The Unending Quest for Valid, Useful Software Engineering Theories

Mark Staples
NICTA, Level 4, 223 Anzac Parade, Kensington, NSW 2052, Australia
School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia
Email: mark.staples@nicta.com.au

Abstract—Using scientific disciplines as inspiration, some researchers have called for the creation of general theories for software engineering. I present a contrary view, drawing on recent work in the philosophy of engineering. Engineering theories are different to scientific theories, and are not judged by the same criteria. Software engineering researchers should strive to create valid theories about the uses of software-based systems, even if that leads to a multitude of theories that have narrow, overlapping scopes, are approximate, and have little explanatory power. The engineering imperative is that the predictions of software engineering theories should be consistent with actual behaviour of software-based systems, and that theories and should support assurances and explicit justification that systems will meet their requirements. Process theories are about the organisation of work and project management, and are relevant to cost and schedule requirements in software engineering. However, to directly provide evidence that software-based systems meet their main functional and non-functional requirements, software engineering researchers should focus on product theories, about the specifications, designs, code, and behaviours of software-based systems.

I. INTRODUCTION

Software engineering is, or is becoming, a serious engineering discipline. [1] Much of modern software development is for non-critical systems, and is more like a craft activity than engineering. [2] Nonetheless there is an increasing demand that trusted software systems be trustworthy, or dependable. [3] This means that for a critical software-based system, engineers must not only successfully deliver the system, but also deliver assurances, backed by explicit justification and supporting evidence, that the system will perform as required. Engineers in all fields provide justification that artefacts will satisfy their requirements. However, engineering theories also do not need to be explanatory, or fundamental, and so typically engineering theories concern requirements. Process theories are about the organisation of work and project management, and are relevant to cost and schedule requirements in software engineering. However, to directly provide evidence that software-based systems meet their main functional and non-functional requirements, software engineering researchers should focus on product theories, about the specifications, designs, code, and behaviours of software-based systems.

In this paper I argue that in software engineering, as with other engineering disciplines, the most important qualities of a theory are that it should be valid (be consistent with physical phenomena of artefacts’ behaviour in operation) and be relevant to engineering practice (provide justification that artefacts will satisfy their requirements). In order to remove confusion about the nature of engineering and its relationship to science, I note some of the differences between theory in the body of engineering knowledge compared to science. The most important difference is that engineering theories concern requirements. However, engineering theories also do not need to be explanatory, or fundamental, and so typically the engineering body of knowledge contains many overlapping, narrow theories. Moreover, the main core of the body of software engineering theory should be (as with theories in other engineering disciplines) product-focussed, rather than process-focussed.

To support this argument, I draw on a recent framework in the philosophy of engineering [8], [9]. This uses and adapts Popper’s [10], [11], [12], [13] philosophy of science. To provide face validity for this framework for software engineering, I illustrate how example theories from the software engineering literature fit into the framework. I then discuss implications of this view for software engineering research, and outline some limitations with my proposal, before concluding.

II. CRITICAL RATIONALISM AND ENGINEERING THEORY

Critical rationalism [10], [11] is an epistemological philosophy, proposed and used by Popper to explore the growth of knowledge in science. Consistent with critical rationalism, Popper also proposed the three-worlds ontology [12], [13], to explain the nature of objective knowledge. In a previous pair of papers [8], [9], I adapted critical rationalism and the three worlds ontology to investigate the nature and growth of engineering knowledge. In this section I provide a brief summary first of Popper’s philosophy and then of my previous adaptation of it to engineering knowledge.

A. Critical Rationalism and the Three Worlds

Falsification is perhaps the most well-known element of Popper’s philosophy. Critical rationalism is based on the deductive-nomological model of empirical theory, under which
Theories are (in principle) axiomatised as universally-quantified formal logical theories of the form:

$$\forall x. A(x) \Rightarrow P(x)$$

where $x$ are states of the world, $P$ are predicted phenomena and $A$ are conditions under which the theory applies. Any empirical observation $P(x_i)$ is also axiomatised, but as a single instantiation of logical variables, cannot deductively entail the universally-quantified theory.

Nonetheless, Popper recognised that a single empirical observation $\neg P(x_i)$ can serve as a counter-example, and so deductively falsify the universally-quantified theory. The theory must then be rejected, although it might be modified in a progressive way and re-proposed. Falsification preserves the valid use of classical deductive logic to reason about the real world, using tentatively-held empirical theories. Critical rationalism includes a tetradic schema for the evolutionary growth of knowledge: $PS_1 \rightarrow TT \rightarrow EE \rightarrow PS_2$, where $PS$ are problem situations to be addressed, $TT$ are the tentative theories proposed to apply to those situations, $EE$ is error elimination in response to falsification of the theories, leading in turn to a new problem situation. A theory is increasingly corroborated as it survives an increasing amount of severe empirical test, and is iteratively revised in methodologically acceptable ways.

Popper also proposed an ontological framework consistent with critical rationalism. [12], [13] His ‘three worlds’ framework described three kinds of entities: those in the physical world (World 1), the mental world (World 2), and the world of objective knowledge (World 3). Popper proposed that there were relationships between Worlds 1 and 2 (such as sense perception and personal action), and between Worlds 2 and 3 (such as understanding and reasoning about mathematics). Popper held that there was no direct relationship between Worlds 1 and 3, but that objective knowledge (World 3) could nonetheless be represented physically in World 1, for example by being written on paper.

Critical rationalism is an epistemological framework for empirical theories: explicit falsifiable claims used to predict and analyse phenomena. Empirical theories are a kind of objective knowledge (World 3), but their correspondence with physical phenomena (World 1) is mediated by observational judgements (World 2). Popper was primarily concerned with scientific theories. A scientific theory is an empirical theory with qualities that include ontologically corresponding to the world, being as universal and as precise as possible, and being explanatory.

**B. 2×3 Worlds Schema for Engineering Theory**

The theories used in engineering are another kind of empirical theory. An earlier paper provides a rebuttal of the misconception that engineering is the same as science or applied science, and that engineering theories are the same as scientific theories. [8] Perhaps the most notable difference from science is that engineers deal with requirements. However, engineering theories also evaluated on different criteria than theories in science, as briefly discussed below in section II-C.

I previously defined [8] engineering theories as empirical theories used to reason about the performance of artefacts with respect to requirements. Engineering theories can be represented abstractly in logical terms, analogous to the deductive-nomological view of scientific theories. We can use the form:

$$\forall x. [E(x, a); D(a)] \Rightarrow R(x, a)$$

where $x$ are states of the world, including the artefact $a$, $R$ are requirements, $D$ is the design of the artefact, and $E$ are environmental conditions under which the requirements are expected to hold. The environment $E$ and requirements $R$ together characterise needs in the usage situation, but are separated here for clarity. The environment may for example impose conditions on the use of the artefact. The environment, designs, and requirements should be stated abstractly enough to cover a range of usage situations and artefacts, but should not be so abstract that they lose correspondence with the real world. As above, no amount of empirical test will completely justify the theory, and the observation of a single counter-example $\neg R(x, a)$ will deductively falsify the theory.

Critical of an engineering theory is achieved by a demonstration that the artefact does not meet its requirements. This is normal during an iterative design process, but if it occurs during the ultimate use of the artefact, then not only is the theory falsified, but there has also been an engineering failure.

The predicate above can be decomposed using *modus ponens*, for example into claims:

$$\forall x a . [E(x, a); D(a)] \Rightarrow B(x, a)$$

$$\forall x a . [E(x, a); D(a); B(x, a)] \Rightarrow R(x, a)$$

where the behaviour $B$ of the artefact $a$ is analysed separately to whether that behaviour satisfies requirements $R$. This allows lower-level theories of artefact performance to be reused for slightly different requirements in other usage situations.

I previously proposed a schema for objective engineering knowledge that represents artefacts, requirements for use, and their relationship by using two instances of Popper’s three worlds model. [8] This is depicted in Figure 1. Note that this is not a lifecycle model or process model, and does not imply a particular order in which designs or artefacts are created. (For example, in idealised requirements-led rational design, one might start from the top and work down to create a new artefact, whereas when reverse-engineering, one might start with the given artefact itself.) Instead, it is a model of relationships between worlds, and a model of correspondences that should obtain between these worlds.

1) Five Roles for Engineering Theory: In the $2 \times 3$ worlds schema, there are five relationships involving World 3 (objective knowledge), as shown in Figure 1. These relationships can be thought of as different *roles* played by engineering theories. Some theories may play more than one role, and some roles are filled by multiple theories in the engineering of a specific artefact. The roles are:
Fig. 1. The 2 × 3 worlds schema, from [8]. Highlighted are the five roles played by engineering theories.

1) **Requirements Validity**: An engineering theory may express requirements as specifications, and these should correspond with actual requirements for use in the real world. These requirements are determined by both judgements about what is needed, but are also conditioned by physical constraints in the real world. As an example, consider Vincenti’s [14] discussion of the specification of flying qualities in the development of early aircraft. The responsiveness of a plane to a pilot’s control is determined by the physical design of the plane while flying. However, judgements about flying qualities are made by pilots, and Vincenti discusses how specifications for flying qualities evolved over time, in response to failures of aircraft to exhibit good flying qualities.

2) **Requirements Decomposition**: Engineering theories may describe how high-level requirements are decomposed to lower-level requirements. For example, a requirement for safety (in general) might be decomposed to lower-level requirements for safety from an exhaustive range of potential kinds of hazards. For medical electrical equipment, technical standards such as the IEC 60601 family [15] provide this kind of engineering theory. These standards describe a range of specific properties including electrical, mechanical, thermal, electromagnetic specifications that should be met for the equipment to be considered safe.

3) **Operational Principles**: How are requirements satisfied by a design? Using the terminology of Polanyi, an operational principle is how an artefact’s “. . . characteristic parts—its organs—fulfil their special function in combining to an overall operation which achieves the purpose”. [16, p. 328] The development of new operational principles is the cornerstone of technological advance. For example, the fundamental operational principle for fixed-wing flight is for the curved surface of a wing to generate more lift than the weight of the plane.

4) **Design Decomposition**: Designs are not monolithic: they are expressed at many levels of abstraction, with design elements at each level composed in various ways to achieve particular kinds of artefact performance. Engineering theories can codify these design rules as they apply to a broad class of artefacts, or may be developed to apply to specific kinds of systems.

5) **Design Validity**: At the lowest level of a design, the behaviour of design elements should correspond with their performance as built. At low levels, engineers often rely on technical data sheets about component or material properties. However, engineers may also conduct empirical tests to confirm or determine component performance at higher levels of the artefact’s design.

C. **Qualities of Engineering Theory**

Engineering theories, like scientific theories, are empirical and should stand in valid correspondence with observations of the real world. However, the primary goal of engineering is to change the world rather than to understand it. [17] Perhaps the main difference is that engineering theories concern requirements. However, engineering theories are also judged on different terms than scientific theories. Some of these are discussed below.

1) **Phenomenological**: The main goal of science is to understand the physical world, and so scientific theories should be fundamental and explanatory. In contrast, to support an effective change to the world, it can be sufficient for engineering theories to be phenomenological (sometimes called ‘instrumental’), i.e. to merely predict phenomena concerning the performance of artefacts. One clear example of such theories are the data tables of aerodynamic performance of a parametric space of wing sections discussed by Vincenti [14]. These data tables had no explanatory power, but nonetheless were sufficient to inform the selection of designs of wing shapes in the design of early aeroplanes. If and when ‘deeper’ theories are developed in the sciences or in engineering research, they may supplant phenomenological theories. However, Vincenti argues [14, p. 193] that sometimes there is no demand from engineering practice for explanatory theories.

2) **Broader Class of Variables of Interest**: In addition to physical phenomena such as space, time, matter and energy, engineering also analyzes artefacts with respect to socio-technical phenomena that are not part of science. [18] These can include the cost to build, design and operate artefacts, and the safety or usability of artefacts in operation. Engineering
requirements can define constraints on all of these qualities, and engineering theories can be stated to support reasoning about them.

The cost and time required to design and build artefacts is certainly part of the concern of engineering. There are typically trade-offs in design between cost, time, quality, and the level of functionality offered by engineered artefacts. The constraints for these are given by the artefact’s requirements and the project context. Engineers often make these trade-off decisions, or at least characterise them for other decision makers. Because of these trade-offs, there can be an interplay between designing and the organisation of work on engineering projects. So, the engineering body of knowledge includes knowledge about about work practices and project management.

3) Narrower Scope of Application: Scientists aspire for their theories to be as general as possible, but in engineering, it can be sufficient to use a highly specialised theory that can only be used in highly limited circumstances. In particular, engineers sometimes develop theories about the performance of a class of artefact designs (e.g. data table theories of the performance of a limited range of wing sections, as discussed above [14]) or materials (e.g. theories of spring performance that apply only to materials within a limited elastic range [19]). Narrow theories may be developed de novo through empirical test on a limited range of example artefacts using parameter variation. Or, narrow theories may be developed as limited forms of general theories, where assumptions about environmental conditions or designs may be factored into theories to improve their cost or ease of use.

4) Might be Less Precise: Scientists create theories that are as precise as possible, but engineering theories sometimes only provide conservative approximations of artefact performance. An example are safety factors used in the design and analysis of artefacts. [20] There are a variety of reasons for using less precise theories: it can be much easier to empirically validate such theories; and it may be less difficult or costly to perform calculations with a less precise theory. If an engineer is faced with strict and demanding requirements, then highly precise theories may be needed. Often weaker requirements are acceptable, and less precise theories may be used. In engineering, more precise theories might be preferable, but only if they are needed to support analysis of important requirements, or if they are no less general or easy to use than other less precise theories.

Another way in which empirical theories may be less precise is by being probabilistic approximations. This complicates the idea that a single counter-example will falsify a theory, because it may only be one of the few random exceptions allowed by the theory. In principle this might be dealt with by stating empirical theories in probabilistic terms, and by constructing severe statistical tests over sets of observations.

5) Instrumental Properties: Engineering theories are used as instruments to reason about whether artefacts achieve the goal of meeting given requirements. Some theories may be easier to use than others (require less expertise, or give rise to fewer mistakes in calculations), or may take less time to use in calculations. These are the ‘instrumental properties’ of a theory. There are typically trade-offs between instrumental properties and other theory qualities. For example a more general or more precise theory is typically harder to apply.

6) Multiple, Overlapping: An empirical theory must be rejected if it has been falsified. For fundamental scientific theories, imprecision and lack of generality are also ‘errors’ that should be eliminated. (Imprecise or narrow theories are not necessarily empirically false, but merely do not satisfy the epistemological goal of science, which is to create fundamental explanatory theories.) In engineering, theories support assurances about the use of artefacts, and so should not be empirically false. However, engineering theories are used instrumentally, and need only be general and precise enough to reason about whether an artefact meets acceptable requirements for use. For this reason, imprecise or narrow theories need not be eliminated in engineering, and so the body of engineering knowledge consists of many overlapping, narrow theories.

As discussed in this section, there can be benefits in using less precise or more narrow theories. For example they may be easier to validate, or may be easier or less costly to use. This creates trade-offs in the selection and use of engineering theories. An unfalsified engineering theory that continues to be relevant to artefact requirements is only likely to drop from the body of engineering knowledge if it is dominated across the range of criteria by competing theories.

It might be argued that some scientific fields also maintain a body of multiple, overlapping theories, with different scope and precision, [21] and thus that engineering theories are not different to scientific theories in this way. However, the goal of science is to understand of the world, which forces scientists to seek the most general theory, the most precise theory, and the theory with a fundamental correspondence with structure of the world. This drives science towards the ideal of establishing a single theory within each field. Engineering does not have the same goal as science, and its theories are judged on different criteria, and as noted earlier sometimes there is no demand from engineering practice for explanatory theories [14, p. 193]. So, although some fields of science and engineering may be similar in practice in maintaining a multitude of overlapping theories with different scope and precision, nonetheless the ideals for scientific theories are not necessarily a strong preference in engineering.

III. SOFTWARE ENGINEERING THEORIES

The previous section provides background on an ontological framework for theories in engineering. Does the framework apply to software engineering? Indeed, can we have empirical theories about software at all? Software is sometimes argued to be purely intangible, mathematically abstract, and not subject to empirical investigation.

Popper [13] identified software as a kind of objective knowledge in World 3. As with other kinds of objective knowledge, software can be represented physically in many different ways
without changing its essential character or identity. A program is ‘the same’ regardless of whether it is printed on paper or encoded in a binary representation on a magnetic hard disk. In that sense, software is intangible, because its identity does not depend on how it is physically represented. Nonetheless, when software code is represented in the magnetic or electronic media in a physical computer, it can cause that computer to behave in a certain way. Software-based systems are part of the physical world, World 1. The execution of a computer uses electricity, generates heat, controls devices, and transmits signals to other systems. These are physical phenomena. The functional behaviour of these physical systems can be predicted by analysis using empirical theories. Theories about the performance of computer hardware can be important, especially for requirements on power consumption or heat. However, to predict and analyse the functional behaviour of a computer system, empirical theories about executing software are vital. Software engineering theories such as programming language semantics and formal methods are not just mathematical. The controversy and lessons [22] from Fetzer’s [23] paper on program verification remind us that the execution of a program on a physical computer is categorically different to the objective content of the program. Ultimately theories in formal methods are also empirical theories when they are claimed to apply to the behaviour of real software-based systems. [8], [9]

Software testing is a common practice within software engineering. Software testing is itself partly driven by theory. White-box testing exploits structure within the program, and black-box testing explores the space of functionality defined by an interface specification. The idea underlying software testing is that the correct behaviour on test cases is representative of correct behaviour on other untested parts of the state space. The limitations of software testing are well understood in software engineering. [3] Software testing is not a very severe test of a program. Because of the huge size of the state space of a typical software-based system, and because of the discontinuous nature of the behaviour of digital computers, software testing is not usually a reliable way to corroborate the precise functional behaviour. Performance data tables are a kind of phenomenological theory based on testing and are widely used in other fields of engineering [14]. However, they rely on the assumption of valid interpolation between points in the parameter space. This is not usually a valid assumption for the prediction of precise functional behaviour of complex software-based systems.

A. Process Theories and Product Theories

The $2 \times 3$ worlds schema frames the ultimate purpose of engineering theory: ensuring that artefacts meet requirements for use. In other fields of engineering, the overwhelming majority of theories are directly concerned with the properties of artefacts, and properties of the materials or manufacturing systems used to construct artefacts. We may call these ‘product theories’, to distinguish them from ‘process theories’ covering the work practices and organisational methods in engineering projects. The main part of the body of engineering knowledge within a field of engineering are product theories, because they directly support assurances about whether artefacts meet major functional and non-functional requirements. Requirements on cost and time during design and development can be an exception, as discussed in section II-C. The cost and time required for a team of engineers to design and provide assurance about an artefact can be significant and is normally highly constrained. So, process theories guiding work organisation and project management can be important. There are also relations between process and product theories. For example, the work breakdown structure on a project is commonly driven by the structure of the design, and Conway’s ‘Law’ [24] talks about the relation between design structure and team structure.

In software engineering research there are similarly both product theories and process theories. Product theories typically arise in areas such as formal methods, software architecture, or systems research. Process theories often arise in areas such as development lifecycles, agile methods, and global software development.

Consider the Essence framework [7], which has arisen in response to the SEMAT call [6] and vision [25]. Essence is proposed as a general theory for software engineering structured around a collection of entities called ‘alphas’, and relationships between them. We can see some parallels between the Essence alphas and the elements of the $2 \times 3$ worlds ontology shown in Figure 1. The Essence ‘Opportunity’ and ‘Stakeholders’ alphas might together correspond with the ‘Usage Situation’ of the $2 \times 3$ world schema. And the Essence ‘Requirements’ and ‘Software System’ alphas might match ‘Requirements Specifications’ and ‘Artefact as Built’ respectively in the $2 \times 3$ worlds schema. However, the Essence model does not recognise as distinct entities the software itself, nor other kinds of system design. So Essence does not provide a space to discuss the structure of product theories. (Popper’s three worlds model would include the physical embodiment of the Essence alphas ‘Work’, ‘Team’, and ‘Way of Working’ in World 1, and theories about them in World 3. However, the way I have depicted the $2 \times 3$ world schema in Figure 1 does not neatly capture these key conceptual elements of process theory.) Project management and process management are important, and their failure can lead to project failure in practice. However, excellent project management does not itself provide a basis for assurance about whether a software-based system will meet substantive functional and non-functional requirements.

There has been a significant focus on process theories within software engineering research. Perhaps this has arisen because development cost and time have been so problematic in industry, or because product theories have been too immature or expensive to use. However, with an increasing demand for trustworthy software, product theories will become increasingly important. Process theories are a second-order concern for engineering design and assurance. As noted by a National Academy of Science report on software engineering for dependable systems, “process is essential for preserving
the chain of dependability evidence but is not per se evidence of dependability.” [3, p. 106] In contrast, product theories can provide such evidence for dependability.

B. Software Engineering Theories in the 2 × 3 Worlds Schema

In this section I briefly highlight some examples of software engineering theories that can play each of the five roles described in section II-B1. This provides face validity that the 2 × 3 worlds framework is consistent with the body of software engineering theory.

1) Requirements Validity: I would be remiss to not first note the growth in knowledge of requirements for software-based systems discussed by MacKenzie [22], concerning requirements for computer security. MacKenzie recounts the history of the early formalisation of security properties in the Bell-LaPadula model [26], and its challenge by McLean’s ‘System Z’ [27]. System Z was logically consistent with the Bell-LaPadula model, but had the undesirable practical consequence of allowing any user access any file. There were various responses to this counter-example from within the security community, ranging from denying that it was a valid example, to adjusting the concept of security to allow that it was acceptable. Ultimately, the specification of security was both re-conceptualised and reformulated in new requirements specifications for security. MacKenzie observes that this process resembles that of Proofs and Refutations for mathematics as described by Lakatos [28]. I have previously argued [9] that the growth of knowledge about requirements in engineering is also similar, and all fit the general tetradic problem solving scheme in critical rationalism.

So, within a specific community, the understanding and specification of acceptable requirement properties can evolve over time. However, software engineering research also develops more general techniques to define and validate requirements. For example, the requirements research community develops techniques for requirements elicitation and early-lifecycle requirements validation, the human-computer interaction research community develops methods for task analysis. Much of the value of early-lifecycle prototyping and even of iterative development is from their contribution to requirements validation.

2) Requirements Decomposition: In software engineering industrial practice, requirements decomposition is recorded and managed systematically using explicit requirements traceability matrices. Traceability in general can apply across all five of the roles discussed here, and provides the basis for coherent arguments that software-based systems meet their requirements. However, traceability is particularly important for requirements decomposition, and is often first adopted by software engineering organisations for that purpose.

The justification for requirements decomposition captured through traceability relationships is often not explicitly provided. However, in fields such as software safety, there are attempts to explicitly capture rules for goal decomposition which are often represented diagrammatically or in explicit argument structures. These rules are explicit theoretical claims and arguments about requirements decomposition, and include a place for capturing supporting evidence. [29]

Within the formal methods community, the decomposition of requirements specifications is well understood. Examples are reasoning about Z schema [30], [31], and specification refinement rules in the refinement calculus [32], [33], problem frames [34], [35], and behavior trees [36]. More recently, Rushby [37] provides a logical treatment of abstract requirement specifications in the context of safety case arguments. All of these approaches are or could in principle be formalised in deductive logic. (However, Rushby’s formalization simulates a defeasible logic which arguably makes those theories unfalsifiable, and thus would not be acceptable from the perspective of Critical Rationalism [8].)

3) Operational Principles: Linking requirements to designs is a critical part of the design activity. In software engineering, software architecture often plays this role. Software architecture researchers and practitioners are especially aware of the importance of non-functional requirements as drivers for architectural design decisions. [38]

Jackson [5] provides an extensive discussion about the connection between requirements and design in software engineering. He describes how explicit connections are made during design about how problem world (constrained by requirements), interface with the designed machine, which includes both hardware and software. This is all broadly consistent with the 2 × 3 worlds model discussed in section II-B.

There are also specific operational principles used in specialised areas within software engineering. One example is relational database design, which uses principles of organising data according to tabular mathematical relations [39] to meet requirements for flexible, structured, and fast access to database information. Another example is from real-time software systems, where scheduling schemes [40] are used as a key operational principles to achieve and guarantee timeliness requirements.

If requirements and design are all formalised in a single mathematical space, then operational principles can be represented as logical theories within that space. An example of this can be seen in program refinement in formal methods, for example as seen in the formal verification of isolation requirements for the seL4 microkernel [41], where the requirements specification, its properties, the high-level design, and the detailed design (semantics of the C implementation) are all expressed in higher-order logic.

4) Design Decomposition: In software engineering, the most common kind of design decomposition is programming. As discussed in section II-B, software programs are part of World 3 (objective content), and their creation and decomposition play the role of design decomposition in software engineering. There are many theories of program composition and refinement, for example structured programming [42], design patterns [43], and data refinement [44].

However, higher-level design decomposition is also important in software engineering, and is often carried out at the level of software architecture. Software architectural patterns
and styles [38] provide a space of rules for design and design decomposition, and justifications for choices between design options.

5) Design Validity: Do the low-level models of machine behaviour correspond with the behaviour of real executing computers? There are two ways to establish this correspondence: prescriptively (ensure that machines behave according to a given machine model), or descriptively (ensure that the machine model specifies the actual behaviour of the machine).

In formal methods, the validation of low-level machine models is perhaps not always well established. However, there are some good examples from the literature. One of the features of the ACL2 theorem prover is to be able to execute formal machine models quickly, and this has been used in the design-time analysis of commercial microprocessors. [45] During design, test cases were run in both the theorem prover and on an engineering test sample of the microprocessor. Problems with the formal model and problems with the physical hardware were both discovered. Modifications to either could restore the correspondence between them. A similar approach has been used in ACL2 to execute formal models of virtual machine software. [46]

Bishop et al. [47] have developed approaches for the detailed rigorous specification and conformance testing of communication protocols, and applied that to TCP, UDP and sockets. This revealed ambiguities in the traditional natural language specifications, and a number of irregularities and differences between implementations of the protocols. The authors note that because of this ambiguity, it is not clear from the traditional natural language specifications whether or not implementations are conformant. The authors also note that irregularities are hard to definitively classify as 'bugs' when major implementations have force as a de facto standard. In such cases, the formalisation may be more descriptive than prescriptive. A key benefit of a detailed and explicit formal theory is making it clear when an empirical observation counts as a counter-example, falsifying the theory.

These examples concern the validation of formal models of hardware or machine code, but it is important to note that the extra-logical leap from the formal world (World 3) to the physical world (World 1) can in principle be taken at any level of design abstraction. The formal verification of the seL4 microkernel [41] was first conducted with a formal model of the C programming language [48]. The seL4 formal verification was later extended down a level of design abstraction to apply to the level of machine code [49] using a formal model of the ARM instruction set [50]. The ARM model was in turn validated by systematic testing of the model against execution of real ARM computers. There is always a leap from the formal world to the physical world at some level of design abstraction, but the nature of required environmental assumptions changes at lower levels of design.

As an example at a high level of design abstraction, consider architectural theories of enterprise system performance [51]. These are not grounded in the formal modelling of a programming language or the detailed behaviour of a specific computer. Instead, their validity is established by parameter variation and systematic testing of enterprise systems and observing their correspondence with predictions of the models. As discussed in section II-C, there are trade-offs in choosing the most appropriate theory. Architectural models of enterprise system performance are simpler and easier to use than more detailed models of machine semantics, but only offer approximate predictions, and only for a narrow range of enterprise systems coded in conventional ways. However, these limitations may be completely acceptable for some software engineering purposes.

C. Qualities of Software Engineering Theories

In section III-C, I presented a list of some of the qualities seen in theories in engineering. Here I briefly note software engineering theories holding the same range of qualities.

1) Phenomenological: The theories of enterprise system performance [51] noted above have phenomenological elements. The performance of a whole distributed system is determined through an explanatory theory based on the interaction and performance of constituent components. However, the performance of individual components is determined through experiment. This part is a phenomenological theory, like the performance data tables generated through parameter variation discussed by Vincenti [14].

Data-driven cost estimation models [52], such as case-based reasoning, are another example of phenomenological theories used in software engineering. In order to estimate the cost or time required for a new software engineering project, these approaches use cost and time measurements from similar historical projects, and combine these to create estimates for the new project. Such theories have little explanatory power, but may nonetheless generate predictions of project cost and time that are reliable and accurate enough to use in practice.

2) Broader Class of Variables of Interest: As discussed above, software engineering theories may be concerned with predictions about time [51], [40] or functional behaviour [41]. However, software engineering theories can also address socio-technical phenomena such as safety (e.g. structuring safety cases [29], [37] or development cost (e.g. estimation models [52]).

3) Narrower Scope of Application: Drawing on ideas from sociology [53], [54], Stol and Fitzgerald [55] have called for a focus on 'middle-range' theories in software engineering, rather that highly general ones. The body of software engineering knowledge has many narrow and middle-range theories. [56]

Jackson [57] draws on literature on engineering practice and method, much as I have done. He argues that specialisation (especially by kind of artefact) is critical to achieving dependability in engineering and software engineering. As in other engineering disciplines, only by evolving a repertoire of normal designs and supporting theory for specialised kinds of software artefacts can software engineering mature enough to widely support high levels of dependability.
Nonetheless, arguably there are also some general theories. For example, Turing machines underlie computability theory and complexity theory, and process algebras provide general theories for concurrency. There are also initiatives to build unifying theories of programming for highly general theories of programming language semantics. [58] These general theories from theoretical computer science or mathematics abstract away from the details of real-world computer systems. To apply to concrete software-based systems they must be given an empirical interpretation, and must be specialised to achieve valid correspondence with the world.

4) **Might be Less Precise:** The theories of enterprise system performance mentioned earlier [51], [40] are approximate in a conventional way by giving numerical predictions with error ranges.

However, there are other kinds of approximate theory in software engineering. For example, the type system of a programming language is an approximation of the functional behaviour of programs within the language. In a type-safe language, all evaluations of a variable or expression might be guaranteed (within the formalism) to stay within the values of a specific type. The type theory does not say precisely which value will be seen, only that it will be one of the type-safe values. For a software-based system, using that programming language, the type theory becomes an empirical theory that provides an approximation of aspects of the behaviour of the system.

5) **Instrumental Properties:** For the adoption of a software engineering theory (or the adoption of its realisation as a tool or method) into practice, key concerns include its cost-effectiveness, and whether it is easy to understand and use. [59]

There can be widely different ranges of instrumental properties, which typically form a trade-off with other qualities of the theory. For example, for most programming languages, their type system is fully decidable, and types are typically calculated and checked in a time linear in the size of the program text. For a moderate-sized program this would take at most seconds. In contrast, with a full operational semantics of a Turing-complete programming language, the problem of proving arbitrary properties of the functional behaviour of arbitrary programs is undecidable. For a moderate-sized program and a subtle specification, formal verification could take many persons-years. These are perhaps two extremes of one instrumental property: the time required to use the theory. Of course, as noted above, the level of precision also differs for these two theories.

6) **Multiple, Overlapping:** The SEMAT call [6] decries the large number of disparate methods and theory in software engineering. However, I have argued previously [8] and in this paper that this situation is not unexpected or negative in an engineering discipline. Each of the multitude of overlapping theories is “best” in terms of a specific combination of qualities required by the problem situation. The scope of a theory should fit the environmental conditions in the usage situation, should predict the requirements of system in the usage situation to a sufficient level of precision, should be usable by the software engineers working on the system, and should be as inexpensive and quick to use as possible.

IV. **Discussion**

In this section I discuss some of the implications that the $2 \times 3$ worlds schema raises for software engineering research. I then acknowledge some of its limitations, and discuss related work.

A. **Implications for Software Engineering Research**

Perhaps the main lesson for software engineering research is that narrow theories are entirely legitimate. Researchers should not abandon work on narrow or middle-range theories in order to search for general software engineering theories. I am not arguing that we should reject general software engineering theories! Nevertheless, I would agree with Merton [53] who advocates developing “progressively more general conceptual schemes” built on established middle-level theories, rather than “suddenly revealing” general theories.

Other things being equal, a more general theory is better. However, other things are rarely equal—typically there are trade-offs between qualities of theories. For example a more general theory may have less precision, or be more complicated, costly, or time-consuming to use. Cartwright [60] argues that there is a trade-off between explanatory power and descriptive adequacy for empirical theories. Fundamental, explanatory theories separate the component causes of physical situations. They abstract away from the concrete complications that arise in specific situations in the real world, and cannot be taken at face value. To apply to specific situations, they rely on either hand-waving *ceteris paribus* conditions or essentially ad-hoc combination with other fundamental theories using ‘bridging’ theories. Cartwright observes in contrast that engineering focuses on descriptive adequacy at the expense of explanatory power, and Pirrle [61] expands on this to argue that engineering theories may be better at telling the truth than fundamental scientific theories.

Wieringa and Daneva [62] make a somewhat similar case for the importance of middle-range theories for software engineering, both for their better descriptive adequacy, and as a valid basis for incremental theory generalisation. Like me, they admit models as a kind of theory, but unlike my proposals, they argue that theories for software engineering should be considered to be scientific theories.

Some authors calling for general theories in software engineering often argue that software engineering research should look to scientific theories as exemplars. For example, Johnson and Esstedt [63] argue that software engineering theories, like scientific theories, should be comprehensive (general), precise, consistent (i.e. not logically inconsistent), and correct (i.e. validly correspond with the real world). However, I instead argue that engineering is not science, and that engineering theories are different to scientific theories. [8] The principal difference is that engineering theories concern requirements,
but engineering theories are also evaluated differently. Engineering theories are not necessarily comprehensive, and only need to be as precise as is required to address meaningful requirements. Of course, as with all empirical theories, engineering theories should be consistent and correct.

The other significant lessons for software engineering research are to be explicit about the claims, scope, and assumptions of theories. Only when these are explicit can we clearly know when a theory is falsified, and learn which parts of the theory are critical to achieving a valid correspondence with the physical world. The framework reinforces the importance of traceability (and similarly, assurance cases) in software engineering, because they express explicit coherent engineering arguments about requirements and designs.

Researchers must also subject their theories to severe test against physical software-based systems, and report on this to provide evidence that the theories are well corroborated. These lessons are largely consistent with approach proposed in the National Academy of Sciences report on software engineering for dependable systems [3], to make explicit claims and assumptions, and to provide evidence that directly supports those claims. Dijkstra famously said “Program testing can be used to show the presence of bugs, but never to show their absence!” [64, p. 7] For formal models of non-trivial programs (in World 3), Dijkstra is correct. However, Popper might counter that nothing can show the absence of ‘bugs’ in a physical computer system (in World 1)—we can only ever tentatively hold to empirical theories, and the only way to corroborate an empirical theory is to subject it to severe test.

B. Limitations

Critical rationalism has limitations as an epistemological philosophy, and these are inherited by my use of critical rationalism to understand engineering knowledge.

One objection is that neither scientific nor engineering theories are in fact axiomatised in formal logic. The answer to this objection is that the use of formal logic for empirical theories plays the same abstract role as it does in mathematics. We say that one could in principle axiomatise all mathematical theories and construct proofs using explicit proof steps in a formal logic. However, in practice, mathematical theories and proofs are usually given in a less rigorous manner, with the understanding that details could in principle be provided on demand. Just so with scientific and engineering theories. Ironically, in software engineering, some theories (in formal methods, for the specification and behaviour of software-based systems) are explicitly axiomatised in formal logic, often with proofs provided in exhaustive detail (typically in a mechanised theorem prover). Providing a highly detailed logical formulation does not mean the theory cannot also be an empirical theory.

Another concern may be that engineering theories are sometimes not expressed mathematically, but are instead expressed diagrammatically or in drawings. [65] Schematic diagrams are also common in software engineering. The answer to this objection is that such drawings and diagrams can be given a formal interpretation and so be considered to be mathematical or logical theories, and can be given an explicit empirical interpretation and so be considered to be empirical theories. The structured interpretation of diagrams is well understood in software engineering.

There are other more serious limitations, two of which are discussed below.

A falsified theory should be rejected, but how much confidence should we attach to an as-yet-unfalsified theory? Popper rejected the possibility that specific concrete tests could provide complete justification for a universal empirical theory. Popper said that theories should be subject to ‘severe test’ and that as theories survived such tests, they became better ‘corroborated’. However, this has not yet provided a satisfactory account of positive justification or of the verisimilitude (‘truth-likeness’) of theories. Engineers may be called upon to provide different levels or strengths of evidence to support different levels of requirements criticality. How can one theory be more justified than another, and can we quantify that? Critical rationalism does not answer this well.

Critical rationalism is a model of the growth of objective knowledge. However it is well known [14], [65], [66] that there are other kinds of engineering knowledge. For example, Vincenti [14, p. 208] identifies two kinds of tacit knowledge: practical considerations, and design instrumentalities (including design know-how). If procedural or design knowledge is codified and claimed to improve or influence the suitability of artefacts to meet requirements, then it would also qualify as an engineering theory as discussed here. Otherwise, it is not addressed by the framework discussed in this paper.

Beyond tacit knowledge, there are also other intangibles which are not captured by this framework. As noted in section III-A, the $2 \times 3$ worlds ontology could in principle represent conceptual elements of process theories, such as engineering teams, ways or working, or work itself, as they are modelled in the Essence framework. [7] However, the logical formulation of engineering theories I have presented does not neatly represent these elements. Beyond this, cultural elements in the engineering organisation and among their customers and users are not recognised explicitly in either the $2 \times 3$ worlds framework or the Essence framework. [7] The National Academy of Sciences report on software engineering for dependable systems [3, p. 105] notes the critical importance of safety culture in achieving more reliable software than one might expect given the state of software engineering theory. Nonetheless, although culture is important, it is neither necessary nor sufficient for a defensible argument that software is dependable.

C. Related Work

Here I briefly compare the $2 \times 3$ worlds schema with three other meta-theoretic schemes: the situated Function-Behavior-Structure (FBS) ontology of design [67], the Research Path Schema (RPS) [55], and the Problem-Oriented Software Engineering (POSE) [68] approach.
FBS is an ontology of design in general, and has been used to discuss design in software engineering [69]. The FBS ontology is based on three classes of variables: Function (the purpose of the designed artefact), Behaviour (derived or expected properties of the artefact), and Structure (the components of the design and their relationships). In the $2 \times 3$ worlds schema of Figure 1, Function is the requirements specification, while both Behaviour and Structure are captured in the design specification. The situated FBS ontology views the three variables through three worlds, but these worlds are different to Popper’s three worlds. The FBS worlds are: the external world, the interpreted world, and the expected world.

The FBS ‘external world’ is “composed of representations outside the designer” [67, p. 377]. In Popper’s ontology, these are World 3 objects represented in World 1. For a designed artefact, the external world includes the written blueprint for the artefact, but not the artefact itself. The FBS ontology does not recognise the physical artefacts nor the physical use of those artefacts. In contrast, the $2 \times 3$ worlds schema does recognise the physical world. Requirements are not just desires—as Vincenti’s [14] example of flying qualities shows, requirements are also modulated by the physical reality of usage situations. The behaviour of artefacts is not necessarily predicted from the design by tentatively-held engineering theories—those theories may be wrong. Engineering design is based on theories that must correspond with the physical world. Both the FBS interpreted world and expected world are part of Popper’s World 2. The expected world is located within the interpreted world, but is distinguished for methodological purposes. FBS thus explicitly distinguishes between desired (expected) behaviour, and behaviour that is predicted from the design structure. In contrast, in the $2 \times 3$ worlds schema, expected properties arise from the translation of the requirements to the design level, and behaviour predicted from the design structure is calculated from the design, using engineering theories. If the predicted behaviour does not satisfy the required behaviour, then overall the argument will not be logically sound.

The Research Path Schema (RPS) [55] is a model of the role of theory and conceptualisation, especially in software engineering research. The RPS consists of a number of ‘domains’ which represent elements of research or theorizing. The RPS model is somewhat similar to Popper’s three worlds ontology: the RPS ‘substantive domain’ is like World 1, and the ‘conceptual domain’ is like World 3. However, Popper’s World 2 has no direct correspondence in the RPS framework, and Stol and Fitzgerald’s ‘methodological domain’ has no direct correspondence in the three worlds ontology. Methodology can be discussed using Popper’s three worlds model, but methodology is not the topic of the model. When compared to the $2 \times 3$ worlds ontological model used in this paper [8], the distinction between requirements and artefacts is not made in the RPS schema, although these could perhaps be distinguished as constructs in distinct instances of their domains. Overall, the RPS provides (as its name would suggest) good support for modelling the research process, but is perhaps less well suited to modelling the structure of engineering theories.

The Problem-Oriented Software Engineering (POSE) [68] approach is largely consistent with the philosophy of engineering discussed in this paper. If we factor out the stakeholder-relativization of POSE problem transformation rules, their schema can be considered to be isomorphic with the logical formulation of engineering theories discussed above. POSE requirements are as requirements $R$, POSE solution domain as are designs $D$, and other domains in the POSE context are as environments $E$. Specific POSE transformation rules are then logical rules representing specific engineering theories. The POSE authors also demonstrate nicely [70] how an assurance argument can be structured according to the logical proof constructed when using software engineering theory to reason about whether a software-based systems will meet its requirements.

V. CONCLUSIONS

Software engineering is, or is becoming, an engineering discipline. Increasing the maturity of software engineering will need increasing awareness of the importance and purpose of theory [55], and will need continuing improvements in the quality of software engineering theories.

Drawing on recent work in the philosophy of engineering [8], [9] I have argued that the importance of software engineering theories is to support rational justification for assurances that software-based systems meet their requirements. The most important quality of any empirical theory is that it be valid, or at least well-corroborated and not yet falsified. An invalid empirical theory is of no use in reasoning about the world. In engineering, empirical theories should also address real requirements and cover designs for real artefacts. A valid empirical theory that does not address the kinds of requirements or designs encountered in real systems is again of no use to practising engineers.

Engineering is not science. Engineering theories concern requirements for the use of artefacts, and may concern socio-technical requirements (such as cost, or safety) which are not the direct concern of science. The qualities of engineering theories are evaluated differently than scientific theories. Theories in software engineering, like theories in other engineering disciplines, may legitimately be phenomenological (and so not be fundamental, or have explanatory power), may be approximate (and only be precise enough to reason about acceptable requirements for an artefact), and may have a limited scope (but wide enough to cover acceptable requirements, environmental conditions for use, and designs for artefacts).

This view of engineering theories and software engineering has implications for software engineering research. Two of these are key methodological lessons. First, theories are a kind of objective knowledge, and so researchers must be explicit in the formulation of the theories, and especially of their scope of applicability and of their claims, including their claimed precision. Second, software engineering theories are empirical theories, and must be subjected to severe empirical
test, and be reported as explicit evidence to corroborate the theory. These recommendations echo those from the National Academy of Sciences report on assurance of dependable software systems [3].

From the perspective of critical rationalism, the growth of knowledge about empirical theories is an unending quest. Because we can never know if an empirical theory is completely true, we can only ever tentatively hold it, and attempt to falsify it with ever more severe empirical tests. Other things being equal a general theory is better than a narrow theory, but as Cartwright argues [60], greater explanatory power can be in tension with descriptive adequacy, and so make theories less suitable for engineering purposes. For software engineering, a quest to create general, explanatory, descriptively adequate theories may be at best premature, and at worst quixotic.

ACKNOWLEDGMENT

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

REFERENCES
