Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 44 (2015) 433 – 442

Procedia Computer Science

2015 Conference on Systems Engineering Research

An Evolutionary Theory-Systems Approach to a Science of the Ilities

Ke Dou, Xi Wang, Chong Tang, Adam Ross (MIT), Kevin Sullivan

University of Virginia, 85 Engineer's Way, Charlottesville, Virginia 22904-4740, USA and MIT

Abstract

For system engineers to effectively document requirements for non-functional properties (ilities), to reason about tradeoffs, and to implement and verify such properties, the language used in these endeavors must be precise enough to support rigorous engineering activities. Yet the language used today is often ambiguous and imprecise. Moreover, many past attempts to improve it with natural language definitions and explanations have not converged. We propose an *embedded theory-systems* (ETS) alternative approach. It replaces natural language with theories (models) in an expressive formal language; mechanically derives software from these models to foster community engagement with the theories; and uses feedback based on interactions with the software to drive theory evolution and validation. We hypothesize that this approach can accelerate convergence on models that are precise and validated enough for rigorous systems engineering. We present an early case study on applying this method to the Ross et al. *semantic* approach to defining change-related ility terms. Results include a clarifying formalization of their informal model, its evolution through four stages of feedback, insights into key remaining shortcomings, and evidence that the approach can promote engagement with theories in ways that drive convergence toward shared, precise, useful language for engineering system ilities.

Keywords:
systems engineering, non-functional properties, ilities, tradeoffs, synthesis, web services, evolution

1. Problem

The system shall be adaptable enough to meet new requirements. It shall be resilient enough to continue operating acceptably under unexpected conditions. It shall be easy to use. It shall be scalable.

These and other similarly vague statements about non-functional systems properties (*ilities*) plague systems engineering today. These statements reflect deeply important concerns, but in ways that are easily misunderstood, not subject to reliable validation, and that cannot support rigorous reasoning, design, and verification. Yet engineering critical non-functional properties is among the most demanding systems engineering challenges.

The underlying problem is that we lack foundational scientific and engineering knowledge, and *corresponding language,* needed to manage the broad range of non-functional system properties and tradeoffs for complex systems. These properties include changeability, affordability, dependability, usability, resilience, and many more. Gaps in precise, shared understanding of ilities and how they interact in specific environments make it hard to communicate unambiguously about them. The results are seen in costly project and operational failures, major down-side surprises late in development, and unacceptable risks, costs, and difficulties in developing and certifying critical systems.

Email address: {kdou, xw4bt, ctang, sullivan}@virginia.edu; adamross@mit.edu (Ke Dou, Xi Wang, Chong Tang, Adam Ross (MIT), Kevin Sullivan)

1877-0509 © 2015 Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Stevens Institute of Technology.
doi:10.1016/j.procs.2015.03.064

2. Why Ility Engineering is Hard

Several factors combine to make engineering of system ilities and tradeoffs a demanding challenge. First, the success of any complex system depends on the realization of a broad range of ilities important to stakeholders. In addition to transforming information, energy, and matter in desired ways (functional properties), success depends on *how well* systems carry out these transformations, i.e., on additional quality attributes, e.g., how reliably and safely, with what possibilities for affordable change, with what endurance, with what resiliency to off-nominal conditions, with what protection against subversion, at what cost, what what ease of use, etc.

Second, many ilities are hard to characterize in ways that are precise enough to support rigorous engineering. We lack modeling frameworks, definitions, and languages in which to communicate effectively about these properties and tradeoffs. Different terms are often used to refer to the same property, and the same term to different properties [9]. Most natural language definitions of the meanings of ility terms are too ambiguous to be useful for rigorous engineering use, in any case. Most crucially, the properties themselves are often understood at best vaguely. The problem is thus not only syntactic, but semantic, and ultimately ontological.

Third, while ilities must be defined, understood, and engineered as *objectively measurable system properties* (where the system boundary is properly circumscribed), the *value* (broadly construed) of given ilities, and in some cases even the set of ilities one wishes to define and measure, are ultimately subjective and stakeholder-specific. In large-scale systems, the value propositions of individual stakeholders are thus often complex functions of multiple ilities, some of which have definitions that are shared across an entire project (e.g., system reliability with respect to overall mission failure), while others may be role-specific (e.g., the remaining risk of a cost overrun under a particular project manager) or idiosyncratic (e.g., that a particular set of users find a particular form of interaction to be highly *usable*).

Fourth, stakeholder ility-to-value functions are also generally time-varying, and in unpredictable ways. At a minimum, stakeholders tend to learn as they gain experience with systems. This learning often leads to shifts in understanding about what is needed and what it will cost, producing changes in how one values properties and functionalities. The likelihood that ility valuations change as a system is developed and used implies the need to engineer systems with architectures and implementation that provides both a complex set of ilities and an envelope within which ilities can be varied readily to ensure that all success-critical stakeholders remain satisfied to an acceptable degree [2].

Fifth, the implementations of many ilities crosscut many system components, including frequently the *meta-systems* (comprising the people, materials, tools, processes, etc) responsible for constructing the actual systems of interest. Unlike functional features, ilities are generally hard or impossible to modularize. Indeed, achieving some ilities often requires design of both system and meta-system components and processes. For example, reliability is famously achieved in part by the careful measurement, control, and improvement of design and manufacturing processes.

The crosscutting nature of the ilities has numerous important consequences. One is that non-functional properties are often hard to achieve, because they require system-wide coordination and consistency. For example, tight tolerances in system performance usually require even tighter tolerances in the performance of all of the lower-level components that the top-level service uses. A second consequence is that it is usually hard, and often impractical, to add demanding ilities once systems are built, because such changes require uprooting of many deeply embedded decisions across the system design. For example, it would be highly impractical to add new security properties to the Internet protocol, e.g., as might be used to unambiguously identify attackers.

Sixth, the envelopes within which ilities will be subject to variation at reasonable cost must often be planned from the early stages of system and meta-system design, but we lack adequate theories and models for reasoning about what major early decisions are consistent with the delivery of given sets of ilities the users of a system will require.

Finally, just as the design impact of any given ility is typically scattered across a system design, so are the impacts of multiple ilities typically tangled with individual system or meta-system design decisions; and consequently, *changing* any major design decision typically impacts multiple ilities, some positively and others negatively. In a manner of speaking, the partial derivatives of the ilities with respect to individual design decisions can be extraordinarily complex [11], and controlling ility outcome states by manipulating controllable design decisions becomes a major challenge.

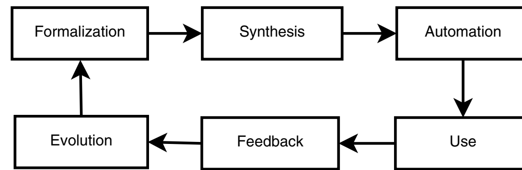

Figure 1: The embedded theory-system fitness evolution loop.

3. Contribution

Not only do we lack adequate models in terms of which to represent many ilities, but we even lack effective approaches to developing and validating truly effective models. The main contribution of this paper is a new approach to development, evolution, and the eventual validation of such models for systems engineering, and some evidence, in the form of a case study, that the approach is promising. Our approach is not meant as a solution to the broad range of complexities involved in ility engineering, as discussed in the preceding section, but as one key step to having stronger scientific foundations and tools in this domain.

4. Approach

We propose an alternative to the tradition of trying to impose natural language or quasi-formal definitions of various ility terms on diverse engineering communities. By contrast, ours is an *embedded and evolutionary theory-systems* approach to building precise and validated ility models and languages. Figure 1 illustrates our approach. We replace (or augment) the use of natural language definitions with mathematically precise, abstract, formal theories (models) presented in a highly expressive, formal (mathematical, computational, logical) notation. For the work reported in this paper, we have used the higher-order logic and pure functional programming language of the Coq proof assistant [3]. We then synthesize software from such models, using Coq's extraction function. We embed this extracted code in, in order to expose its abstractions as, REST web services [4], and we build clients to allow easy access to these services, thus automating the theory. The goal is to foster community engagement with theories by way of interaction with their corresponding services. We then use feedback based on interactions with the software to drive theory evolution and validation. We envision a whole ecosystem of such interoperable models as a basis for an evolving science of and technology base for engineering ilities, tradeoffs, and affordability.

From a scientific perspective, the insistence on mathematical precision has many benefits. First, it avoids ambiguity, accidental incompleteness, and inconsistency in ostensibly rigorous models. Second, it charts a path to a properly mathematical formulation of a science of ilities, in which models are abstract, generalized, and subject to rigorous test and evaluation. Third, formalizing concepts in a computational logic helps one to think much more clearly about how to formulate a model, in part because it immediately connects to a very large body of computational and mathematical knowledge. Concepts from algebra, graph theory, and other areas of mathematics and computer science can be leveraged to define concise and elegant models. In one case study, presented later in this paper, we formalized a model, that had been presented informally as a kind of spreadsheet, in terms of context free languages and theories of inductive data types and semantics of programs.

The synthesis step creates a tightly coupled, and assuredly consistent, theory-system pair that make the essential abstractions embodied in the theory accessible to the research and practitioner communities in a form amenable to interactive exploration, use, and evaluation. The software synthesis approach greatly facilitates theory-system evolution insofar as updates are made mainly to the abstract theory, with changes to implementations being mainly a matter of re-generation of code from updated theory specifications.

The result is closed *theory-system evolution loop* that passes through the intended community of use—a central feature of the approach. The approach is at once formal and theoretical, in ways that are essential for constituting scientific foundations for engineering practice (but that practitioners might find difficult and unnecessary to comprehend); and practical, in the sense that the approach projects formalisms into dynamic software tools, thus embedding theories into the community of use in highly accessible forms. Community engagement with the theories though interactions with derived tools—which we realize in the form of web services and interactive web-based client tools—fosters user

exploration, questions, insights, and feedback. That feedback, in turn, drives theory evolution, which in turn triggers synthesis of updated software services and tools.

This method bootstraps a generative recursion that—we propose—tends to drives both theory and tools to higher and higher states of fitness for purpose, ultimately producing consensus theories, definitions, and reference tool implementations. That is, we believe that this approach—if scaled up adequately—has the potential catalyze the development, validation, and value-driven evolution of precise, formal, abstract, software-supported, and broadly useful set of models and languages for engineering system ilities and tradeoffs.

In sum, our approach has four basic parts. First, we use formal (mathematical, computational, logical) methods to define *theories* (formal models) in terms of which ility requirements can be expressed. Second, from such models we synthesize software to create working *systems* that researchers and practitioners can use to explore, understand, critique, drive the evolution of, and eventually validate and use these models in practice. Third, we package these coupled theory-system complexes in the form *web services* and *clients* to make the theories and the systems that implement them accessible to the research and engineering community. Fourth, based on feedback from the community, we drive the evolution of these coupled theory-system complexes to states of greater fitness.

5. Case Study

To illustrate our approach, as a proof of concept, and as an early data point suggesting that the approach can satisfy our goals, we describe a case study in which we applied it to formalize, automate, disseminate, gather feedback about, and evolve (through a few improvement steps) an informal theory proposed by Ross et al. [9]. They presented their informal theory in a working paper entitled, "A Prescriptive Semantic Basis for System Lifecycle Properties."

This paper and the quasi-formal theory it presents addresses the same basic problem that we address here. Its main aim was to find a better way to give a range of change-related ility terms—e.g., evolvability, flexibility, and adaptability—clear meaning. We thus share with Ross et al. concerns about confusion in the definition and use of such terms, and about the poor record of success in earlier attempts to drive convergence toward better definitions. We credit Ross et al. with having emphasized these issues in their earlier works.

Where we differ is in our manner of theory development, presentation, evolution, and validation. Whereas Ross et al. presented their theory in the form of a spreadsheet table explained in a prose report, our approach was to formalize the core concepts in Coq, using lessons learned from theory of programming languages, and then to implement our *embedded theory-systems evolution loop* process to drive understanding, evaluation, critique, and evolution of the theory toward a state of greater fitness for purpose. For this case study, Ross, his students, and we ourselves were the community of use that interacted with the system in order to understand, to evaluate, to critique, and to drive the evolution of the theory and derived software service and tool.

5.1. The Ross et al. Model

Ross et al. [9] presented a promising idea: Rather than trying to impose definitions of ility terms top-down (which has not worked), one can instead (1) build a template that encompasses a comprehensive range of statements of ility requirements, and then (2) classify expressions constructed from this template by labeling them with ility terms that apply. Their template has variable elements for change agents, circumstances that trigger the need for change, and numerous others. Some change-related requirements are then be labeled as *evolvability* requirements, others as *flexibility* requirements, and others as *adaptability* requirements, etc., based on the particular ways in which they are instantiated from the template. Classes are distinguished by such factors as whether *the agent* responsible for making a change (a template element) is *inside or outside* the system boundary (another template element), for example.

The ility classes are not necessarily exclusive: a given statement can belong to several classes. The key idea was that instead of trying to write *ex cathedra* pronouncements defining ility terms, one should articulate the *semantic domains* of meaningful statements of (change-related) requirements, and then attach single-word ility labels to subclasses of terms in this semantic domain. Ility terms such as *flexibility* and *adaptability* are thus distinguished and given precise meaning, as specific kinds changeability.

The motivation of Ross et al. is valid, and the approach presented in their paper is attractive. However, it remained informal and paper-based and thus not computable and therefore hard to understand in precise detail, hard to evaluate, and hard to evolve when found wanting in some way. We decided to take their theory as an initial informal expression that we would subject to our approach.

5.2. Applying Our Approach

We applied our approach in accordance with the steps outlined above.

The first step was formalization. We used the higher-order constructive logic of Coq as a language in which to express a formalized and transformed version of Ross's model. Such a formalization is not a mere translation of informal content into formal content, but also incorporates insights gained in parsing the informal theory carefully enough to see how best to formalize it. In this case, the key insight was that Ross et al. had essentially defined a context free language for changeability requirements statements and what one could view as a very simple *type assignment* operator for terms in this language. Coq is especially well suited to support the specification of such languages and semantics, including type systems.

The second step involved synthesis. From our formalized model of the essential content of the Ross et al. model, we extracted code in Haskell, a lazy functional programming language with a strong and expressive static type system [5]. The extracted code implements the content of our specification: essentially an inductively defined data type, values of which are terms in our formalized version of Ross's changeability requirements language, and a simple type assignment function taking such terms and returning lists of corresponding ility labels (values of another data type in our theory).

The third step was automation. We integrated the extracted code into a reusable framework that we developed, based on Haskell's Yesod [13] web service framework, to expose the functionality of our extracted code as a REST web service. Yesod supports development of type-safe, RESTful, high performance web service applications. The main functionality of the specific service we built for this study is to take strings representing requirements expressions in our language, to run the type assignment function, and to return the lists of corresponding ility terms.

With this core functionality exposed as a web service, we then manually crafted a simple web-based user interface using HTML, Javascript, CSS, and other standard tools. Figure 2 presents a graphical depiction of one version of the web interface. The hierarchical structure of the Ross et al. template is visible in the bold-faced materials in the upper left of the window. The pull down menus (such as *Perturbation_disturbance*) allow the user to select values for fields in the template. And the text entry boxes to the right allow for the entry of additional, domain-specific parameters into expressions. Pushing the *Generate* button extracts the data from the form and converts it into string representing a change-related requirements statement, with the resulting string displayed in the *Statement* text box. Pushing the *Send* button dispatches this statement to the REST web service, which computes its ility types, and returns them as a list, which is displayed in the *Result* text box.

Our web site presents four such interfaces, one for each in a sequence of four incrementally evolved versions of our theory-system. In addition to the form described here, each web page presents (1) links to the original informal paper of Ross et al. and to the current version of the formal theory in Coq (as Coq source code and in a pretty-printed HTML form), (2) a graphical interface for instantiating the Ross template and for generating and submitting statements to the current version of the back-end web service for type assignment, (3) a text box for displaying the results (4) and documentation to explain both the theory and the use of the tool to the end user.

The fourth step involving using the tools to understand, explore, and critique the theory. We ourselves formed the community of use for this study. The UVa co-authors did not initially understand many aspects of the Ross et al. model. The availability of a web-based tool made it far easier to explore the model and to formulate, discuss, and get answers to important questions about it. Ross and his graduate students accessed the tool in order to help answer our questions. Its availability provided a basis for productive, ongoing discussion of the model, shortcomings, possible improvements, etc. Without the benefits of formal precision and automation, we (at UVa) would simply not have known what questions to ask, or how to benefit from answers. Having a tool that make the underlying service and theory dynamic and accessible transformed our ability to make progress.

The fifth step was to collect feedback from user (self) engagement with the theory by way of the tool and service. We collected feedback from our own experience with the tool and from conversations in our group. As we were to be the recipients of our own feedback, we have not yet produced mechanisms for providing feedback to us from remote users. We have not yet involved external users in a case study, and that fact constitutes a basis for being cautious about our conclusions. The reason that we did not yet try to involve outside, e.g., industrial, users is that model is simply not well enough developed. The community of use was thus one of (self-interested) researchers. Mitigating this threat to validity, the UVa team was not part group that developed the original informal model. The UVa group thus did serve to some extent as an outside group engaging with what was genuinely an unfamiliar and not very well understood theory in this experiment.

Perturbation		Perturbation_disturbance	low temperature
Context		Context_circumstantial	late at night
Phase		Phase_ops	
Agent		Agent_external	Enter the description of the option you choose
Impetus	**Direction**	Direction_increase	
	Parameter	Parameter_level	knob angle
	Origin	Origin_one	Enter the description of your choice
	Destination	Destination_many	Enter the description of the option you choose
	Aspect	Aspect_form	
Mech	**Mechanism**	Mechanism_description	turning the knob
Effect	**Direction**	Direction_increase	
	Parameter	Parameter_level	temperature
	Origin	Origin_one	Enter the description of the option you choose
	Destination	Destination_many	Enter the description of the option you choose
	Aspect	Aspect_form	
Abstraction		Abstraction_system	Enter the description of the option you choose
Valuable Valuable_simple			

Generate Clear All

Statement:

In response to (Perturbation_disturbance) low temperature (Context_circumstantial) late at night, during (Phase_ops) of system, desire (Agent_external) to be able to (Direction_increase) the (Parameter_level) of knob angle from (Origin_one) state(s) to (Destination_many) state(s) in the system (Aspect_form) through (Mechanism_description) turning the knob that results in the effect of (Direction_increase) the (Parameter_level) of temperature from (Origin_one) state(s) to (Destination_many) state(s) in the system (Aspect_form) for a (Abstraction_system) that is (Valuable_simple).

Send

Result:

[Changeability,Flexibility]

Figure 2: Web interface for creating requirements statements and submitting them to the synthesized web service.

The final step in the loop was theory, service, and tool) evolution. We evolved our specification, web service, and web client based on our exploration of the tool and the many questions and answers produced during that exploration. We made changes in the formal model, extracted code, web service, and web client. We did this four times. A key to making this evolution reasonably easy was our significant initial investment in the framework that allows us to drop newly extracted code into the Yesod framework. What it mainly does is to translate objects whose types are given by (Haskell versions of) Coq types into (native) Haskell types, which is what the Yesod framework operates on. For example, Coq formalizes strings, but Haskell has its own string type. Our framework defines operations to lift and lower values between the Coq and Haskell type realms. Beyond modifying the theory and extracting and then packaging code as web services, we also updated web client documentation to explain the intended meaning and use of the theory. It was a revelation to see how important it was for the web tool to explain the formal model. The last version of the web site thus provides such affordances as *tool tips* that pop up when one hovers over data entry fields for filling in the template, to explain the meaning and intended use of the field.

Our web site lists four separate versions of the theory with corresponding web services and clients interfaces. Changes included consolidation of the formal grammar, its extension to support optional elements, completion of the incomplete type assignment function presented in the original paper, numerous clarifications of intent, and more.

We do not claim that the current version of the theory is ready for use by a broader community. What we have done is to convert a good but informal and under-developed idea (in a working paper, after all), to the point where we have a well-understood formal model, service, and web client as a basis for important next stages of ility theory development. The next stage involves leveraging the computer science notion that we specify changes as state transitions over in time. To do this requires a state space model of the system for which changeability requirements are being stated. We found the concept of such a system model missing from the original theory. Our approach worked well in this case to reveal opportunities to evolve an interesting and promising ility theory to a state of much greater fitness for use.

5.3. Interacting with the Tool

To give a clearer sense of how one interacts with the model/theory as it currently stands though the tool, consider a simple example, from Ross et al., illustrated once again in Figure 2. Assume that you have a requirement that you would like to be able to turn a knob on your system to turn up the temperature when triggered by a low temperature

alert late at night. The knob is part of the physical system instance that you own and operate. The temperature is one of the functional performance parameters. No particular desired end state is not specified. It's just assumed that you can turn it up from an arbitrary starting temperature to many possible increased levels of temperature. You, as an operator, are considered to be an agent external to the system, and to be the instigator of the change in temperature. The particular mechanism you are using to change the temperature is by turning the knob. It is a feature of the Ross model that desired changes in system state (temperature here) are effected indirectly through direct changes to control mechanisms. Thus two changes come into play in a requirements statement in this theory.

Mapping the change requirement we have envisioned into the formalized version of the template is done by entering appropriate choices in the pull-down menus and text boxes. The contents of the form are then extracted to the following rather cryptic string representation, suitable for submission to the web service, where strings in parentheses are options selected from pull-down menus, and underlined strings are parameters of these options: *In response to (Perturbation_disturbance) low temperature (Context_circumstantial) late at night, during (Phase_ops) of system, desire (Agent_external) to be able to (Direction_increase) the (Parameter_level) of knob angle from (Origin_one) state(s) to (Destination_many) state(s) in the system (Aspect_form) through (Mechanism) turning the knob that results in the effect of (Direction_increase) the (Parameter_level) of temperature from (Origin_one) state(s) to (Destination_many) state(s) in the system (Aspect_form) for a (Abstraction_system) that is (Valuable_simple).*

5.3.1. The underlying formal specification

At the heart of our approach is an evolving theory: a mathematical-logical-computational construct that we express in Coq's constructive logic. The theory we developed for this case study is relatively simple. It comprises a set of inductive type definitions that are composed into a definition of a type of change requirements statement, and a function mapping terms of this statement type to lists of ility *tipes*. (We use *tipe* here because these are not Coq types but *values/terms* of a Coq type representing ility statement types.)

Space doesn't permit a presentation of the full theory in the body of this paper. The complete specifications are available on our web site (http://beefcake.cs.virginia.edu). Rather, we present a few snippets of specification to sufficient to communicate the essence of our case study. For example, we define *description* as an alias for the Coq type, *string*, and on this basis we defined the *perturbation* type as an element of our requirements statement grammar having three possible values, as taken directly from Ross et al., who assert that a perturbation can be a disturbance, a shift, or not present (none), and where, in each case, and additional description is given to complete the term.

```
Definition description := string.

Inductive perturbation :=
  | perturbation_disturbance: description → perturbation
  | perturbation_shift: description → perturbation
  | perturbation_none: description → perturbation.
```

As we studied the Ross et al. model, we saw that *impetus* and *effect* (denoting the two levels of change) had nearly the same fields. We thus defined a single production/type in Coq and used this production twice in defining the overall requirements statement type, simplifying the grammar and increasing the conceptual integrity of the model.

```
Record change := mk_change {
  change_direction : direction;
  change_parameter : parameter;
  change_origin : origin;
  change_destination : destination;
  change_aspect : aspect
}.
```

The syntax of the requirements statement is captured equally straightforwardly.

```
Record changeStatement: Set := mk_changeStatement
{
  changeStatement_perturbation : perturbation;
```

```
  changeStatement_context : context;
  changeStatement_phase : phase;
  changeStatement_agent : agent;
  changeStatement_impetus : change;
  changeStatement_mech_mechanism : mechanism;
  changeStatement_effect : change;
  changeStatement_abstraction: abstraction;
  changeStatement_valuable: valuable
}.
```

Here's an example of a change-ility statement expressed as a value of the *changeStatement* type.

```
Definition changeStatement1 : changeStatement :=
  mk_changeStatement
    (perturbation_shift "some event")
    (context_circumstantial "some circumstantial context")
    phase_preOps
    (agent_internal "aAgent")
    (mk_change direction_increase (parameter_level "aParameter") (origin_one "anOrigin") (destination_one "aDestination") aspect_function)
    (mechanism_descption "some mechanism")
    (mk_change direction_increase(parameter_level "anotherParameter") (origin_one "anotherOrigin") (destination_one "anotherDestination") aspect_function)
    (abstraction_architecture "anAbstraction")
    (valuable_compound "valuable because of"
      (reaction_sooner_than 10 unit_time_second)
      (span_shorter_than 1 unit_time_day)
      (cost_less_than 100 unit_money_dollar)
      (benefit_same_as "keep power on")).
```

The set of values of yet another Coq type, *tipe* (elided), comprises the the semantic domain of ility type labels.

```
Inductive tipe : Set :=
  | evolvability
  | changeability
  | flexibility
  ...
```

We define a predicate function for each such ility *tipe*, taking a requirements statement and returning true if it has the given ility *tipe*. These rules formalize the rows of the the Ross et al. spreadsheet. They ask whether a statement has a particular structure. Underscores mean *don't care*. Here's an example: the *checkEvolvability* function returns true if a requirements statement is an *evolvability* requirement, as defined by Ross et al. to include *perturbation_shift*, *context_general*, *phase_interLC*, any *agent*, *direction_increase* or *direction_decrease*, any *parameter*, any *origin*, any *destination*, any *aspect* for *impetus*, *direction_increase* or *direction_decrease*, any *parameter*, any *origin*, any *destination*, any *aspect* for *effect*, *abstraction_architecture*, and any *valuable*. Our function captures this otherwise hard-to-grasp definition in a formal, testable, and computable form.

```
Definition checkEvolvability (s:changeStatement): bool :=
  match s with
    | mk_changeStatement (perturbation_shift _) (context_general _) phase_interLC _ (mk_change (direction_increase | direction_decrease) _ _ _ _) _ (mk_change (direction_increase | direction_decrease) _ _ _ _) . . . _ ⇒ true
    | _ ⇒ false
  end.
```

The remainder of our specification uses a variety of standard functional programming constructs to define a function, *tipeAssignment*, that maps a given requirements statement to a list of ility *tipe* values for which the corresponding

tipe checking predicates return true. Applying this function to *changeStatement1* (the example given above) returns the list *changeability, adaptability, agility, reactivity.*

Extraction of Haskell code from the Coq specification is straightforward.

`Extraction` *Language Haskell*.
`Recursive Extraction` *tipeAssignment*.

5.4. Theory Evaluation

Careful evaluation of the *validity*, or fitness for use, of an ility model is a critical element of our approach. The results of such an evaluation either justify its use of a model or reveal gaps that must be addressed. By the time we produced our fourth formal model of the informal model of Ross et al., we were satisfied that we had captured their intent in a nicely cleaned up, rationalized, fully formal, and automated theory, and at this point we asked the question of validity: Does this theory allow us to express *realistic change-related ility requirements* in useful ways.

To help answer this question, we conducted a single experiment. We picked a seminal paper on changeability from the software engineering literature: Parnas's 1972 paper [8] on the information hiding strategy for modularizing (software) systems to facilitate software evolution. We did a very careful reading of the paper and extracted change requirements explicit or implicit in that paper. We then tried to express these requirements in terms of the constructs provided by the Ross et al. theory. Finally, we reflected on challenges encountered in this effort.

Parnas's paper identifies two design strategies and compares the relative changeability of designs produced by these approaches. The strategies are top-down functional decomposition and information hiding modularity. In the former, one decomposes a design into functions that communicate through concrete data structures. In the latter, one identifies design decisions that are likely to change and designs stable interfaces behind which they are hidden, decoupling them from the rest of the design.

The paper postulates a set of likely changes and compares the changeability of respective designs in the face of each such change. The conclusion is that the information hiding modularization is far more changeable. Changeability of the functional design is relative poor because small changes impact shared data structures to which many function designs are coupled. By contrast, when concrete data structure design decisions are hidden behind what today we would call "APIs", the impacts of changes in data structures are limited to the modules in which they are encapsulated.

Our evaluation led to several insights. First, every change requirement we considered is cast in terms of some kind of system model. The kind of system model implicit in Parnas's paper is (what we call) a design influence graph (DIG), or what today we might view as a design structure matrix (DSM). Rows and columns represent design decisions, environment parameters that influence design decisions, and dependences among these parameters [12]. However, there is no way within the Ross et al. model to represent a design model in terms of which to cast change requirements. Rather, system models remain implicit in the natural language statements incorporated into such models.

Second, it wasn't clear whether Ross's model would consider a human software designer to be internal or external to the system. She could be considered as internal: as distinct from an affordance provided to system installers, operators, or users. On the other hand, a designer can also be seen as external to the system, in contrast to some kind of automated internal mechanism. In reality, there are multiple overlapping boundaries depending on one's perspective, each providing change-related affordances to a class of agents (as suggested above). Again, explicit system models would have been helpful.

Third, absent system models, we found it hard to be precise when either defining a change or expressing its cost. The computer scientist co-authors find it natural to think of changes as the actions of change operators on system states, e.g., adding a new module is a change in the design state of a system. We now see an opportunity to represent change operators over models of system state spaces explicitly. We are considering how to incorporate such a concept into a next version of the theory, or into a new theory inspired by the original work. A new theory would include system models and would re-cast the template for change requirements into one based on formal change operators, including notions of who or what should effect a change, time spans and other costs, and so forth.

Finally, in addition to merely being able to *state* change requirements, we then also want to be able to validate that a given system (model) will *satisfy* such requirements. We also want to be able to compare how well several proposed systems (or system models) do satisfying such ility requirements. Indeed, this is the notion at the very heart of the Parnas analysis. Once again we need not only to express change requirements but to assess costs of effecting such changes for multiple candidate system designs in terms of the impact of changes as revealed by analysis of different

system models. With our DIG system models, for example, we can estimate cost in terms of the weights of coupled design decisions downstream from the immediate target of a change operation. Such an analysis is at the core of Parnas's seminal work.

6. Conclusion and Future Work

To accelerate the development of a science of system properties (ilities), we have proposed a co-evolving theory-systems approach. An expressive math-logic notation enables precise expression and automation, facilitating dissemination, evaluation, and feedback, driving both theory and technology toward fitness for use. Resulting web services are trustworthy because their software is derived from such formalisms. Code synthesis relieves analysts of much of the burden of maintaining software as theories evolve. As an initial test we applied this approach to the *semantic basis* model, of Ross et al. A key shortcoming identified in this work is the need to connect ilities to system models.

Future work will address a broader range of ilities, e.g., *security resiliency*, and mappings between design parameters and ilities, and between ilities to stakeholder value. These ideas are related to those of Suh [11] and others who have developed relational (e.g., coupling matrix) models. Our approach is distinguished by its use of type theory to express and reason about complex, discrete, and computational systems and concepts and propositional properties and relationships involving them.

The goal of this work is to enable system developers to think and to communicate more effectively than they can today about system properties, underlying design decisions, and stakeholder value. We are hopeful that the expressiveness and formality of type theory can help. We also recognize that making such science practical will require *end-user* concepts, methods, and tools. The systems component of our theory-systems approach is meant in part to address these needs.

7. Acknowledgements

This work was supported in part by grant from the National Science Foundation, CMMI #1400294, to Kevin Sullivan and the University of Virginia. This material is also based upon work supported, in whole or in part, by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract H98230-08-D-0171. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense nor ASD(R&E). We also thank the anonymous reviewers for their insightful critiques and suggestions.

References

[1] Luca de Alfaro, Thomas A. Henzinger, and Marielle Stoelinga. Timed interface. In Proceedings of the Second International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2491, Pages 108-122., 2002.
[2] Boehm, Barry W., and Rony Ross. Theory-W software project management principles and examples. Software Engineering, IEEE Transactions on 15.7 (1989): 902-916.
[3] Coq Development Team. Reference Manual (version 8.4pl4), Inria, 2012.
[4] Fielding, Roy T. and Taylor, Richard N., Principled Design of the Modern Web Architecture. ACM Trans. Internet Technol. 2.2 (2002): 115–150.
[5] Haskell Development Team. http://www.haskell.org/haskellwiki/Haskell, 2014.
[6] Martin Glinz, On Non-Functional Requirements. In Proceedings of the 15th IEEE International Requirements Engineering Conference, pages 21-26, 2007.
[7] Thomas Henzinger and Slobodan Matic. An interface algebra for real-time components. In Proceedings of RTAS 2006, April, 2006, pages 253263, 2006.
[8] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Volume 15 Issue 12, Pages 1053-1058, 1972.
[9] Adam M. Ross and D. H. Rhodes, A prescriptive semantic basis for system lifecycle properties, SEAri Working Paper Series, 2012.
[10] Jean-Bernard Stefani. Project-Team Sardes - System Architecture for Reflective Distributed Computing Environments, Activity Report, INRIR Rhone-Alpes, 2006.
[11] Suh, Nam P. "Axiomatic Design: Advances and Applications (The Oxford Series on Advanced Manufacturing)." (2001).
[12] Sullivan, Kevin J., et al. "The structure and value of modularity in software design." ACM SIGSOFT Software Engineering Notes 26.5 (2001): 99-108.
[13] Yesod Development Team. http://www.yesodweb.com, 2014.