satisfying a wide range of stakeholders. On the other hand, it captures such design problem solving processes within a logical framework akin to proof-theoretic frameworks [5], providing a logical foundation to represent and reason about phenomenological relationships and their unfolding in the process. Such process unfolding means that POE also exhibits characteristics of a process theory [6].

Compared to previous POSE and POE publications, which have focused primarily on POSE’s and POE’s technical definitions and applications to practice and evaluation, this paper digs deep into the philosophy of POSE and POE and their position in relationship to (software) engineering theories.

The paper has two parts. In Part I, including Sections 2 to 6, we lay the foundation of POSE and argue is virtues and limitations as an heterogeneous Software Engineering theory. In Part II, from Section 7 onwards, we address the generalisation to an homogeneous theory in POE and position it in its wider academic context. Section 11 draws some conclusions and implications for future research.

*Corresponding author

*Email addresses:* jon.hall@open.ac.uk (Jon G. Hall), lucia.rapanotti@open.ac.uk (Lucia Rapanotti)

Preprint submitted to Elsevier December 16, 2014
Part I

A Special Theory of Software Engineering

2. From Engineering to Software Engineering

G.F.C Rogers defined engineering\(^1\) as [7]:

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

This is easily specialised to give a definition of Software Engineering as:

the practice of organizing the design and construction of any software artifice which transforms the physical world around us to meet some recognized need

For embedded software, the software will transform the physical world through sensors and actuators, perhaps inside an F1 car engine controller driving the wheels in contact with a wet race track [8], or the experimental machinery in a probe on a far distant comet about which little is known at the outset [9]. More mundanely, within the home or business, the software will function within a software environment such as an operating system or a database system within a PC, smart phone or tablet, which, in turn, will interact with its operator, whose actions will be influenced by it to affect the organisation at a level as deep, perhaps, as its culture, [10].

That Software Engineering transforms the physical world to meet a recognised need means that a theory of Software Engineering should enable reasoning of how ‘software causes’ can be combined to create desired effects within these and other distant real world domains.

This is problematic for, as Turski [11]) observes:

‘there are two fundamental difficulties involved in dealing with non-formal domains (also known as ‘the real world’):

\(^1\)Technically, this definition considers only a green-field setting, i.e., \textit{ab initio} development; we comment on its extension to \textit{change problems} later in the paper.
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.

Thus a theory of Software Engineering must bridge between the formal and the non-formal.

One candidate for bridging formal and non-formal worlds is the propositional calculus (for instance, [12]). A proposition can be described in essentially any language, formal or non-formal, and thus can be related to any domain: ‘It is raining’ is as much of a proposition as ‘x=0’. The secret of the propositional calculus is the contextualised mapping of propositions to truth values (true/false): within a particular context — location, interlocutor, *etc* — ‘It is raining’ can be evaluated as true or false.

To define our theory of Software Engineering, then, we will encode Rogers’ notion of Software Engineering within the propositional calculus. To do so, we characterise a Software Engineering problem as a proposition thus: let Env be the physical world, Need be the recognised need of G, the problem holder, and Soln the software artifice (each of which is expressed in an appropriate language — more on this later), then a software problem is the proposition:

$$P : \text{Env(Soln)} \text{ meets}_G \text{ Need}$$

the truth value of which indicates that, when Soln is installed in the environment Env, their combination meets G’s need Need²

When Env is complex, we will sometimes enclose it in angle brackets — ⟨Env⟩ — more clearly to delimit its scope from the other elements of a problem.

**Example:** Michael wants a piece of software for his smart phone so that when the weather forecast predicts rain in his area, he is alerted to take an umbrella.

Michael’s physical world includes a smart phone, the weather, weather station, and an umbrella – the weather station monitors the weather through sensors and issues weather forecasts that are available over the Internet

---

²In software, of course, a fully expressed Need is often referred to as a requirement. We use the terms interchangeably.
which is available on Michael’s phone. Michael has recognised his own need as that of

\[
\text{Need} = \text{‘when the weather forecast predicts rain, I am alerted to take an umbrella’}.
\]

We can represent Michael’s problem as:

\[
P_{\text{Michael}} : \langle \text{Smart phone, Weather, Weather station, Umbrella} \rangle \text{(Soln)} \quad \text{meets}_{\text{Michael}} \quad \text{Need}
\]

With Soln the solution to be found. Michael decides that the platform of choice for the Soln is IFTTT. IFTTT stands for If This Then That is an internet software as a service tool for creating recipes [13] — simple trigger-action programs [14]. Michael’s Soln is the recipe shown in Figure 1.

![Recipe](image)

Figure 1: Left: the IFTTT recipe to remind Michael to take an umbrella if tomorrow’s forecast is for rain; and, right, a received SMS.

The propositional calculus manipulates propositions through meta-level operations on truth values – conjunction, disjunction, implication, etc: we might write ‘It is raining ⇒ x=0’. Reasoning within a domain – formal or non-formal – takes place below this meta-level.

**Example:** For problems, if both Michael and Gordon are stakeholders in the same software problem \( P = \text{Env(Soln)} \) meets \( \text{Need} \), but with Michael more critical than Gordon, then we can say

\[
P_{\text{Michael}} \Rightarrow P_{\text{Gordon}}
\]

to capture the fact that if Michael says a problem is solved, then Gordon will agree. Thus Gordon will be able to use Michael’s IFTTT recipe, too. ■
In the propositional calculus, we establish truth values of propositions by exploring possible proofs (or refutations). In our design theory we establish that a problem is solvable by exploring the design space for possible designs. A proof is a sequence of truth preserving transformations that move a conjecture to known true propositions; a design is a sequence of solvability preserving transformations that move a problem to known solved problems if and only if the original has a solution. In this way, we can arrange that a software problem is solved with respect to the stake-holder iff its proposition is true. From our representation of problems as propositions we may thus construct a system in which Software Engineering can take place.

Software problem transformations capture discrete steps in the software problem solving process and relate a software problem and a step rationale to a (set of) software problems and conform to the following general pattern. Suppose we have problems $\text{Env}(\text{Soln}) \text{ meets}_G \text{ Need}$, $\text{Env}_i(\text{Soln}_i) \text{ meets}_G \text{ Need}_i$, $i = 1, ..., n$, $(n \geq 0)$ and step rationale $J$, then we will write:

$$
\text{Env}_1(\text{Soln}_1) \text{ meets}_G \text{ Need}_1 \quad ... \quad \text{Env}_n(\text{Soln}_n) \text{ meets}_G \text{ Need}_n
$$

$(J)$

\[ \text{Env}(\text{Soln}) \text{ meets}_G \text{ Need} \]

to mean that:

$\text{Soln}$ is a solution of $\text{Env}(\text{Soln}) \text{ meets}_G \text{ Need}$ with respect to stake-holder $G$ and with design rationale $(\text{AA}_1 \land ... \land \text{AA}_n) \land J$ whenever $\text{Soln}_1, ..., \text{Soln}_n$ are software solutions of $\text{Env}_1(\text{Soln}_1) \text{ meets}_G \text{ Need}_1, ..., \text{Env}_n(\text{Soln}_n) \text{ meets}_G \text{ Need}_n$, with design rationale $\text{AA}_1, ..., \text{AA}_n$, respectively.

below the line is the conclusion problem; above the line are the premise problems. By this definition it follows that the design rationale is the conjunction of the step rationales.

**Example:** Returning to Michael and Gordon, we can take the material implication $P_{\text{Michael}} \Rightarrow P_{\text{Gordon}}$ and make a software problem transformation step,

$$
\frac{P_{\text{Michael}}}{P_{\text{Gordon}}}
$$

Michael stricter than Gordon

to mean that

‘Once $P_{\text{Michael}}$ is solved with design rationale $\text{AA}$, then $P_{\text{Gordon}}$ is solved with design rationale $(\text{AA} \land \text{‘Michael stricter than Gordon’})$.‘
Software Engineering design under our theory proceeds in a step-wise manner: the initial software problem (Pnull) forms the root of a tree with transformations being applied that extend the tree upwards towards its leaves. Any software problem transformation that leaves an empty set of problems above the line, i.e., with n = 0 in the above rule, completes a branch of the tree by providing a software solution. Figure 2 shows part of a development tree (from [2]).

3. Software problems

Environment. Our treatment of the physical world follows Jackson’s Problem Frames [15]; a software problem’s environment is characterised by the domains located therein, a domain being a set of related phenomena (i.e., events, commands, states, etc) that are usefully treated as a behavioural unit for some purpose.

As a structure, an environment Env is a collection of named domains D1, ..., Dn each a named description, Name: Description, described in terms of its known, or indicative, properties [15], and which interact through their sharing of phenomena. Behaviourally, a domain maps a collection of phenomena to a timeline of their occurrences and interactions [16].
Example: Returning to our example, in which

\[ \text{Env} = \langle \text{Smart phone, Weather, Weather station, Umbrella} \rangle, \]

within the \text{Weather station} domain, the \text{barometric pressure} phenomenon shared with the physical environment would vary according to environmental conditions; the domain could then output a \text{rain likelihood} phenomenon based on how it interprets changes in \text{barometric pressure} which is, in turn, used as a trigger to the recipe.

Associated with each domain \( D \) are three alphabets of phenomena:

- the \textit{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 \text{Weather station} domain these included \text{barometric pressure}
- the \textit{controlled} alphabet: the phenomena visible to, and that can be shared by, other domains, but that are controlled by \( D \). Again, for the \text{Weather station} these include might \text{rain likelihood};
- the \textit{unshared} alphabet: all phenomena of \( D \) that are not visible to other domains, and so not sharable with them. For the \text{Weather station}, these might include myriad internal communications and processing phenomena.

As in the example, descriptions of a software problem’s elements may be in any relevant description language; indeed, different elements can be described in different languages. They can also be more or less ambiguous in their manipulations of phenomena, depending on the stage of the design.

\textit{Need}. A software problem’s 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, \( \text{Need} = N : D \). A need’s description should always be interpreted in the \textit{optative} mood ([15]), i.e., as expressing a wish. There are two alphabets associated with a \textit{Need}:

- \textit{refs}: those phenomena of a problem that are \textit{referenced} by a need description.
- \textit{cons}: those phenomena of a problem that are \textit{constrained} by a need description, i.e., those phenomena that the solution domain’s behaviour may influence as a solution to the problem.
Software Solution. A software solution is simply a domain, $S = \text{Name} : \text{Description}$, that is intended to solve a problem, i.e., when introduced into the software problem’s environment, the software problem’s need will be satisfied. As such it may have one of many forms, ranging from a high-level specification through to program code.

Software Problem Transformation Schemata. Akin to proof theoretic approaches, our theory sees Software Engineering as the step-wise transformation of a problem into problems that are easier to solve. In concert with [17]’s observation that SE proceeds through a set of identifiable development practices, we observe that there are classes of problem transformations – such as sense making [18] – that recur during software problem solving which have similar rationale forms. To this end, we define a software problem transformation schema as named classes of problem transformations, describing the general way in which the conclusion problem is related to the premise problem(s) through its step rationale. As such as developer will know what is expected of them when they apply a developmental step.

Example: Environment Interpretation\(^3\) is a form of sense making of the environment of a problem, by which detail is added to the environment’s description:

$$\frac{E'(S) \ meets_G N}{E(S) \ meets_G N \ [\text{Environment Interpretation}]}$$

The obligation to provide rationale that will convince $G$ is the condition that must be discharged for environment interpretation to preserve the solvability of the problem with respect to $G$; the rationale should explain why the new environment, $E'$, is preferred to the original, $E$; the meaning of ‘preferred’ will, in general, be defined by $G$ and their context.

We note that the specific form of the step rationale is not mandated by our theory. Indeed, in different contexts, it has taken anything from an informal role as a risk management device [19] not shared with stakeholders to being adopted by General Dynamics UK as part of their revised high integrity software process for use with avionics applications, following Mannering’s PhD [20].

\(^3\)Previously called Context Interpretation in [2].
In manipulating the environment of a problem, Environment Interpretation works in the ‘problem space’ (as does Need Interpretation) to record improved environment understanding. Another, Solution Interpretation, works in the ‘solution space’ to record improved solution understanding. Solution interpretation also introduces an important theme for the remainder of this paper – overlapping, or tangled, problems.

3.1. Architectural decomposition; ‘Tangled’ software problems

Tangled problems were introduced in [21], which gives a definition and complete initial taxonomy of the ways in which problems tangle, together with explanation of how economic value might be delivered by those involved in tangled problem solving. Simply put, two problems tangle when their respective elements overlap.

We have already seen two problems that tangle: Michael’s and Gordon’s problems differ only in their stakeholder so both share Env and Need. The importance of tangled problems is that their solutions may have dependencies: for instance, for $P_{Michael}$ and $P_{Gordon}$, because Env and Need overlap, they can share a solution. In more complex cases, the dependencies may also constrain the solutions.

One source of tangled problems in our theory are the co-design problems that arise when a solution is structured via an architecture.

An architectural structure, or AStruct, (a development of [22]) can be used to add structure to a solution domain through an application of a software architecture (see, for instance, [23]). An AStruct combines, in a given topology, a number of known software solution components (the $C_i$) with software solution components yet to be found (the $S_j$):

$$\text{Name}[C_1, ..., C_m](S_1, ..., S_n).$$

**Example:** An example of the AStruct that encodes the Krasner variant of the Model/View/Controller ([25]) is:

$$\text{MVC}[\text{User}, \text{Display}](\text{Model}, \text{View}, \text{Controller})$$

with phenomena as detailed in Figure 3
Through solution interpretation, the solution domain is replaced by the parametrised AStruct and followed by a new software problem transformation Solution expansion. Solution expansion generates premise problems by moving the already known components $C_i$ to the environment — expanding the problem environment — whilst simultaneously refocussing the problem to be that of finding each of the $n$ solution components. The requirement and environment of the original problem is propagated to all sub-problems.

Solution expansion has the following form:

\[
P_1 : \langle E, C_1, \ldots, C_m, S_2, \ldots, S_n \rangle (S_1) \text{ meets}_G N
\]

\[
\vdots
\]

\[
P_j : \langle E, C_1, \ldots, C_m, S_1, \ldots, S_{j-1}, S_{j+1}, \ldots, S_n \rangle (S_j) \text{ meets}_G N
\]

\[
\vdots
\]

\[
P_n : \langle E, C_1, \ldots, C_m, S_1, \ldots, S_{n-1} \rangle (S_n) \text{ meets}_G N
\]

\[
P : E(S : \text{Name}[C_1, \ldots, C_m][S_1, \ldots, S_n]) \text{ meets}_G N
\]

In syntactically expanding the architectural definition, solution expansion does not generate an obligation to provide a step rationale.

Solution expansion creates $n$ premise problems — one for each unknown solution component $S_i$ — each of which contributes its solution to the other premise problems. The reader will note that premise problems $P_j$ and $P_k$ ($j \neq k$) share the original environment $E$, the $m$ known components $C_j$, the requirement $N$ and the $n$ solution domains $S_i$. The one thing that distinguishes $P_j$ from $P_i$ is that, whereas $P_j$ has $S_j$ as solution domain, $P_k$
has $S_j$ as an environment domain. Thus, any solution interpretation of $S_j$ as part of problem $P_j$ leads to further detail in the environment of $P_k$ (and *vice versa*). In this way, the solutions to the $P_i$ must be solved, or *co-designed*, together. This will become clearer with an example.

**Example:** The MVC AStruct includes two ‘already known components’ and three ‘to-be-designed’ components. On expansion, we are left with three sub-problems, one for each ‘to-be-designed’ component:

$$P_M: \langle \text{User, Display, Controller, View} \rangle \text{(Model)} \text{ meets}_G \text{ Need}$$

$$P_V: \langle \text{User, Display, Model, Controller} \rangle \text{(View)} \text{ meets}_G \text{ Need}$$

$$P_C: \langle \text{User, Display, Model, View} \rangle \text{(Controller)} \text{ meets}_G \text{ Need}$$

These three problems tangle as they share elements: in fact, each is composed of precisely the same elements, only the solution domain distinguishes between them.

Examining the model problem, $P_M$, it may appear that we need descriptions of the Controller and View to be able to solve it, whereas these will only be available when problems $P_V$ and $P_C$ are solved. But $P_V$ and $P_C$ each depend on the other as well as $P_M$. This interdependence is an accurate representation of what occurs in practice, and there are many practical ways of overcoming it.

For the MVC, in the literature (for instance [26, 27]) 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 $M_{Design}$ is the description of the designed model, then we can write:

$$P_V: \langle \text{User, Display, Model : } M_{design}, \text{ Controller} \rangle \text{(View)} \text{ meets}_G \text{ Need}$$

$$P_C: \langle \text{User, Display, Model : } M_{design}, \text{ View} \rangle \text{(Controller)} \text{ meets}_G \text{ Need}$$

from which we can then address the design of the View and the Controller.

The MVC does not actually specify how phenomena must be shared between the three components: this is left as an implementation choice, often guided by other design patterns and principles. In fact, well known design patterns apply in the context of an MVC. For instance, the Observer pattern [28] is often used, in which the Model becomes the ‘subject’ observed
by the View, the ‘observer’: every time the subject changes, the observer is alerted of the change, which triggers appropriate behaviour, such as refreshing data displayed. This type of sharing respects the principles that the Model should be independent of its presentation. This design maintains the independence of the Model, which does not need to know anything about the View. Similarly, the Strategy pattern [28] is often used between View and Controller, by which the View delegates to the Controller the responsibility to handle user actions. In fact, this is a standard way of coupling View and Controller within application frameworks. In summary, heuristics have developed through practice to help developers tackle difficult co-design problems as that captured by the POE problems above.

In POE, as well as those tangles generated by solution expansion after solution interpretation, there are other routes to tangled problems – we examine one such in the sequel.

4. Developmental Risk in Software Engineering

Environment interpretation is a problem exploration step by which improved understanding of a problem’s environment is recorded: other examples of problem exploration steps include Need Interpretation and problem progression [2] by which, respectively, the need for the software solution are refined; a real-world domain is removed from the problem, and the need rewritten to compensate (see [15] for the original description of problem progression).

The aim of problem exploration is to understand the problem — the environment and the need — so to form a description of it; mutatis mutandis the solution. Typically, problem and solution exploration will not continue until ‘perfect’ problem or solution descriptions are arrived at. Rather, early problem explorations will interleave with early solution explorations until the required level of understanding is gained. Iteration between problem and solution domains is typical of Software Engineering endeavours ([29, 30]) in which the intention is to avoid a ‘single-pass sequential, document-driven, gated-step approach’ [30]. Iteration is also the basis of modern agile methods [31] for which the evolutionary advancement of a software solution is key.

Both problem and solution exploration are resource intensive – a problem exploration team will, for instance, expend resources exploring the problem
with stakeholders, drafting software requirements, capturing domain descriptions, *etc*; a solution exploration team will, for instance, spend their time architcting or prototyping, collaborating in scrums, writing test harnesses and documentation, *etc*.

Solution exploration therefore commits resources to uncertain outcomes — which is a process risk: that the solution will have been inadequately understood and described, and so will fail to satisfy its stakeholders, perhaps necessitating rework (for an example, see [32]). Likewise with problem exploration the risk is potentially greater in that inadequate understanding will incur the loss of any resources that had been committed to concomitant solution explorations.

Risk can be managed through validation — by which problem and solution understanding are checked during development with stakeholders, including the problem owner, lowering the risk of precipitous commitment of resources to poorly understood problems and solutions ([1]). The resulting interaction of exploration and validation steps results in the problem solving process pattern shown in Figure 4 in which iterative problem and solution exploration can be interleaved with any amount of validation checkpoints.

It is worth noting that the pattern mandates neither how much exploration should be completed before validation (whether problem or solution), nor that problem validation should occur before solution exploration begins and *vice versa*, as to do so would negate our goal of capturing the essential relationships among practices within Software Engineering. Work is in progress to characterise the costs of exploration and the trade-offs involved in balancing risks so that these can be better understood.

5. The Interplay between function and quality

According to Summerville [33], software needs can be classified into functional and non-functional requirements:

**Functional requirements**: ‘statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.’

---

5Sommerville also includes Domain Requirements, i.e., constraints imposed by the domain, which can be ‘either functional or non-functional’. In our theory, domain requirements are indicative properties of the environment domains.
Non-functional requirements: ‘constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.’

We have argued elsewhere [34] that the nature of functional requirements — essentially that they determine an output given an input — leaves them vulnerable to implementation in a hostile environment. For instance, consider the functional requirement

‘notify me whenever the weather forecast indicates that it will rain tomorrow.’

by which Michael’s need is determined above. Now, whereas it was practically trivial for Michael to satisfy his need using [ ], ensuring that the desired behaviour makes it into the real world — across what we have elsewhere called the ‘Turski disconnect’ [35] — is rather more difficult. The issue is that the real-world is a very aggressive place: a notification — in this case an SMS — may be delayed or lost by a mobile operator, for instance; the [ ] servers may be subject to maintenance meaning that the recipe is never run. The terms of use (ToU) for [ ] include the statement:
‘However, there will be occasions when the Site and/or Service will be interrupted for maintenance, upgrades and emergency repairs or due to failure of telecommunications links and equipment.’

specifically to indemnify themselves from the effects of such failures.

Reliability is, of course, a non-functional requirement. If Michael’s requirements are for a more reliable service than can be guaranteed under Michael’s ToU, then Michael will not provide him with a solution. However, Michael may ultimately be satisfied with a less than perfect service: during problem exploration, alongside his functional needs, his quality needs would also be captured as part of his problem.

Notwithstanding the achievement of formalists in codifying the machine and in producing techniques to reason about absolute correctness, in the real world, without the mitigating effects of non-functional requirements, functional requirements are unsatisfiable. Equivalently, when $F$ is a purely functional need

$$\text{Env}(\text{Soln}) \text{ meets}_G F$$

has no solution, i.e., no software Soln can meet functional $F$.

In contrast, with a judicious choice of non-functional $Q$

$$\text{Env}(\text{Soln}) \text{ meets}_G Q \land F$$

may have a solution$^6$.

Besides mitigating harmful effects on functional requirements in the sense just discussed, non-functional requirements also constrain properties of a system and its development process. In particular, according to [37], non-functional requirements can be divided into execution qualities, which constrain the run-time properties of a system, such as availability and reliability, and evolution qualities, which constrain design-time properties, such as testability or flexibility.

An important insight from Software Engineering is that such qualities can often be factored into a system via a judicious choice of architecture. For instance, reliability may well result in choosing a triple module redundancy [38] architecture, while the MVC might be chosen when flexibility of user interfaces is needed. This particular relationship between qualities and architectures can be captured in our theory as follows.

$^6$Incidentally, this provides us with an interpretation of the notion of satisficing [36] as ‘[that] permit[s] satisfaction at some specified level of all of its needs’.
in which AStruct $A_{ExQ}$ mitigates the harmful effects of the real world and which, thus, characterises the relationship between many of the important elements of Software Engineering, including environment, software components, functional and non-functional requirements, architecture and design rationale. Subsequent interpretation and expansion of $Env$ may lead to a simpler, formalisable problem of finding some combination of the $S_i$ that discharges $F$.

Equation 1 is a variant of *problem progression* as defined by Jackson [15] and expounded in [39]. This suggests that AStructs are a general vehicle for problem decomposition whose use is motivated by non-functional requirements.

6. Summary

We have so far discussed a special theory of Software Engineering, which we refer to as Problem Oriented Software Engineering (POSE). We have seen how it allows the expression of software problems and of processes for their systematic solution, and have argued that its logical basis, inspired by propositional calculi, allows us to bridge between the formal and the non-formal, an essential endeavour in Software Engineering. We have exemplified salient features of the theory and the way it accounts for key Software Engineering concepts and practices, such as the use of architectural structures for radical design or for reuse, the interplay between functional and quality requirements, the tangling of software problems through co-design, and the use of validation as the means to mitigate developmental risks.

There is, however, at least one criticism that can be made of POSE: in considering software as the only solution technology, other engineering problems that arise naturally within software problem solving, say, the need for documenting design rationale or safety cases, cannot be treated uniformly within the theory. For instance, the form of the step rationale $J$ (see Section 2) is not prescribed by POSE, so that for practical application it may take diverse forms depending on the context in which the software is developed. For example, for safety-critical systems [20] develops a form of step rationale able to support the expression of assurance cases, and essential part of an integrated safety and development software cycle.
This begs the question of whether we can generalise our theory to define an homogenous theory in which software is properly located alongside a wider range of solution technologies deployed in Software Engineering and beyond. One such generalisation is the subject of Part II.

Part II

A General Theory of (Software) Engineering

7. From Software Engineering to Engineering

Rogers writes [7]:

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

Software is the most malleable of the product technologies, and has been used to enhance the properties of myriad others. For example, in combination with laser, material deposition and/or stereolithography technologies we achieve the 3D printer, a device that can make anything. In combination with automative technologies, we get a car that drives itself. With medical technologies, we get an artificial heart.

Product technologies surround software to embody it. As already mentioned, in a PC, the operating real-world environment will contain computer hardware with installed software environment such as an operating system or a database system upon which the software artifice will run. That will be connected to its environment through, typically, a mouse and keyboard that the operator will manipulate to control the software. In an embedded system, the interface of the software to the environment will be through sensors and actuators.

In all of these systems, given software requirements, it is theoretically possible to conduct the engineering of the software solution in isolation from that of all the other product technologies [29]. Practically, however, this is unrealistic and, anyway, ignores the evolutionary system gains that can be achieved through iteration. Boehm writes of his spiral model [40], for instance:
the spiral gets started by a hypothesis that a particular operational mission [...] could be improved by a software effort. [...] Usually, experience with the operational mission leads to further hypotheses about software improvements.

As so far defined, a software problem can have as solution no product technology other than software; and yet we know that software is often just a component of a complex system that involves many product technologies. Moreover, as we have seen, even when the solution is software, there are engineered artefacts that arise during Software Engineering that are not software: the assurance case for instance, use cases, product manuals, user-training materials, etc.

In our theory so far, then, we would have to give these special consideration outside of the theory. The remainder of this article considers the benefits of relaxing the constraint that the solution be pure software.

8. POE problems

We first observe that POSE actually placed little reliance on the software nature of the solution domain; thus, many of the technical details carry across unchanged from the previous sections, including the notion of an engineering problem:

\[
\text{Env(Soln)} \text{ meets}_G \text{ Need}
\]

Our intention is now that Soln range over (all combinations) of product technologies. That the solution domain does not regard software as special means that we are defining an homogenous, more general theory of engineering that includes the special theory of Software Engineering defined above.

Only minor reinterpretations are needed in this general theory: for AStructs, as defined in Section 3.1, both known and unknown components can also now range over combinations of product technologies; as an example, we might develop an Enterprise Architecture [41] for an organisation. Of course, as we focus down through the levels of the system, through solution expansion, the AStructs would become more and more specialised until we come to a single product technology, say, the MVC AStruct that is used to structure interactive software that forms part of the enterprise information system.

Moreover, the homogenous nature of POE allows more to be said about the process pattern of Figure 4.
8.1. The POE Process Pattern

The expansion of the solution domain allows us to consider artefacts that arise as part of the Software Engineering lifecycle, but that could not be considered as solutions under POSE without special treatment.

One source of such artefacts is the collection of validation problems that arise during problem solving in which validatable representations of current understanding are created to be shared with stakeholders:

Example: A safety case [42] is a documented body of evidence providing a compelling, comprehensive and valid argument that a system is adequately safe for a given application in a given environment. It is written for the regulator, a solution validator, and must satisfy their requirements for a safety case in the regulatory environment.

In this case, a safety-critical software solution cannot be deployed without the regulator’s sign-off, so that problem solving consists of finding both a system that solves the Customer’s problem and a safety case that satisfies the Regulator.

Whereas this was not possible in the heterogeneous POSE, in our homogenous POE we have the problems:

\[
\text{Customer\_Environment, \textit{Software\_Solution meets}_\text{Customer \textit{Customer\_Needs}}}
\]

and

\[
\text{Regulator\_Environment, \textit{Safety\_Case meets}_\text{Regulator \textit{Regulator\_Need}}}
\]

which have the same general form, that of a solution validation problem, viz.:

\[
\text{Solution\_Validation\_Environment, \textit{Solution\_Description meets}_\text{Solution\_Validator \textit{Solution\_Validation\_Need}}}
\]

in which the Solution\_Validation\_Environment is a description of the environment in which the validation of the solution will be undertaken, and the Solution\_Validation\_Need is the need to be met for the solution to be successfully validated. A safety case above is the solution to a solution validation problem.

A problem validation problem has an analogous form, viz.:

\[
\text{Problem\_Validation\_Environment, \textit{Problem\_Description meets}_\text{Problem\_Validator \textit{Problem\_Validation\_Need}}}
\]
We note that the Software Solution and the Safety Case are interdependent – neither solve the core problem individually. Indeed, they can be seen as projections from a single core problem which combines both Customer’s and Regulator’s problems.

In the general case, suppose we have a collection of problem (PSH) and solution (SSH) stakeholders, with \( SH = PSH \cup SSH \), then the validation of a core problem \( P : Env(Soln) \ meets_{SH} Need \) depends on all its validation problems being solved, i.e.,:

\[
\bigwedge_{G \in PSH} PVC_G(Env_G, Need_G) \ meets_G PVN_G \\
\land \bigwedge_{G \in SSH} SVC_G(Env_G, Soln_G, Need_G) \ meets_G SVN_G
\]

(2)

(where \( Elem|_G \) is the projection of problem element \( Elem \) for stakeholder \( G \)).

We note that many real-world developmental artefacts can be considered as the solution to validation problems. For instance, discussions that lead to an agreed (i.e., validated) collection of use-cases [43] can be seen as a technique for producing a problem description that satisfies the problem validation problem. Moreover, discussions that lead to an agreed collection of acceptance tests can be seen as a solution description that satisfies the solution validation problem. Requirements engineering, consisting of elicitation, analysis, and specification, can be seen as a technique for partial problem exploration; pattern oriented analysis and design is a technique for partial solution exploration. The reader will, no doubt, be able to come up with more.

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 POE problem solving process composition which is the embedding of problem solving instances within problem and solution explorations, leading to the upper part of Figure 5. This we term fractal composition in cognisance that validation problems may again have embedded validation problem associated with them, the embedding process proceeding until problems are encountered in which validation is not required.

Other forms of composition on the POE Process Pattern (PPP) can be defined including sequential and parallel composition (see [1]) where:

- sequential composition is the identification of end state of one problem solving instance with the start state of another;
- parallel composition is the concurrent running of more than one problem solving process

and as illustrated in the lower part of Figure 5.

Figure 5: The POE Process Pattern (PPP) with (above) embedded (or fractal) problem solving processes and (below, left) sequential and (below, right) parallel composition of problem solving processes (see text).

8.2. Risk revisited

In the macro sequencing of problem solving steps shown in Figure 4, backtracking due to unsuccessful validation was limited to the accumulation of resources between problem and solution exploration phases. This limited the risk too. With the more complex POE process patterns shown in Figure 5, the potential for backtracking in POE, 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.
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 trade-off between validated and unvalidated problem solving.

Example: Don Firesmith [44, page 25] citing the selection of the Pratt and Whitney PW2037 Engine for the Boeing 757 ([45]), talks about the need for architecture re-engineering in the light of inadequately specified quality requirements:

[...]

In this case, inadequately specified quality requirements (as might be discovered through detailed but incorrectly or unvalidated, problem exploration) causes backtracking spanning many developmental activities. One remedy would have been to validate the specified quality requirements/architectural choices early; Hall et al. discuss this issue in the context of safety-critical systems in detail in [46] introducing preliminary safety analysis, a form of solution validation, the goal of which is to:

(a) confirm any relevant hazards allocated by the system level hazard analysis; (b) identify if further hazards need to be added to the list; and (c) analyse an architecture to validate that it can satisfy the safety targets associated with the identified relevant hazards.

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 iftttt recipe is, to some extent, a throw away activity, needing no validation. Even in an organisational setting, an end-user created spreadsheet may not need validation if nothing mission-critical rests on it.
In not requiring validation, these problem solving activities are, in some sense, ‘trusted.’ 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 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. We also note 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’s environment and need need not incur untenable levels of risk.

9. Socio-technical system engineering

We now briefly examine socio-technical system engineering within our general theory. By socio-technical system we mean systems that involve the interaction of humans and technology. Our focus here is information systems. Such systems form the basis of organisations and their development spans much of the work of Information Systems within.

Figure 6: The 3 ellipse model of socio-technical systems, adapted from [47]

Hall and Rapanotti ([47]) introduce the three ellipse model for socio-technical systems, as shown in Figure 6. Alongside the software solution,
IS_Soln, with the introduction of the Human\textsuperscript{7}, we identify and separate two new areas of interest, which now form explicit foci for design:

- the IS/Human Interface which determines the Human-Machine interface, which comprise the tools by which the Human knows the state of the system and vice versa and which will require code to be written and a UI interface to be developed; and

- the Human_Soln which determines the knowledge and behaviour that is expected of the human as a component of the socio-technical system and for which training materials may need to be written.

The 3 ellipse model is suggestive of elements of an architecture, A\textsubscript{ST}, that captures the relationship between the human and the Information system:

\[ A_{ST} \left[ (IS/\text{Human\_Interface}, IS\_Soln, Human\_Soln) \right] \]

As there are 3 to-be-designed components, on application within a socio-technical problem, this AStruct produces three tangled problems, one each for the design of the IS_Solution, the IS/Human Interface, and the Human_Soln.

\[ P_{IS/\text{Hint}} : \langle \text{Env}, IS\_Soln, Human\_Soln \rangle (IS/\text{Human\_Interface}) \text{ meets Need} \]

\[ P_{HSoln} : \langle \text{Env}, IS\_Soln, IS/\text{User\_Interface} \rangle (\text{Human\_Soln}) \text{ meets Need} \]

and

\[ P_{ISSoln} : \langle \text{Env}, Human\_Soln, IS/\text{User\_Interface} \rangle (IS\_Soln) \text{ meets Need} \]

As before, the question now arises as to which of these three co-design problems effort should initially be focussed on. This reduces to a question of which elements of the Need will be catered for by Human actions and which by Information System actions.

If we assume that the Need is to be able to collect, store, manage and interpret data from its many business activities, then many choices arise:

- the organisation might consider an off-the-shelf software package, such as SAP;

\textsuperscript{7}The description of the Human as a product technology doesn’t quite feel appropriate, even though we will be designing materials that address their future suitability in the role of User.
if the expertise exists within the organisation, it might build a bespoke system;

if the expertise does not exist within, it may commission a system.

Each choice has ramifications for the order in which the co-design problems are solved; simplifying slightly:

- for SAP, $P_{ISSoln} : SAP$ is a solution which focusses subsequent design on the relationship of the Human with the Organisation, i.e., the changing of business processes to meet the requirements for integration of the SAP system;

- in which case, the allocation of function between the Human and the Information System could be a priority. Actually, this area is well explored in the literature, including Fitt’s early MABA-MABA (‘Men Are Better At–Machines Are Better At’) lists [48], Price’s Decision Matrices [49] and, more recently, Dearden’s scenario method [50].

- in which the further development of the Need, in collaboration with the external provider, would come first.

As previously, POE mandates no particular method of problem solving, and many other alternatives exist.

10. Related Work

The theories we have presented were originally motivated by wishing to broaden the applicability and practicality of Jackson’s Problem Frames: in [51], Jackson covers the ‘problem space’ topics of ‘functional requirements, software specifications, and the path by which you get from one to the other.’ With POSE the intention was to improve their practicality by extending and connecting across both problem and solution spaces to include navigating from non-functional requirements to architectures. With POE the intention was to apply to engineering what was learned in the development of POSE, incidentally allowing Software Engineering to be embedded therein. Consequently, in POE we have built complex problem solving processes that characterise Rogers’ notion of engineering [7] and which comment on Jackson’s untreated concerns, including ‘how to elicit requirements, how to make the business case, how to manage the project, to facilitate meetings, or negotiate compromise [51, page 5].
10.1. POE as a Design theory

POE has features of a design theory in the sense that it concerns problem solving within and via the design and creation of artefacts. As defined by [52], design problems are real-world problems which are ill-structured and complex: ill-structuredness implies that the starting point is often vague goals and intentions, with unclear success criteria, hence many degrees of freedom in the problem statement and no unique path to solution ([53, 54, 55]); complexity refers to number of issues and variables involved and their relationships, and to their stability over time and uncertainty beyond the problem solver’s grasp or control ([56, 57]), which makes design problems difficult to solve. Of particular notice is the explication of the relationship between unsatisfied goals or needs which motivate, inform and instigate the design activity, and provide evaluation criteria for the designed artefact – and constraints – which determine what is feasible by establishing relationships between properties and features of the designed artefact and its context. In POE, the relationships between goals and constraints is explicated by structuring a problem conceptually as three distinct parts: the need to be satisfied, the solution to be designed and its context, whose properties constrain the design activity. Each part is characterised in terms of domain specific phenomena, which are grounded in the problem domain, requiring specific domain expertise for their identification and definition. Moreover, constraints do not just arise from the context: they also emerge as part of the design activity when earlier design decisions narrow down the range of possible future design choices. In POE, the introduction of constraints as a result of design choices is achieved through transformation steps which allow descriptions of the solution to be provided on which future design choices will be based.

Another important characteristic of design problem solving is the possibility for alternatives, intended as artefact concepts, that is ideas of what the designed artefact may ultimately be like. Differently from other type of problem solving such as mathematical proof or case-based analysis, design alternatives may be generated via experiential knowledge (personal and/or collective) and creative imagination, alongside reasoned analysis. As we have discussed, POE problem solving proceeds by following a regular pattern of discrete problem transformation steps which may affect each problem part separately, with backtracking allowing the exploration of alternative paths to solution.

Domain knowledge is seen as fundamental in design problem solving,
with an ensuing tension between generic and domain specific problem solving skills and approaches. The argument for problem solving being highly domain specific is based on the observation that it often relies on “cognitive operations” specific to that domain ([52], citing [58, 59, 60]). More precisely, in the process of structuring ill-structured problems, domain knowledge and expertise is needed to separate important from irrelevant information relative to that domain, as well as providing domain-specific representations, say, a software engineer using a state machine diagram to capture the behaviour of a software component. Domain and experiential knowledge is particularly important and frequently employed, often taking the forms of repertoires of proven, ready-made solution alternatives which can be readily adapted to the specific problem at hand: design patterns in architecture [61] and software development [62] are notable examples. As demonstrated in examples, POE’s AStruct allows captured knowledge to be factored into the problem solving process.

Representation is pervasive to design problem solving to a point to be considered the ‘language of design’ [3], used both as a means of communication, to drive the design process, both for analysis and synthesis, and to record design decisions and rationale. As [63] puts it, solving a design problem can be conceived as

‘representing it as to make the solution transparent.’

Particularly in the psychology literature concerned with complex problem solving by humans, representations are often linked to the structure of the external representation of a problem, with expert problem solvers combining heuristic thinking [64] and representations to cope with difficult complex problems [56]. Indeed, representation is integral to POE, which provides the means to represent explicitly both problems, their transformations, design rationales, validation arguments and the process to solution followed.

Besides the need for increased precision in the design activity, representations also differ to provide appropriate descriptions of problems vs solutions, which often use languages which are remote from each other, say some expression of users’ goals vs. formal drawings of a complex technical solution. In POE, regular classes of transformations are defined to allow problem descriptions to become more precise as the design activity unfolds.

---

8 Although in this paper we have focused primarily on textual representations, POE also provides accompanying graphical representations, see e.g., [65].
and diverse problem and solution-specific languages and notations are allowed within the same problem representation. Such linguistic diversity, while essential to capture phenomena of diverse nature and for communication with diverse stakeholders, may hinder the validation process: establishing whether the solution addresses the problem is not a simple issue of formally proving satisfaction of one description by the other, as it may be the case for mathematical problems, or be shown to be optimal in the presence of alternatives, as is the case in optimisation theory. Instead the notion of satisficing [63] was introduced in design problem solving as the notion of establishing some threshold of acceptability of a solution, which does not necessarily imply logical satisfaction or optimality. In POE, the establishment of thresholds of acceptability is embedded in transformation rationales and validation steps involving stakeholders which also provide the locus for conflicting goals to be explored and risks mitigated.

Representation is often complemented by decomposition as a further means to deal with complexity: this, of course, presents its own challenges as to know when and how to decompose, and problem solvers often resolve to domain or experiential knowledge to partition a problem and the problem solving activity into a set of meaningful parts and tasks. In POE, specific classes of transformations are defined to allow the decomposition of problem parts, whether through analysis (particularly of properties of the context) or creative ingenuity or experiential knowledge (particularly in structuring and definition the solution), like architectural interpretation and expansion transformations we have exemplified in the paper.

10.2. Design vs Engineering

There are strong links between the concepts of design and engineering, and often some confusion too. If we were to compare Rogers’ definition of engineering (see Section 2) the notion of design theory just discussed, we would find many similarities: both have to do with creating an artefact; both see a ‘need’ as a starting point; both are situated in a real-world context.

There are, however, differences in the use of the term ‘design’ due, primarily, to historical reasons: in the context of engineering the term design was, and still often is, used to denote a specific activity within the engineering process in which some representation, usually of a technical nature, of the artefact to be constructed is provided prior to construction. Indeed, Software Engineering, which first originated from traditional manufacturing engineering, still makes such a use of the term. On the other hand, design
theory has generalised the term design to its much wider meaning which we’ve just discussed. In fact, capitalisation is sometimes used to stress this distinction: Design (with a capital D) to indicate the new meaning vs. design (in lower case) for its older use. Indeed it is the former meaning we are concerned with in this paper.

With this clarification the similarities between design and engineering become even more obvious. If there were a notable difference between the two is perhaps in what is hinted by the ‘practice of organising’ part of Rogers’ definition: engineering strives for systematic ways of solving design problems, something which in design may or may not be a concern. This is because engineering is often constrained by issues of repeatability, efficiency and quality control. For instance, manufacturing design problems need to be solved as to minimise production costs and product defects, which require a robust and repeatable design process. In safety engineering, solutions must be safe on first use, with very little, if at all, margin of error: this too required a very structured process that can be appropriately monitored and assured. From a discipline viewpoint, therefore, it is not surprising that engineering has a greater propensity towards disciplined problem solving methodologies than design does.

An observation which is sometimes made of engineering problems is that, as complex as they may be, they are also often well-structured and of a technical nature, hence not as difficult to solve as more general socio-technical design problems. This is a claim that most practicing software engineers today would probably dispute, and that, in fact, has also been refuted by academic studies. For instance, [66] reporting on an empirical study of engineering practice found that: engineering projects exhibit a large mix of well-structured and ill-structured sub-problems, the latter often arising from multiple and contrasting sub-goals, and indeed engineers need to deal with ambiguity as well as complexity; even initially well-structured problems often become unstructured as a result of unanticipated phenomena, often after project completion and solution deployment, and due to a combination of engineering and non-engineering issues; no unique path to solution exists and the engineer often applies professional judgement and previous experience; engineers use many forms of representation, with drawing being prominent; knowledge and expertise needed to solve engineering problems is distributed not just across the members of a team, but within the organisation and across organisational boundaries; and engineering problems are constrained not just by engineering standards, but by a wide range of other
considerations, from legal to ethical, cultural and economical, and indeed project success is measured primarily in terms of customer satisfaction and completion in time and on budget.

10.3. Design problem solving vs systems thinking

Perhaps complementary to design problem solving is systems thinking [67], which can be regarded as a discipline aimed at understanding and influencing how complex systems behave. It emphasises that a system is more than the sum of its interconnected parts, and needs to be understood as a whole for problem solving and decision making.

Within systems thinking, a number of methodologies have been developed which are broadly categorised as either hard or soft. Hard methodologies are oriented towards problems which are amenable to quantification and measurements, and are often based on mathematical and statistical methods, often making use of formal models and computer simulations of the system: such problems may be complex, but not necessarily ill-structured, and the problem solving process relies primarily on objective data. By contrast, soft methodologies are oriented towards complex, ill-structured problems and rely primarily on people’s opinions and judgement, shared understanding and learning.

Checkland’s soft systems methodology is perhaps the best known soft approach in systems thinking [68, 69]. Under this methodology, problem solving proceeds through a learning process in which people reach some shared understanding of a problematic situation within a system, and arrive at an agreement of some positive action needed to address it, which takes the form of some intervention within the system to change its behaviour. Informal sketches and rich pictures of the system may be used to support the development of such an understanding.

While both design problem solving and systems thinking can be applied to complex ill-structured problems, their approach is different: design problem solving sees addressing the problem as the construction of an artefact, while systems thinking sees it as understanding an existing system to establish where to intervene to affect the desired change. It could be argued that they complement each other in the sense that understanding the context in which the artefact is located is a systems thinking activity: such context is a system for which artefact creation constitutes an intervention, which will affect the system behaviour. This complementarity is also reflected in the types of representation the two approaches encourage: design problem
solving tends towards representation of the designed artefact, while systems thinking towards that of its context.

Despite such complementarity, the literature on combining the two is very sparse. From a POE perspective, we could argue that systems thinking is primarily situated within problem exploration, while design problem solving crosses the problem/solution divide repeatedly. In its current form, however, POE is concerned with intervention in a system via ‘green-field’ solutions: a new artefact is to be designed to meet an identified need in context. Interventions where change is needed are more difficult to accommodate within our current theory and, in fact, are the focus of current research on extending POE to deal with this sort of system analysis.

11. Conclusions and Future Work

In an essay on theories of information systems [70], Gregor argues that what distinguishes such theories from other fields is their concerns with artefacts and interactions in human-machine systems, requiring links to be established between the natural, the social and the designed. In adopting an ontological position, she regards theory as an abstract entity, separate from the subjective understanding of individual researchers, which aims to ‘describe, explain, and enhance understanding of the world and, in some cases, to provide predictions of what will happen in the future and to give a basis for intervention and action.’ Both in intent and formulation, POE aligns with Gregor’s definition of theory in that it strives to bring together phenomena of interest to Software Engineering by introducing a systematisation of concepts and practices [17], in a way that attempts to capture how software engineers go about addressing real-world problems through the design and construction of software, within their specific development context and taking into consideration the fundamental interplay between the technical and the social. It should be stressed, however, that it is not POE’s intention to reduce Software Engineering to some prescriptive process or method. Instead, the theory accounts for key relationships among the phenomena of interest and how these unfold in the Software Engineering process. In doing so, it aims to bring together within an explicit scientific theory some of the implicit theories [71] which are already embodied in current Software Engineering practice.

The paper has discussed at length the analytical and explanatory features of POE, including the phenomena of interest within the theory and their relationships, and has demonstrated the level of generalisation afforded
by POE, extending to a wide range of key Software Engineering concepts and practices, from domain modelling, requirements capture, solution development, etc., into sensemaking operations present in problem and solution understanding, to characterise architectures as generalised problem decomposition operations that extend divide and conquer, and other traditional problem solving techniques. Some of POE’s explanatory power has also been discussed, particularly in Section 7 where the relation between design and validation problems was explored, which allows developmental risk to be managed.

Prediction is also a desirable quality of a theory. Although we haven’t touched on POE’s predicted power in this article, some of its prediction capabilities have been exercised in the context of real-world practice, particularly aimed at business process re-engineering. For instance, predictions based on the PPP led to a revised high integrity software process for use with avionics applications within General Dynamics UK [20], where the introduction of pre-design time validation led to a reduced error rate downstream in the process. Similarly, the application of POE within a Bank of America’s subsidiary [32] led to a revised process for dealing with software defects in business critical software, where the introduction of an early validation formal step led to a reduction in software solutions been rejected at the acceptance stage. Ongoing research [72] is aiming to expand such predictive capabilities further toward estimation: the intent is to associate appropriate metrics able to quantify the trade-offs and benefits which could be derived from business process re-engineering prior to implementation.

A concurrent strand of research is looking at extending POE beyond green-field design, focusing on problems which typically emerge in the context of organisational change. Early results [73] appears to indicate that an extension to change problems is feasible by introducing a theoretical layer on top of POE and appropriate, semantic preserving mapping into the underlying framework.

Acknowledgements

We acknowledge our colleagues within The Open University and elsewhere for stimulating discussions on the topic of Problem Orientation. In particular, we thank Michael Jackson for introducing us to Problem Frames.
References


[55] V. Goel, P. Pirolli, Motivating the notion of generic design within information-processing theory: The design problem space, AI magazine 10 (1) (1989) 19, for the
description of design problems in Jonassen’s taxonomy.


