

# An Artificial Science for System Value Engineering and Assurance

Chong Tang, Kevin Sullivan, Ke Dou, Koleman Nix  
*Department of Computer Science*  
*University of Virginia*



May 14, 2015

# Problems

- ▶ The space of system properties is not flat
- ▶ Stakeholders have conflicting property preferences
- ▶ Properties can be coupled in complex ways
- ▶ Lacking foundations for rigorous engineering of properties

# Consequences

- ▶ Projects canceled after consuming billions of dollars
- ▶ Projects overrun their budgets and deadlines
- ▶ Delivered systems have less capability than required
- ▶ Systems experience serious operational failures
- ▶ System developers game the slack

# Causes

- ▶ Lacking focus on multi-property, value-driven engineering
- ▶ Related research have been lacking in rigor and precision

# Contributions

- ▶ Formalizing and improving Boehm's informal taxonomy
- ▶ Producing general but also specializable formal theories
- ▶ Integrating Boehm's taxonomy with Ross's approach
- ▶ Testing two propositions through the integration effort
- ▶ Unifying definition, specification and assurance cases
- ▶ A theory that supports formal reasoning about properties
- ▶ An example of applying the theory to a specific system

# Purpose

To provide an approach for:

- ▶ Making accessible of the theories to practitioners
- ▶ Evolving the theories with the needs of practitioners

# State of The Art

Ross's Semantic Approach (?, ?)

## ► Problem:

No precise understanding of particular system properties

## ► Key Idea:

A semantic approach for defining change-related ility terms

## ► Main Contributions:

- *Informal grammar* for changeability requirements
- Rules for *classifying statements* by *ility*
- Providing *semantics* to ility terms

Ross's semantic basis approach <sup>1</sup>[illegible]

Perturbation	Context	Phase	Agent	Inst* (optional)				Mechs	Outcome				Abstraction	Reaction	Span	Cost	Details	
				Nature	Parameter	Origin	Destination	Aspect	Mechanism	Effector	Parameter	Origin						
shift		opt							same			few						By Label
disturbance		opt							same			few						Value Robustness
shift		opt							same			few						Value Survivability
shift		opt		not same					same			few						Robustness
shift		opt		same					same			few						Active Robustness
shift		opt		same		few	few		same			few						Passive Robustness
shift		opt	none	same		few			same	level		few	form	system				Classical/Passive Robustness
disturbance		opt							same			few						Survivability
									not same									Changeability
shift	general	inter-LL	either	not same					not same					architecture				Evolvability
				internal					not same									Adaptability
				external					not same									Flexibility
				not same					not same	level								Scalability
				not same					not same	set								Mobility
		opt	either	not same					increase	set								Extensibility
				not same					not same									Agility
				not same					not same									Reactivity
		opt		Disorient per	one	one	form	operations	not same	Link per			form		sooner			Form Recognoizability
		opt		same	one	one	operations		not same	Order per			operations					Operational Reconfigurability
		opt		same	one	one	form/ops		not same	set		few/many	function					Functional Versatility
		opt		same	one	one	form/mot		not same	set		few/many	operations					Operational Versatility
				same	one	one			not same	set		form						Substitutability

Figure: Ross's prescriptive semantic basis for change-type ilites



# Ross's semantic basis approach

- **Pros:**

- Defining change-related ilities requirements statements

- **Cons:**

- Informal, not computable, hard to evaluate and evolve

# State of The Art

Boehm's top-down Taxonomy (?, ?)

## ► Problem:

System designs are deficient in balancing system ilities

## ► Key Ideas:

- Defining language grammer for full range of ilities
- Balancing ility values for the system's stakeholders

## ► Main Contributions:

- Proposing a stakeholder-value based property hierarchy
- An ontology for reasoning about a system's ilities
- Studied Synergies and Conflicts among key properties

# Boehm's top-down Taxonomy <sup>2</sup>

Stakeholder Value-Based QA Ends	Contributing QA Means
Mission Effectiveness	Stakeholders-satisfactory balance of Physical Capability, Cyber Capability, Human Usability, Speed, Endurability, Maneuverability, Accuracy, Impact, Scalability, Versatility, Interoperability
Resource Utilization	Cost, Duration, Key Personnel, Other Scarce Resources; Manufacturability, Sustainability
Dependability	Security, Safety, Reliability, Maintainability, Availability, Survivability, Robustness
Flexibility	Modifiability, Tailorability, Adaptability
Composite QAs	
Affordability	Mission Effectiveness, Resource Utilization
Resilience	Dependability, Flexibility

Figure: Stakeholder-value based property means-ends hierarchy

<sup>2</sup>Figure from (?, ?)

# Boehm's top-down Taxonomy

## ► Pros:

- Clarifying the nature of system ilities
- Reasoning about the tradeoffs among ilities
- Addressing stakeholder value conflicts

## ► Cons:

Informal, difficult to validate, hard to apply

# State of The Art

## Assurance Cases

- ▶ Claim - Assertion about key requirements and properties
- ▶ Evidence
  - ▶ Testing, Proofs, Process and people, Review and analyses
- ▶ Argument - How the evidences support the claims
  - ▶ Inference rules: deterministic, probabilistic, qualitative
- ▶ Inductive reasoning
  - ▶ Providing evidence, not proof that the claim is certain

# Assurance Cases <sup>3</sup>

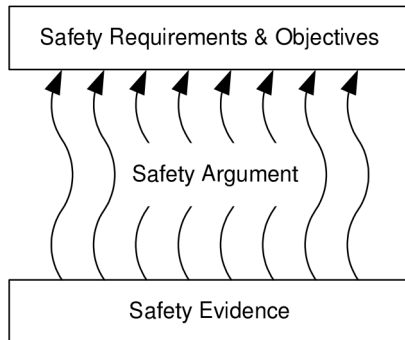


Figure: The relationship among safety case elements

<sup>3</sup>Figure from (?, ?)

# State of The Art

Kelly's Goal Structuring Notation (?, ?)

## ► Problem:

Safety arguments are often poorly communicated

## ► Key Idea:

Develop safety cases in a reader-friendly manner

## ► Main Contributions:

- Using graphical notations to annotate the assurance cases
- Applying *inductive* argumentation to safety cases

## Kelly's Goal Structuring Notation

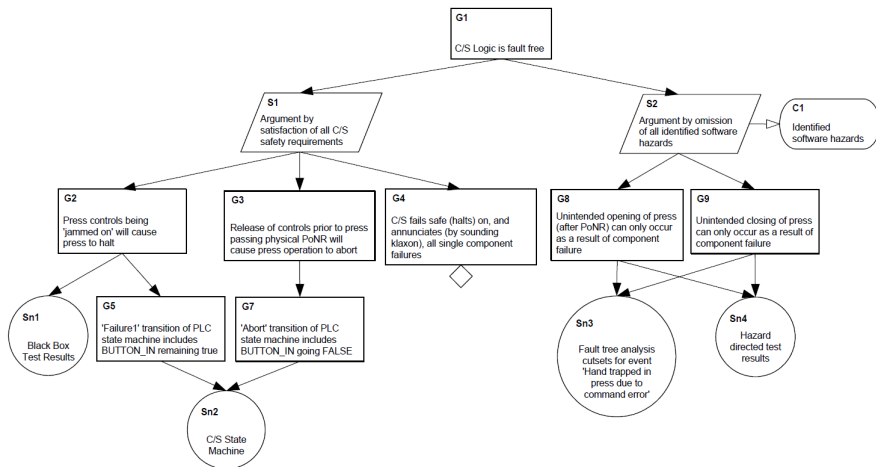


Figure: Example GSN (Figure from (?, ?))



# Kelly's GSN safety argument notation

- **Pros:**

Facilitate comprehension and communication of arguments

- **Cons:**

Informal, syntax rules are defined in prose text, not scale

# State of The Art

Rushby's Theory (?, ?)

## ► Problem:

Increasing confidence in the soundness of a given case

## ► Key Ideas:

- Applying formalism to safety cases
- Eliminating logic doubt and focusing on epistemic logic

## ► Main Contributions:

- Formalizing parts of a safety argument into deductive logic
- Providing mechanized support for assurance case argument
- Helping engineers focus on evidence instead of argument

# Rushby's Theory

- ▶ **Pros:**

Improving efficiency and cost of safety argument checking

- ▶ **Cons:**

No empirical evidence

# State of The Art

Knight's Assurance Based Development (?, ?)

- ▶ **Problem:**

Assurance cases often fail to guide developers' decisions

- ▶ **Key Idea:**

Co-developing the software system and its assurance case

- ▶ **Main Contributions:**

- ▶ Integrating assurance into development process.
- ▶ Assurance requirements drive development decisions

# Knight's Assurance Based Development

- ▶ **Pros:**

Detecting the assurance difficulties from the earliest stages

- ▶ **Cons:**

Hard to validate that their approach is optimal

# State of The Art

Basir's Automatically Generated Argument (?, ?)

- **Problem:**

Formal proofs are complex and machine-oriented

- **Key Idea:**

Automatically generating a safety argument by converting natural deduction style proofs

- **Main Contributions:**

- helps human understand the formal proofs

# Basir's Automatically Generated Argument

► **Pros:**

Providing easier-to-understand proofs

► **Cons:**

- No benefit over an hand-generated, informal argument
- Far from satisfactory as the proofs contain too many details

# State of The Art

Bosch's Mobile Service Oriented Architectures (?, ?)

- ▶ **Problem:**

It's hard to achieve success in realizing mobile services

- ▶ **Key Idea:**

Defining the architecture drivers that make success

- ▶ **Main Contributions:**

- ▶ Identified the goals for mobile service oriented architectures
- ▶ Identified ilities that influence the success of mobile services
- ▶ Predicted future trends of mobile service



# State of The Art

## Lundberg's Architecture Design Guidelines (?, ?)

### ► **Problem:**

There are conflicts between modifiability and performance

### ► **Key Idea:**

Providing guidelines in software architecture design

### ► **Main Contributions:**

- A taxonomy for performance and modifiability related QA
- Four software architecture design evaluation approaches
- Four architecture design transformation strategies
- Eight guidelines in software architecture design

# Lundberg's Architecture Design Guidelines

## ► Pros:

- Revealed the relationships among architecture, quality attributes, and implementation
- The guidelines are extracted from real industry experience

## ► Cons:

- Only focus on performance and modifiability
- Such studies may not fit domains other than software design

# State of The Art

## Knight's Success Arguments (?, ?)

### ► **Problem:**

Failure rate of software development efforts is high

### ► **Key Idea:**

Defining success argument to establish confidence

### ► **Main Contributions:**

- Structuring and documenting the argument
- Recording the argument and exposing it to examinations

# Knight's Success Arguments

## ► Pros:

- Helps structure the reasoning and expose it to criticism
- Helps explain the evidence to the reviewers

## ► Cons:

- Informal, Hard to validate

# Our approach

- ▶ Combining Bosch's innovation experiment systems theory
- ▶ Integrating Boehm's theory and Ross's approach
- ▶ Using rigorous formal specification and software synthesis
- ▶ Refining and expressing quality theories using Coq
- ▶ Building web-based tools to implement the theory concepts
- ▶ Driving theory testing, evolution, and validation with tools

# Framework Architecture

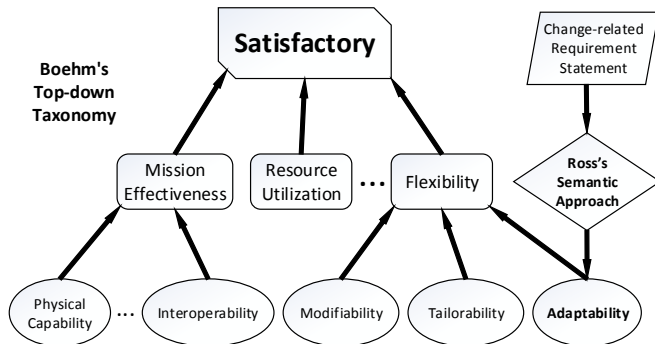


Figure: The overall architecture of our framework

# Top-Most System Value – Satisfactory

```

Class Satisfactory (System: Set) (Stakeholder: Set) (Context: Set) := {
  sys: System

  ; physicalCapability : System → Stakeholder → Context → Prop
  ; cyberCapability : System → Stakeholder → Context → Prop
  ; humanUsability : System → Stakeholder → Context → Prop
  .....
  ; adaptability : System → Context → Prop

  ; me: MissionEffective System Stakeholder Context sys physicalCapability cyberCapability
  humanUsability speed endurance maneuverability accuracy impact scalability versatility interoperability
  ; ru: ResourceUtilization System Context sys cost duration keyPersonnel otherScareResources
  manufacturability sustainability
  ; dp: Dependable System Context sys security safety reliability maintainability availability survivability
  robustness
  ; fl: Flexible System Context sys modifiability tailorability adaptability

```

# *Mission Effectiveness* in QA Taxonomy [Boehm, to app]

Mission Effectiveness: a System has achieved a  
Stakeholders-satisfactory balance of  
Physical Capability, Cyber Capability, Human Usability,  
Speed, Endurability, Maneuverability, Accuracy, Impact,  
Scalability, Versatility, and Interoperability.



# Second-Level Property – Mission Effective

```

Inductive MissionEffective (System: Set) (Stakeholder: Set) (Context: Set) (sys: System)
  (physical_capable: System → Stakeholder → Context → Prop)
  (cyber_capable: System → Stakeholder → Context → Prop)
  (human_usable: System → Stakeholder → Context → Prop)
  (speed: System → Stakeholder → Context → Prop)
  (endurable: System → Stakeholder → Context → Prop)
  .....
  (interoperable: System → Stakeholder → Context → Prop)
  : Prop :=

```

```

mk_mission_eff:

```

```

  PhysicalCapable System Stakeholder Context sys physical_capable →

```

```

  CyberCapable System Stakeholder Context sys cyber_capable →

```

```

  HumanUsable System Stakeholder Context sys human_usable →

```

```

  Speed System Stakeholder Context sys speed →

```

```

  Endurable System Stakeholder Context sys endurable →

```

```

  Maneuverable System Stakeholder Context sys maneuverable →

```

```

  .....

```

```

  MissionEffective System Stakeholder Context sys mission_effective physical_capable

```

```

  cyber_capable human_usable speed endurable maneuverable accurate impact scalable versatile
  interoperable.

```

## Second-Level Property – Flexible

Inductive ***Flexible*** (*System*: Set) (*Context*: Set) (*sys*: *System*)  
 (*flexible*: *System* → *Context* → Prop)  
 (*modifiable*: *System* → *Context* → Prop)  
 (*tailorable*: *System* → *Context* → Prop)  
 (*adaptable*: *System* → *Context* → Prop)  
 : Prop :=

*mk\_flexibility*:

***Modifiable*** *System Context sys modifiable* →

***Tailorable*** *System Context sys tailorable* →

***Adaptable*** *System Context sys adaptable* →

***Flexible*** *System Context sys flexible modifiable tailorable adaptable*.

# Leaf Property – Adaptable

Inductive ***Adaptable*** (*System*: Set) (*Context*: Set) (*sys*: System)  
 (*adaptable*: System → Context → Prop)  
 : Prop :=

*mk\_adaptability*:

( $\forall$  *cx*: Context, *adaptable sys cx*) →  
***Adaptable System Context sys adaptable***.

# Example – Smart Home

Define System, Stakeholder, and Context for a Smart Home

Require Import Satisfactory.

Require Import Changeable.

Definition Smart\_Home\_System := **Datatypes.unit**.

Inductive **Smart\_Home\_Stakeholder** := investor | end\_user |  
developer | maintainer | public.

Inductive **Smart\_Home\_Context** := normal.

# Example – Smart Home

Create a Specific Adaptability Requirement using Ross's Approach

Definition smart\_home\_system\_adaptability\_requirement :

**changeStatement** :=

mk\_changeStatement

(perturbation\_shift "low temperature")

(context\_circumstantial "late at night")

phase\_preOps

(agent\_internal "controller")

(mk\_change direction\_increase (parameter\_level "knob angle")

(origin\_one "degree") (destination\_one "degree") aspect\_function)

(mechanism\_description "regulating the airflow")

(mk\_change direction\_increase (parameter\_level "temperature")

(origin\_one "degree") (destination\_one "degree") aspect\_function)

(abstraction\_architecture " ")

valuable\_simple

# Example – Smart Home

## Corresponding requirement statement:

In response to (Perturbation\_shift) *low temperature* (Context\_circumstantial) *late at night*, during (Phase\_preOps) of system, desire (Agent\_internal) *controller* to be able to (Direction\_increase) the (Parameter\_level) of *knob angle* from (Origin\_one) state(s) to (Destination\_one) state(s) in the system (Aspect\_function) through (Mechanism\_description) *regulating the airflow* that results in the effect of (Direction\_increase) the (Parameter\_level) of *temperature* from (Origin\_one) state(s) to (Destination\_one) state(s) in the system (Aspect\_function) for a (Abstraction\_architecture) that is (Valuable\_simple).

# Example – Smart Home

Check a given system meets the adaptability requirement

Inductive **systemMeetsSpecificAdaptabilityRequirement**:

$\text{Smart\_Home\_System} \rightarrow \text{changeStatement} \rightarrow \text{Prop} :=$

$\text{systemMeetsSpecificAdaptabilityRequirement\_proof}:$

$\forall s: \text{Smart\_Home\_System}, \forall c: \text{changeStatement},$

**In** adaptability (typeAssignment  $c$ )  $\rightarrow$

**systemMeetsSpecificAdaptabilityRequirement**  $s\ c$ .

# Example – Smart Home

Check a given system has adaptability quality

Inductive **adaptability** (*sys*: Smart\_Home\_System) (*cx*:  
**Smart\_Home\_Context**): Prop :=  
 adaptability\_proof:  
**systemMeetsSpecificAdaptabilityRequirement** *sys*  
 smart\_home\_system\_adaptability\_requirement →  
**adaptability** *sys cx*.



# Example – Smart Home

Formalize two properties with trivial proofs

Inductive **systemCanControlFurnaceOnOffSwitch**:

Smart\_Home\_System  $\rightarrow$  Prop :=

systemCanControlFurnaceOnOffSwitch\_proof:  $\forall s$ :

Smart\_Home\_System, **systemCanControlFurnaceOnOffSwitch**  $s$ .

Inductive **systemCanControlGarageDoorOpener**:

Smart\_Home\_System  $\rightarrow$  Prop :=

systemCanControlGarageDoorOpener\_proof:  $\forall s$ :

Smart\_Home\_System, **systemCanControlGarageDoorOpener**  $s$ .

# Example – Smart Home

Check a given system has Physical Capability quality

Inductive **physicalCapability** (*sys*: Smart\_Home\_System) (*sh*:  
**Smart\_Home\_Stakeholder**) (*cx*: **Smart\_Home\_Context**): Prop :=  
 physicalCapability\_proof: **systemCanControlFurnaceOnOffSwitch**  
*sys*  $\wedge$  **systemCanControlGarageDoorOpener** *sys*  $\rightarrow$   
**physicalCapability** *sys sh cx*.

# Example – Smart Home

Define an instance of Satisfactory for a smart home project

```
Instance Smart_Home_Instance: Satisfactory Smart_Home_System
Smart_Home_Stakeholder Smart_Home_Context := {
    sys := tt

    ; physicalCapability := physicalCapability
    ; cyberCapability := cyberCapability
    ; humanUsability := humanUsability
    .....
    ; tailorability := tailorability
    ; adaptability := adaptability
  }.
```

# Our Contributions

- ▶ A parameterizable hierarchy of qualities and relationships
- ▶ Quality-specific languages for expressing requirements
- ▶ Integration of the distinct, previously conflicting theories.
- ▶ Web-based software implementations of the theory concepts
- ▶ An approach for theory testing, evolution, and validation

The overall contribution of this work is a novel, rigorous, and promising new approach to developing, promulgating, testing, evolving, and validating the scientific theory that is needed to underpin rigorous new approaches to comprehensive system quality engineering.

# Why do we think it will work?

- ▶ Replaces vague prose with *verifiable propositions*
- ▶ Every proposition has corresponding *assurance case*
- ▶ Practitioners never have to see formal specifications
- ▶ Web-based tools provide for *broad accessibility*
- ▶ Evolution of theory driven by *feedback from use*
- ▶ Social process of learning, testing, *theory validation*

# Conclusion

- To be added

# Bibliography