Rigorous System Design

IFIP W.G. 2.2 meeting
IMT Lucca
September 21, 2015

Joseph Sifakis
The BIP team Grenoble
EPFL Lausanne

Practice vs. Theory – Poorly Engineered Systems

- Theory has had a decreasing impact on practice over the past decades
- The need for rigorous disciplined design is often directly or indirectly questioned by developers of large-scale systems (e.g., web-based systems) who privilege purely experimental approaches
- The aim is to find laws that govern/explain observed phenomena rather than to investigate design principles for achieving a desired behavior: *"On line companies . . . . don't anguish over how to design their Web sites. Instead they conduct controlled experiments by showing different versions to different groups of users until they have iterated to an optimal solution" .*
- Poorly engineered systems with the exception of a few well-regulated areas, e.g. avionics, the majority of systems are poorly engineered
- The IoT vision cannot come true, under current conditions. The main roadblocks to its achievement
  - poor dependability of infrastructures and systems
  - impossibility to guarantee response times in communication

Practice vs. Theory – SW Engineering

WHAT HAPPENED TO software engineering?
What happened to the promise of rigorous, disciplined, professional practices for software development, like those observed in other engineering disciplines?
What has been adopted under the rubric of “software engineering” is a set of practices largely adapted from other engineering disciplines: project management, design and blueprinting, process control, and so forth. The basic

Today’s software craftsmanship movement is a direct reaction to the engineering approach. Focusing on the craft of software development, this movement questions whether it even makes sense to engineer software

One might suggest computer science provides the underlying theory for software engineering—and this was, perhaps, the original expectation when software engineering was first conceived.
In reality, however, computer science has remained a largely academic discipline, focused on the science of computing in general but mostly separated from the creation of software-engineering methods in industry.
While “formal methods” from computer science provide the promise of doing some useful theoretical analysis of software, practitioners have largely shunned such methods (except in a few specialized areas such as methods for precise numerical computation).

Practice vs. Theory – Increasing Gap

Practically-oriented research

- *frameworks for programming or modeling real systems are constructed in an ad hoc manner - by putting together a large number of constructs and primitives.*
- *these frameworks are not amenable to formalization. It is also problematic to assimilate and master their concepts by reading manuals of hundreds of pages.*

Theoretical research

- *has a predilection for mathematically clean theoretical frameworks, no matter how relevant they can be.*
- *results are "low-level" and have no point of contact with real life problems - they are mainly based on transition systems which are structure-agnostic and cannot account for real-languages, design principles, architectures etc.*

Practice vs. Theory – What Kind of Theory?

letters to the editor

DOI:10.1145/2702734

Software Engineering, Like Electrical Engineering

THOUGH I AGREE with the opening lines of Ivar Jacobson's and Ed Seidewitz's article "A New Software Engineering" (Dec. 2014) outlining the "promise of rigorous, disciplined, professional practices," we must also look at "craft" in software engineering if we hope to raise the profession to the status of, say, electrical engineering. Most of safety-critical systems using three different techniques.

Modern craft methods like Agile software development help produce non-trivial software solutions. But I have encountered a number of such solutions that rely on the chosen framework to handle scalability, assuming that adding more computing power is able to overcome performance and

Practice vs. Theory – What Kind of Theory?

Both computing and physics deal with systems X'= f(X,Y) where

Computing

X' is the next state

X is the current state variable

Y is the current input variable

Discrete variables

Physics

X'=dX/dt

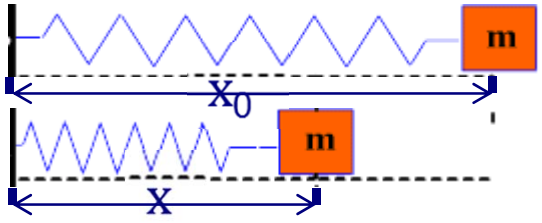
X is the state variable

Y is the input variable

Variables are functions of time

while $x \neq y$
do if $x>y$ then $x:=x-y$
else $y:=y-x$

Law: $GCD(x,y)=GCD(x_0,y_0)$



$m\frac{d^2x}{dt^2}=-kx$

Law: $\frac{1}{2} kx_0^2 - \frac{1}{2} kx^2 = \frac{1}{2} mv^2$

Significant differences:

- For physical systems behavior is described by equations
- Physical systems are inherently synchronous and driven by uniform laws.
- Computation models ignore physical time and are driven by specific laws defined by their designers

Practice vs. Theory – What Kind of Theory?

- Achieving correctness
  - Verification is a stopgap – it should be applied whenever possible and cost-effective
  - The ambition to build completely flawless systems is simply not realistic

- Shift focus from formal methods to design as a well-defined process leading from requirements to systems
  - Learn from successful design paradigms – HW, critical systems
  - Theory and techniques for reusing not only components but also principles based on a minimal number of well-defined and expressive concepts and constructs
  - Correctness by construction along the design flow based on principles of compositionality and composability

OVERVIEW

- System Design

- Rigorous System Design
  - Separation of Concerns
  - Component-based Design
  - Semantically Coherent Design
  - Correct-by-construction Design

- Discussion

System Design – About Design

Design is a Universal Concept!

.

Apple Pie

Proceduralization

RECIPE
(Program)

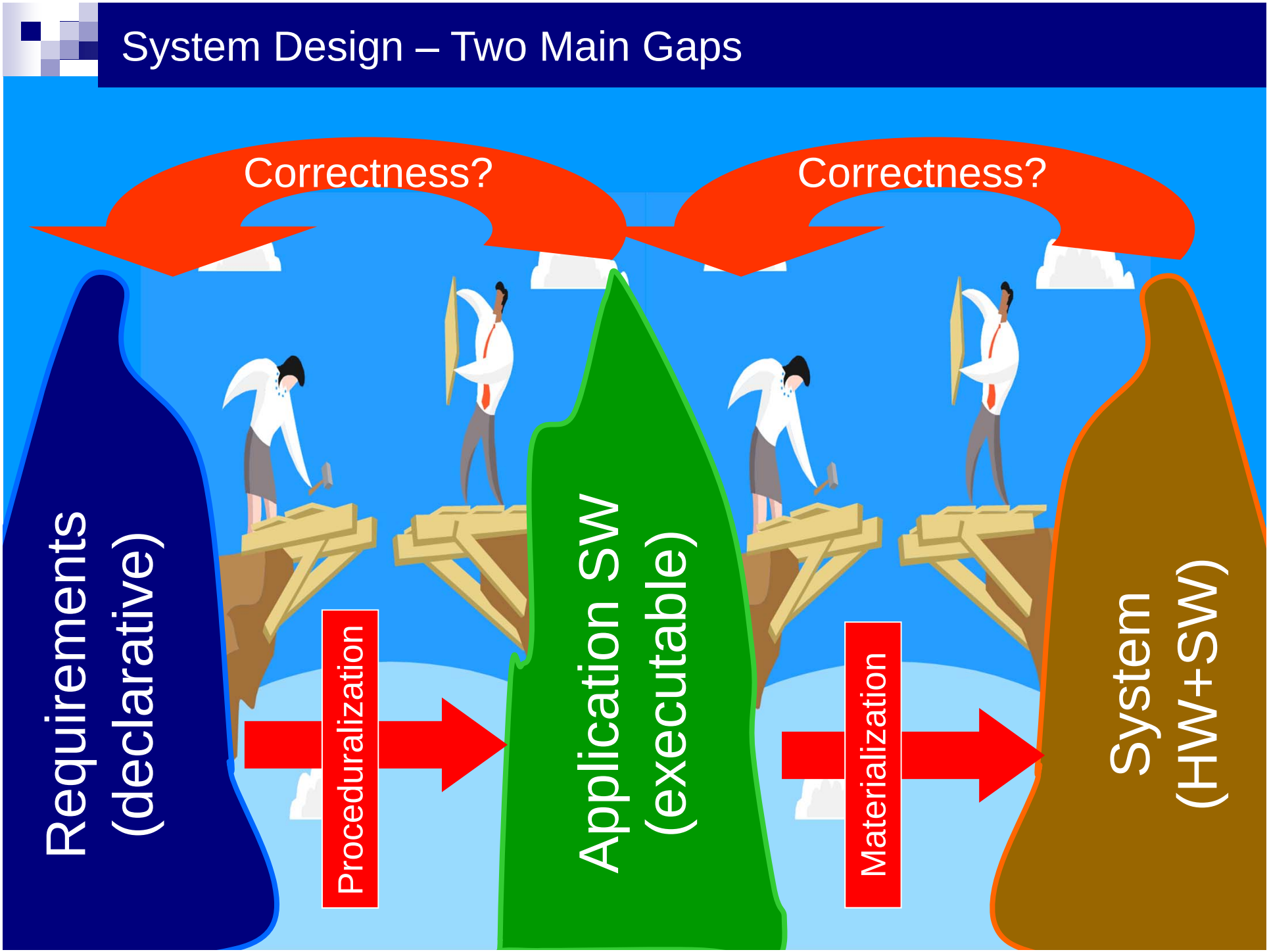
- Put apples in pie plate;
- Sprinkle with cinnamon and 1 tablespoon sugar;
- In a bowl mix 1 cup sugar, flour and butter;
- Blend in unbeaten egg, pinch of salt and the nuts;
- Mix well and pour over apples;
- Bake at 350 degrees for 45 minutes

Materialization

INGREDIENTS
(Resources)

- 1 pie plate buttered
- 5or 6 apples, cut up
- ¾ c. butter, melted
- 1 c. flour
- ½ c. chopped nuts
- 1tsp cinnamon
- 1tbsp sugar
- 1c. Sugar

System Design – Two Main Gaps



System Design – The Concept of Correctness for Systems

Trustworthiness requirements express assurance that the designed system can be trusted that it will perform as expected despite

HW failures

Design/Programming Errors

Environment Disturbances

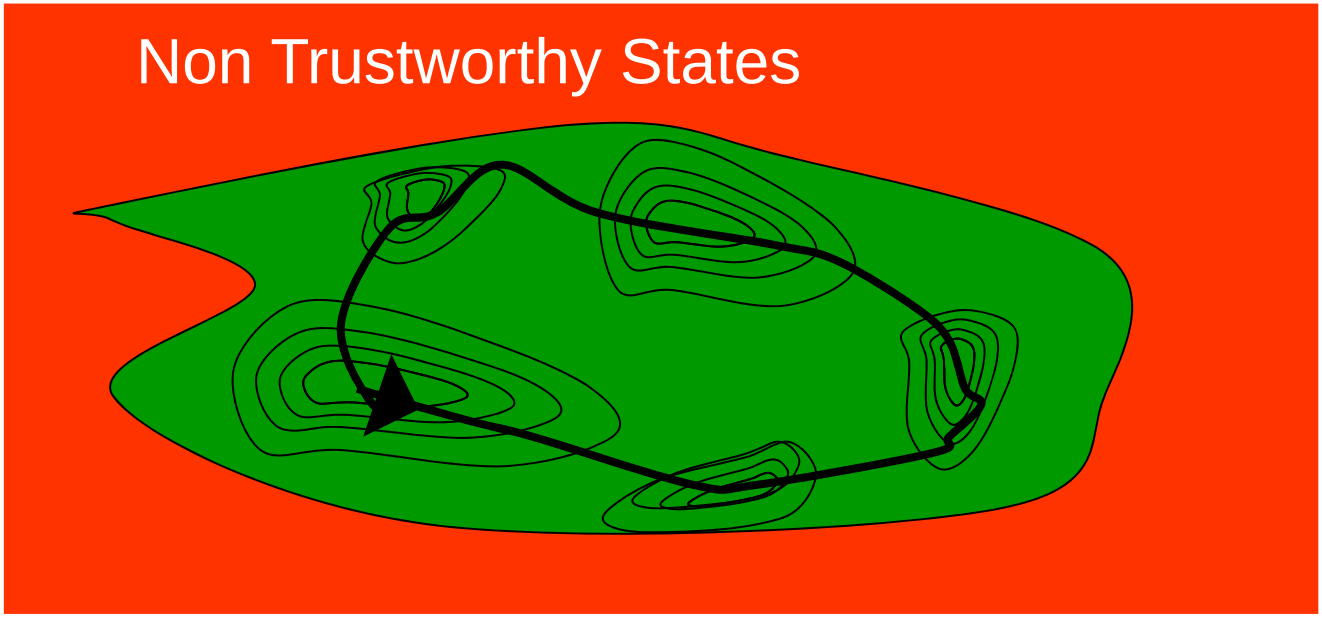
Malevolent Actions

Optimization requirements are quantitative constraints on resources such as time, memory and energy characterizing

1) performance e.g. throughput, jitter and latency
2) cost e.g. storage efficiency, processor utilizability
3) tradeoffs between performance and cost

System Design – Trustworthiness vs. Optimization

- Trustworthiness requirements characterize qualitative correctness – a state is either trustworthy or not
- Optimization requirements characterize execution sequences



Trustworthiness vs. Optimization

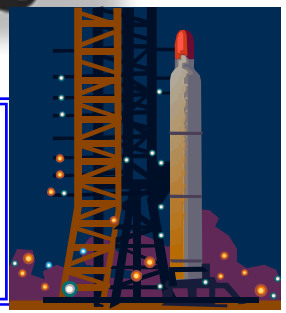
- The two types of requirements are often antagonistic
- System design should determine tradeoffs driven by cost-effectiveness and technical criteria

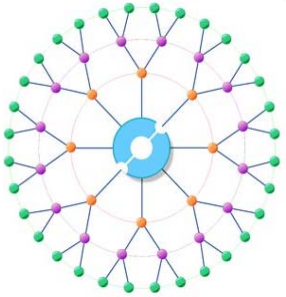
System Design – Critical vs. Non-critical

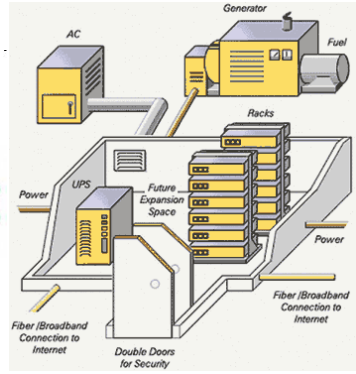
Safety critical: a failure may be a catastrophic threat to human lives

Security critical: harmful unauthorized access

Mission critical: system availability is essential for the proper running of an organization or of a larger system







Best-effort: optimized use of resources for an acceptable level of trustworthiness

amazon.com

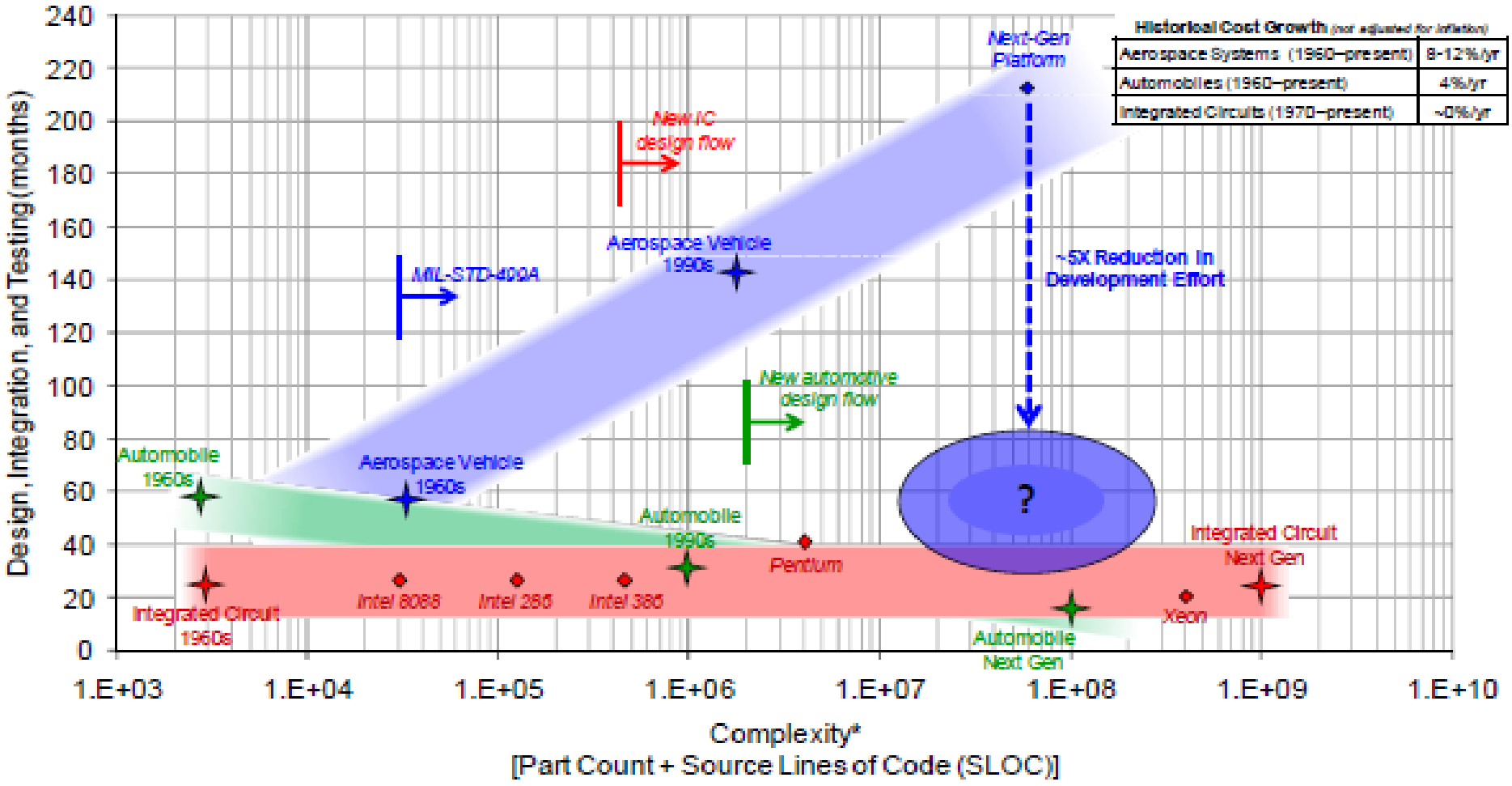
Google

- For critical systems development costs increase exponentially with their size!
- Developing of mixt criticality systems is a challenge!

System Design – The Cost of Trustworthiness

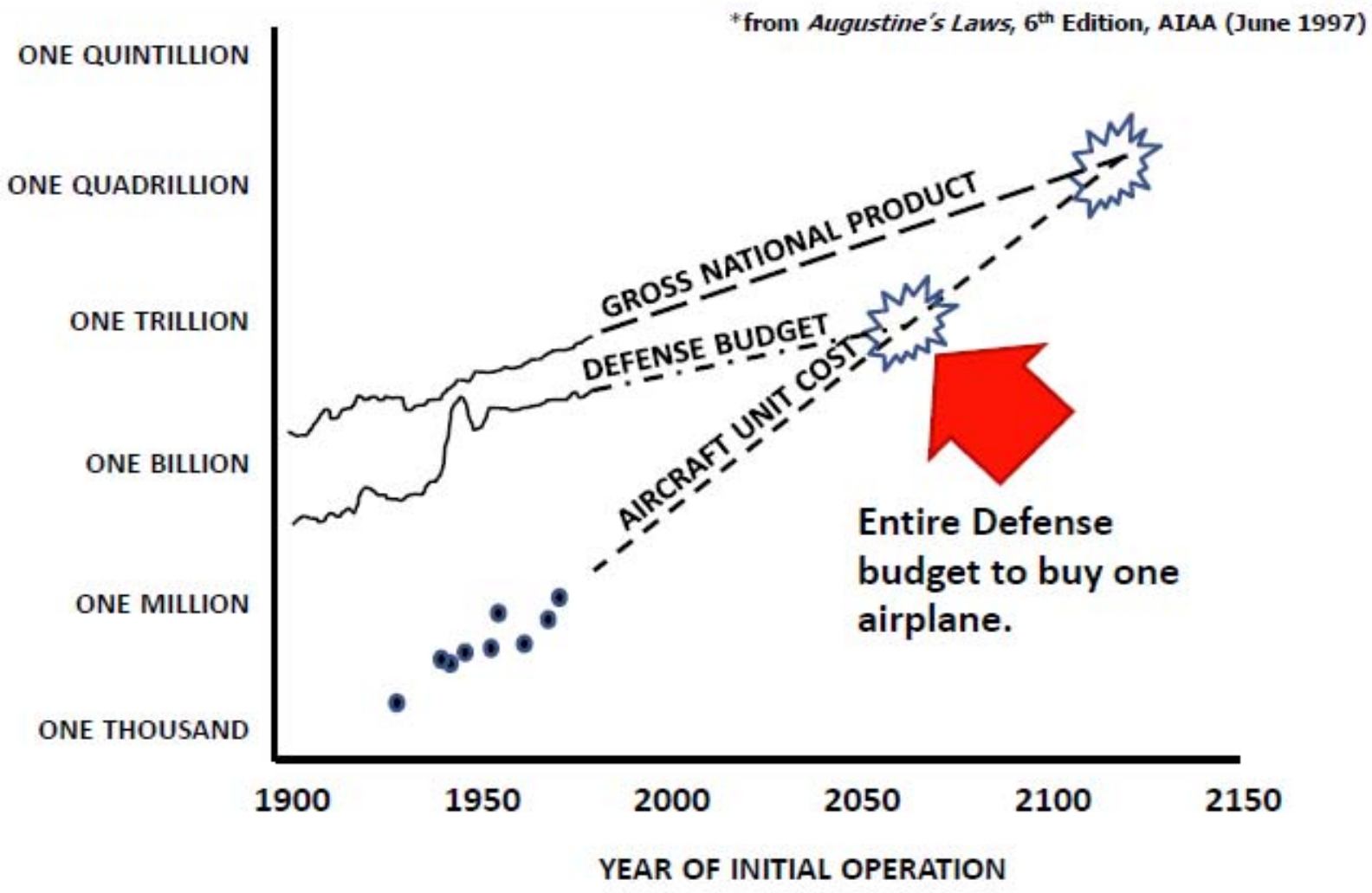
DARPA

Historical schedule trends with complexity

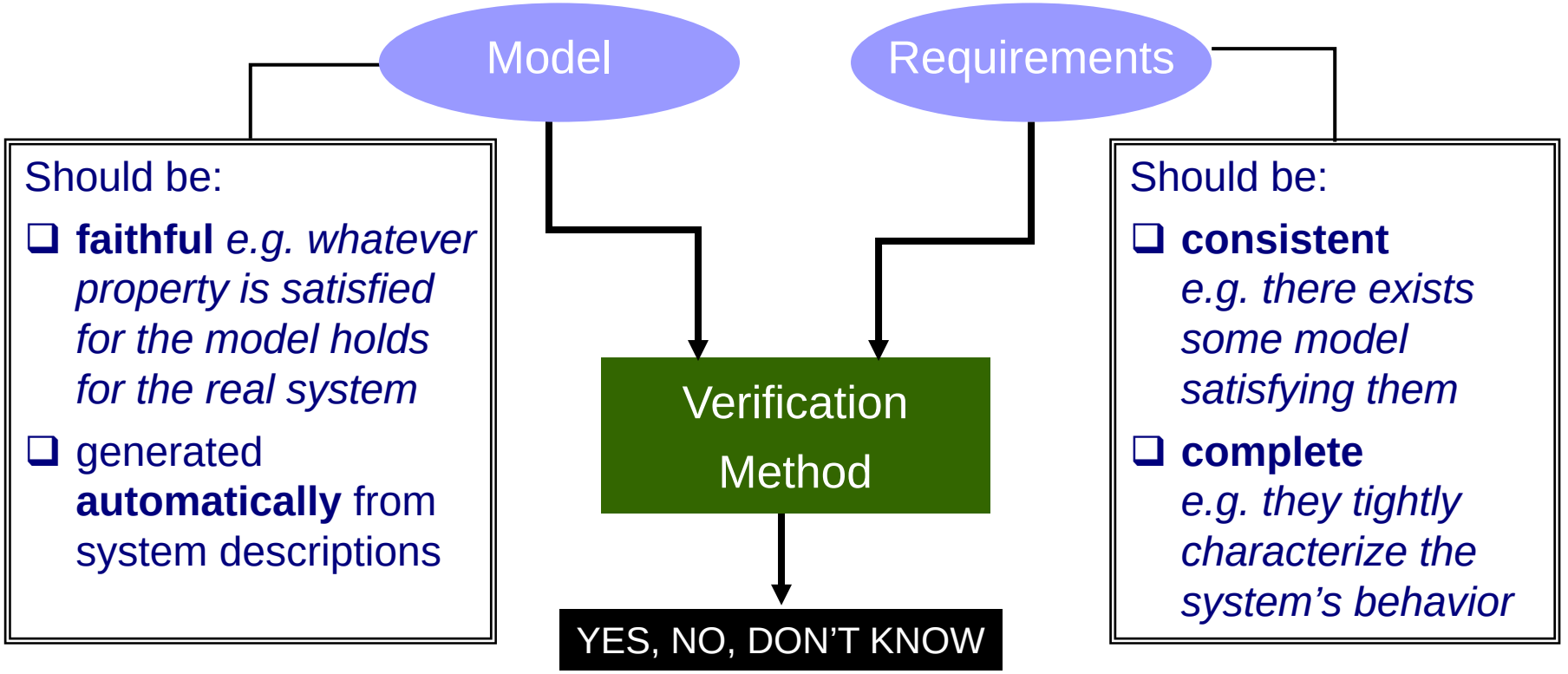


System Design – The Cost of Trustworthiness

The problem



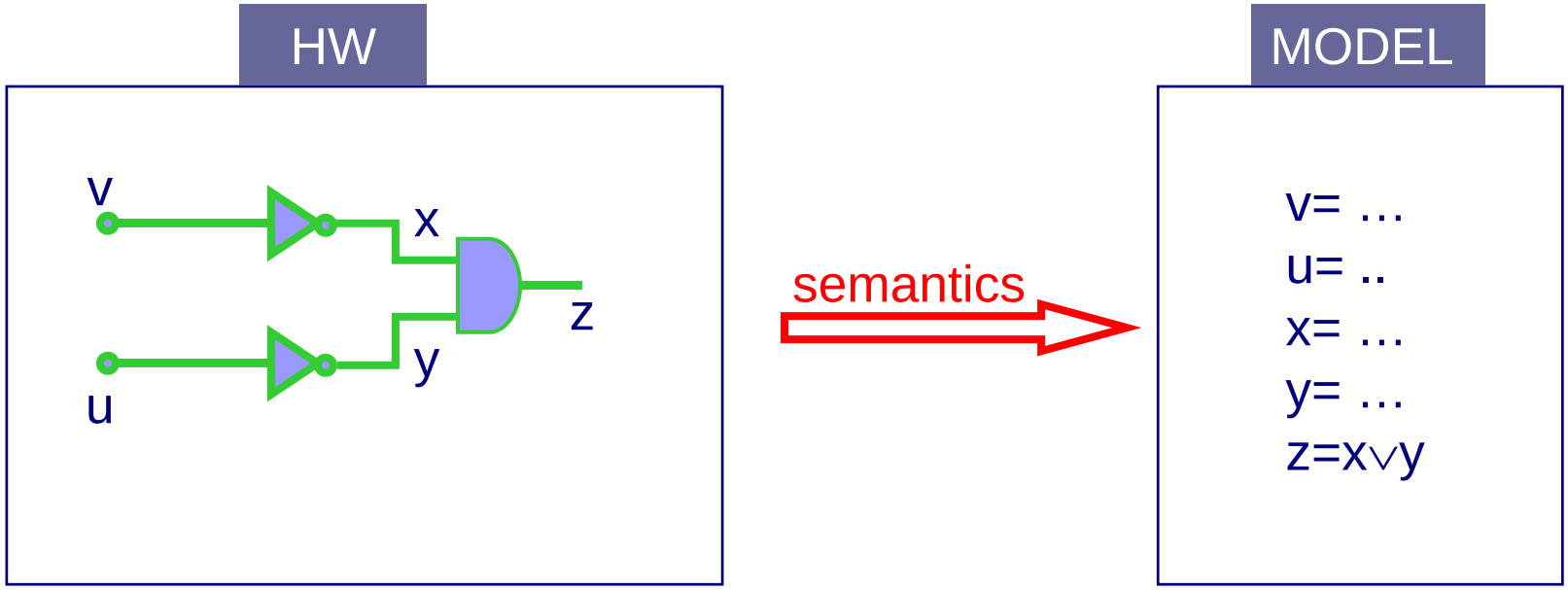
System Design – Verification



Verification techniques are monolithic and highly costly to apply: ~ $1,000 per line of code for "high-assurance" software!

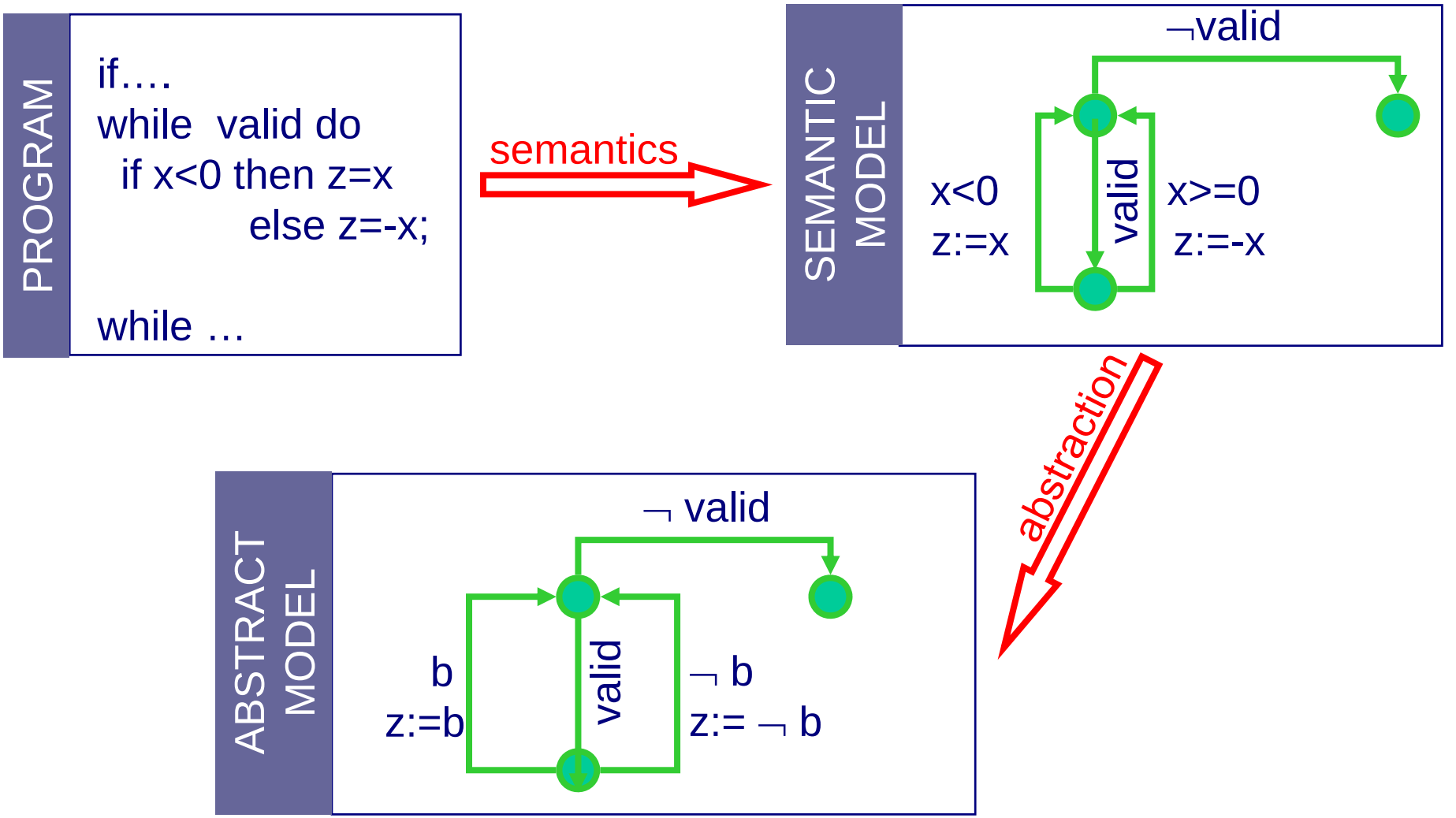
System Design – Verification: Building models

For hardware, it is easy to get faithful logical finite state models represented as systems of boolean equations



System Design – Verification: Building models

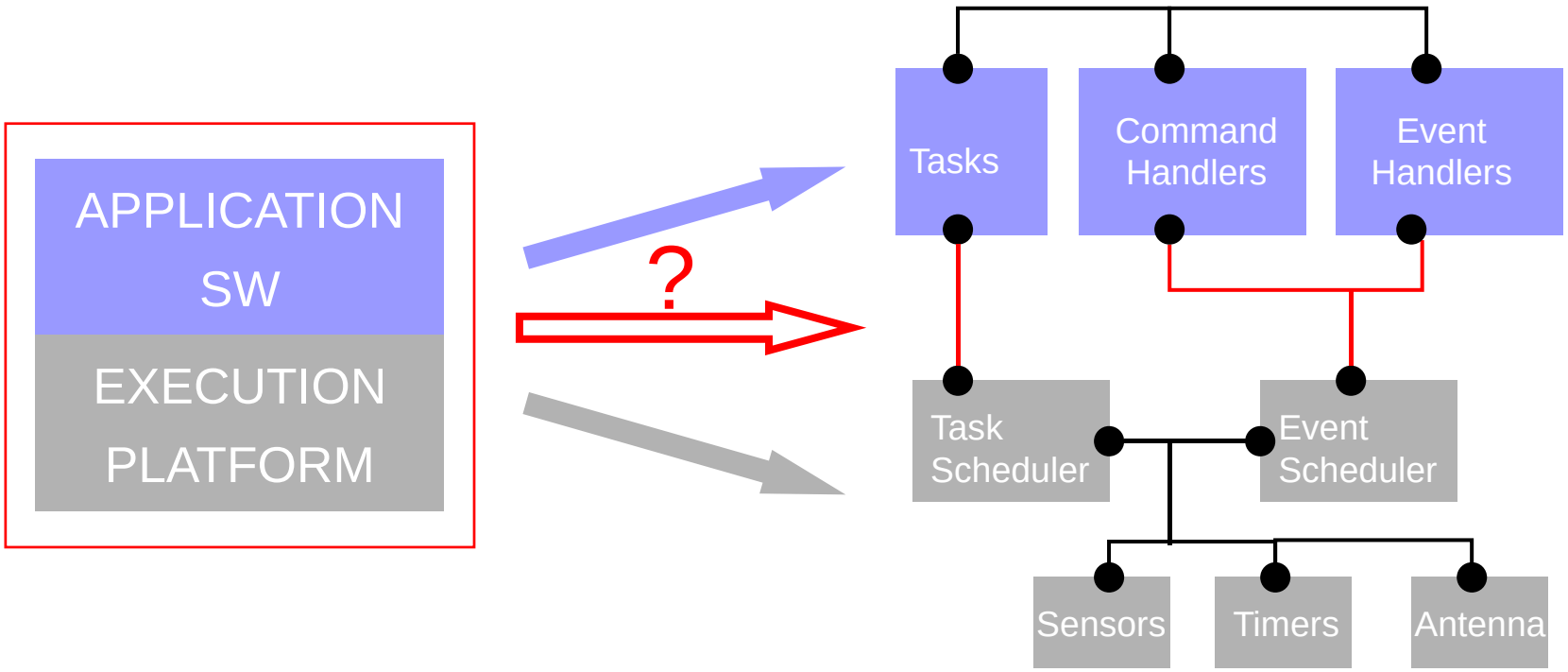
For software this may be much harder ….



System Design – Verification: Building models

For mixed Software/Hardware systems:

- there are no faithful modeling techniques as we have a poor understanding of how **software** and the underlying **platform** interact
- validation by testing physical prototypes or by simulation of ad hoc models



OVERVIEW

- System Design
- Rigorous System Design
  - Separation of Concerns
  - Component-based Design
  - Semantically Coherent Design
  - Correct-by-construction Design
- Discussion

Rigorous System Design – The Concept

RSD considers design as a formal accountable and iterative process for deriving trustworthy and optimized implementations from an application software and models of its execution platform and its external environment

- Model-based: successive system descriptions are obtained by correct-by-construction source-to-source transformations of a single expressive model rooted in well-defined semantics - The Model is the Software!
- Accountable: possibility to assert which among the requirements are satisfied and which may not be satisfied and why

RSD focuses on mastering and understanding design as a problem solving process based on divide-and-conquer strategies involving iteration on a set of steps and clearly identifying

- points where human intervention and ingenuity are needed to resolve design choices through requirements analysis and confrontation with experimental results
- segments that can be supported by tools to automate tedious and error-prone tasks

Rigorous System Design – Four Guiding Principles

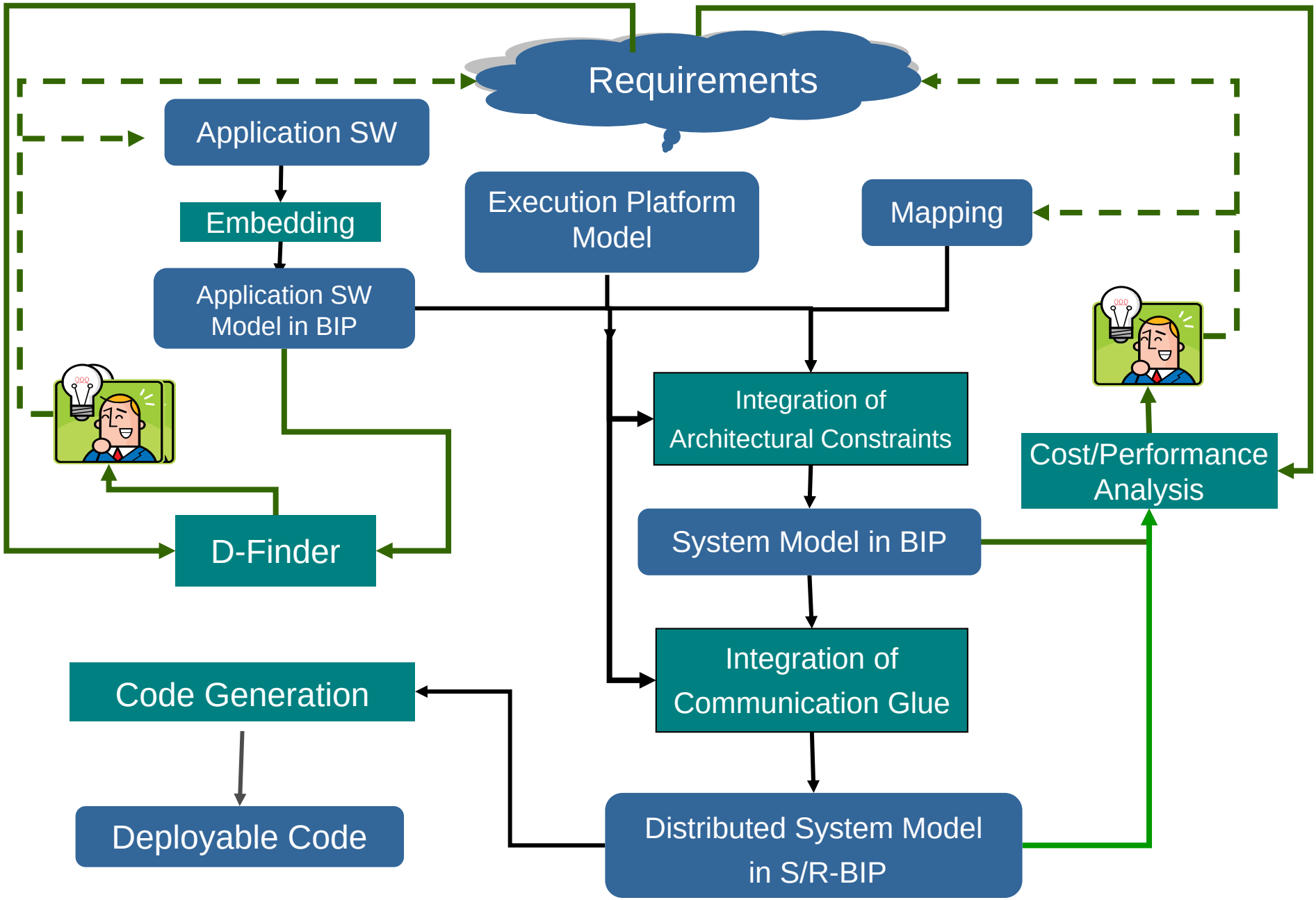
Separation of concerns: Keep separate what functionality is provided (application SW) from how its is implemented by using resources of the target platform

Components: Use components for productivity and enhanced correctness

Coherency: Based on a single model to avoid gaps between steps due to the use of semantically unrelated formalisms e.g. for programming, HW description, validation and simulation, breaking continuity of the design flow and jeopardizing its coherency

Correctness-by-construction: Overcome limitations of a posteriori verification through extensive use of provably correct reference architectures and structuring principles enforcing essential properties

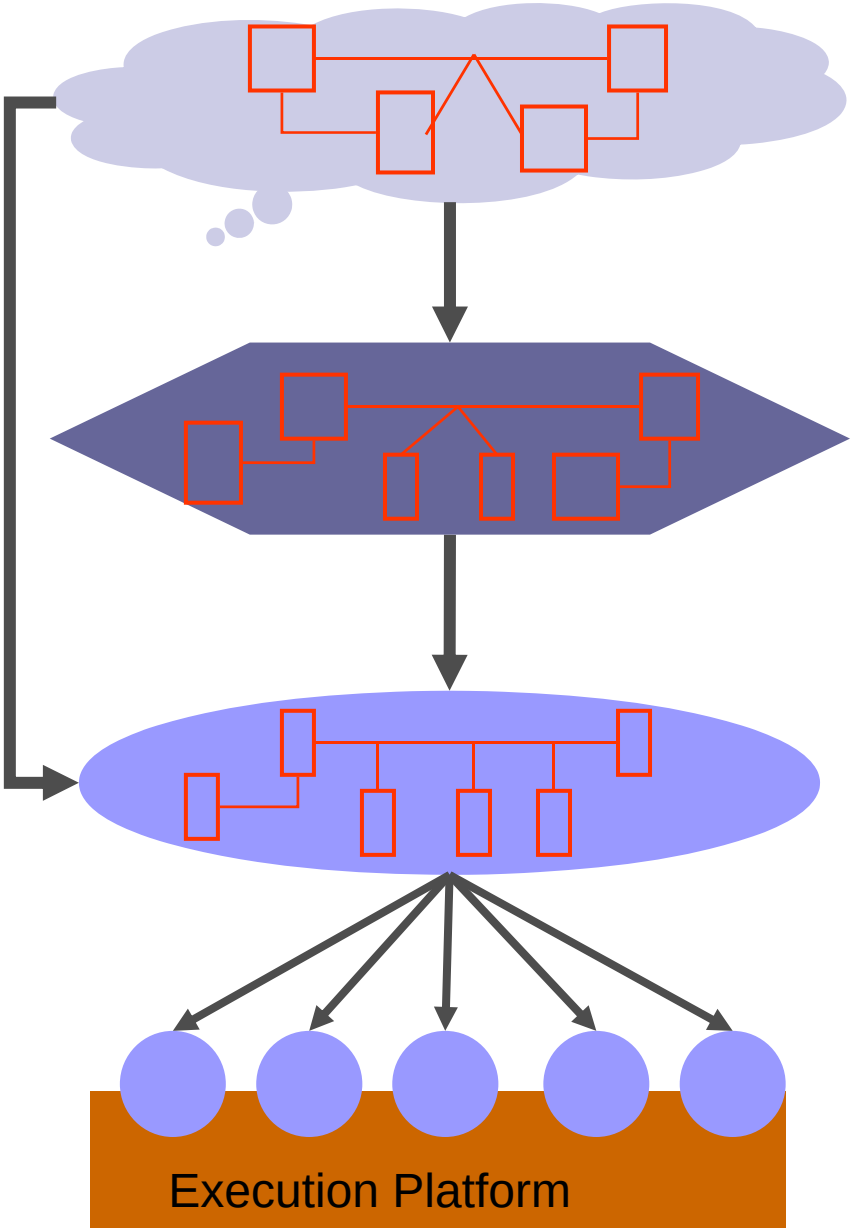
Rigorous System Design – Simplified Flow



OVERVIEW

- System Design
- Rigorous System Design
  - Separation of Concerns
  - Component-based Design
  - Semantically Coherent Design
  - Correct-by-construction Design
- Discussion

Component-based Design

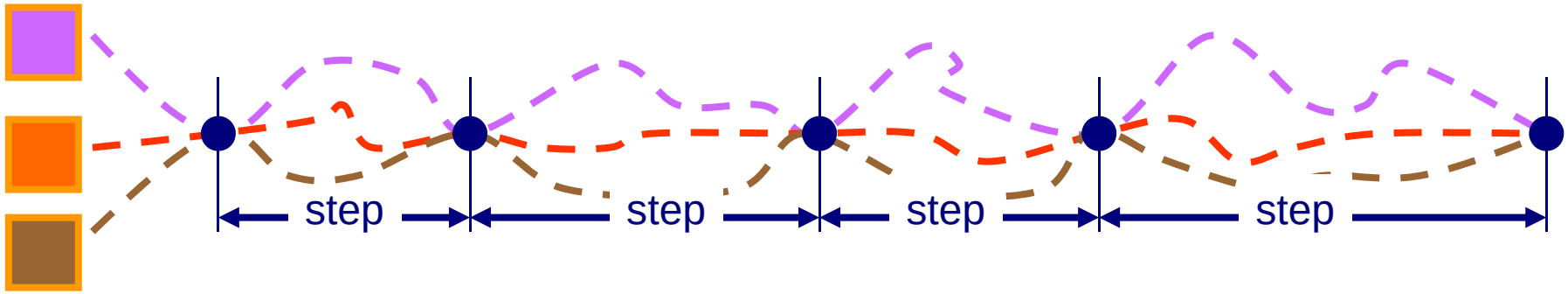


- Complex systems can be build form a relatively small number of types of components and “glue” that can be considered as a composition operator.
- Basic assumptions:
  - The behavior of a composite component can be inferred by composing the behavior of its constituents.
  - The behavior of the components is not altered when they are plugged in a given context
- We have no theory for building computing systems from components – There is no common component model for software.

Component-based Design – Synchronous vs. Asynchronous

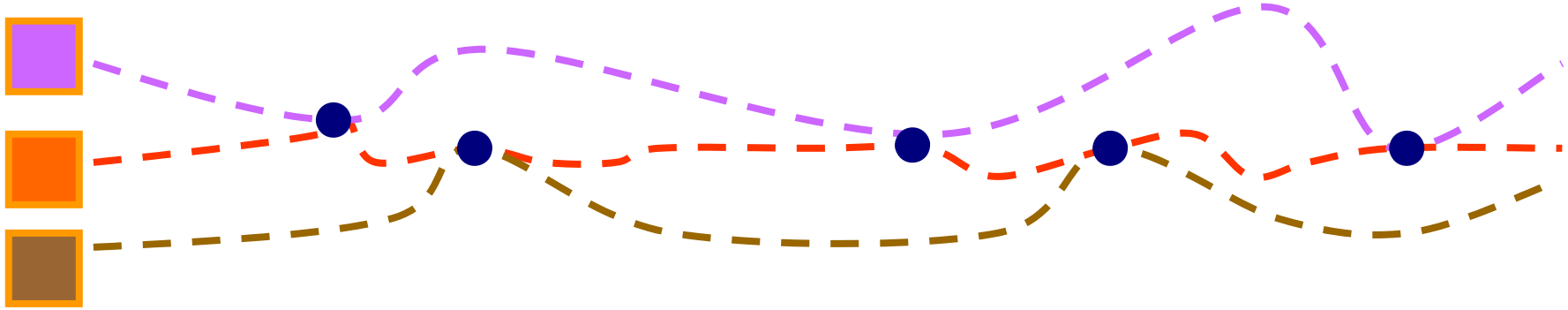
Synchronous components (HW, Multimedia application SW)

- Execution is a sequence of non interruptible steps



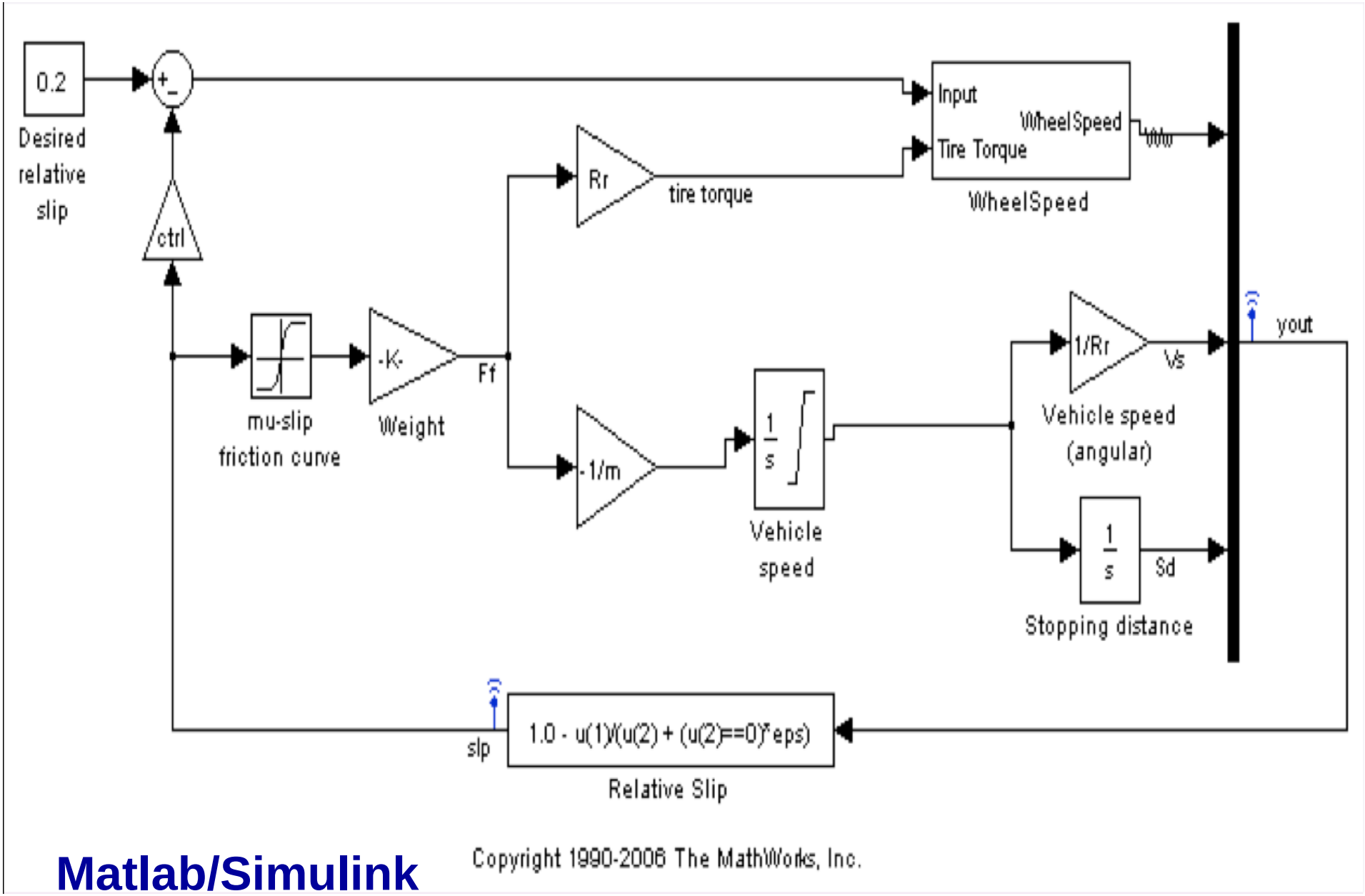
Asynchronous components (General purpose application SW)

- No predefined execution step



Open problem: Theory for consistently composing synchronous and asynchronous components e.g. GALS

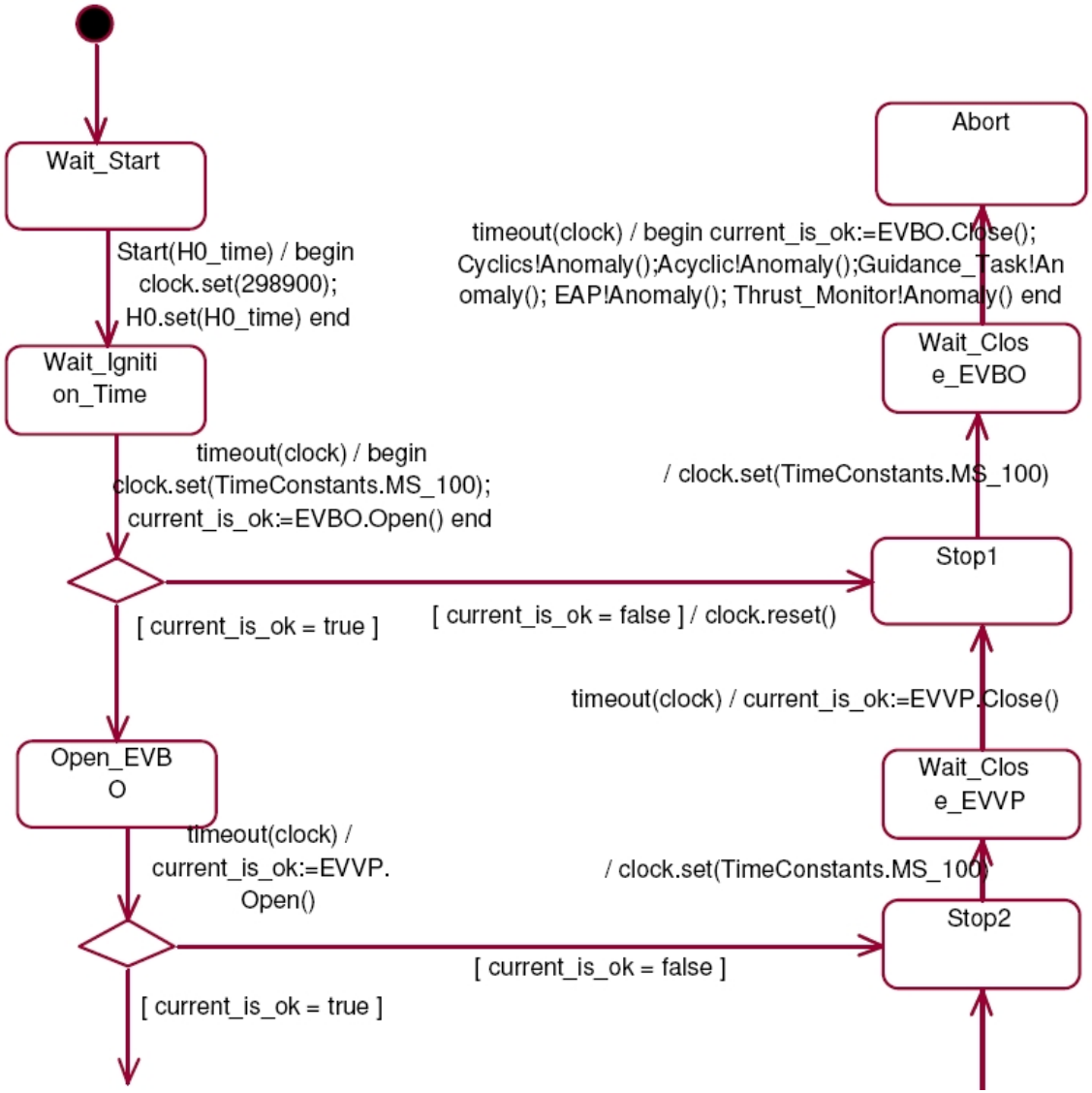
Component-based Design – Synchronous vs. Asynchronous



Matlab/Simulink

Component-based Design – Synchronous vs. Asynchronous

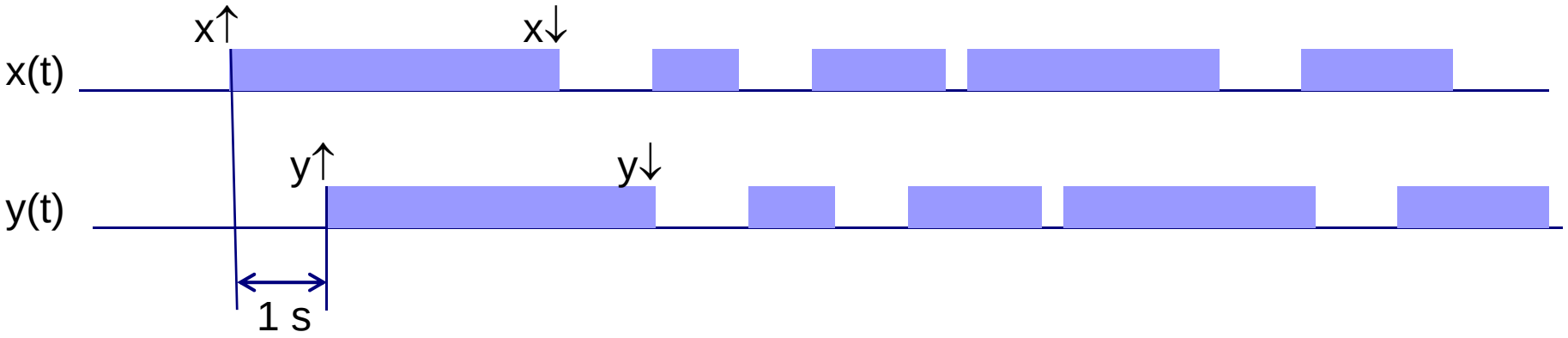
UML Model
(Rational Rose)

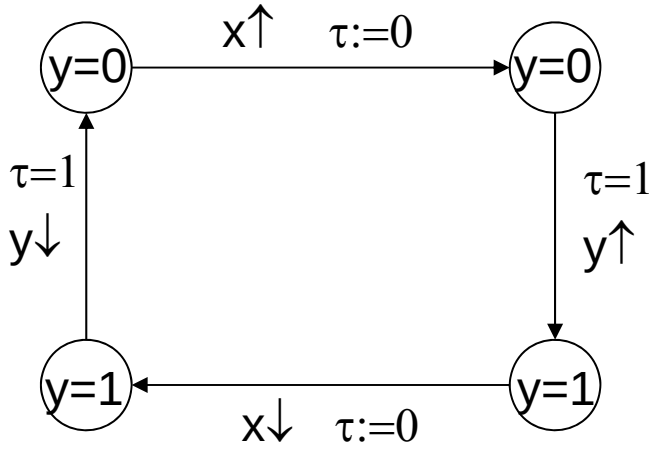


Component-based Design – Synchronous vs. Asynchronous

Mathematically simple does not imply computationally simple!

There is no finite state computational model equivalent to a unit delay!

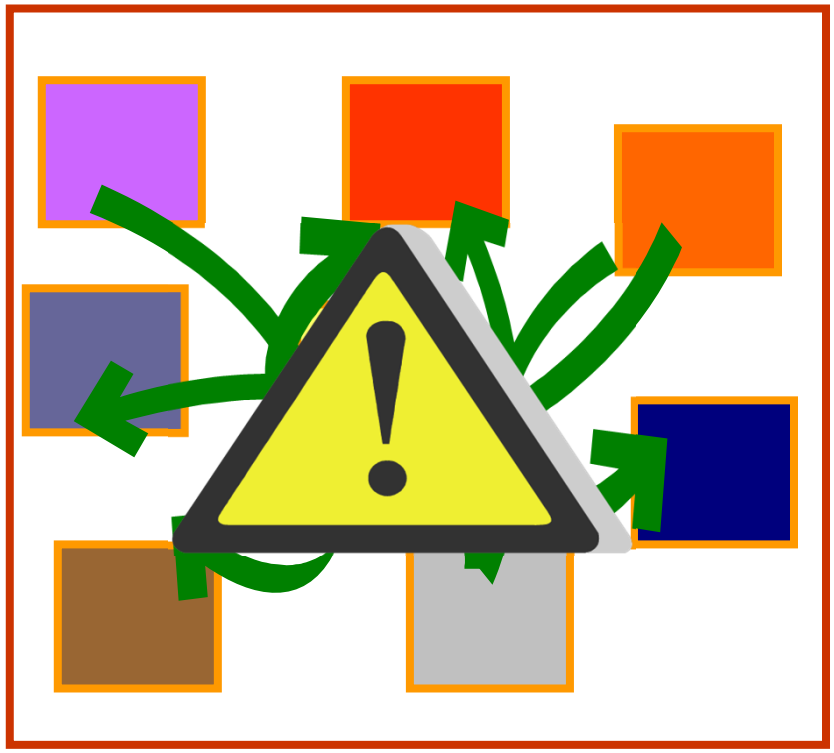




Equivalent timed automaton, provided that the distance between two consecutive input changes is more than 1s.

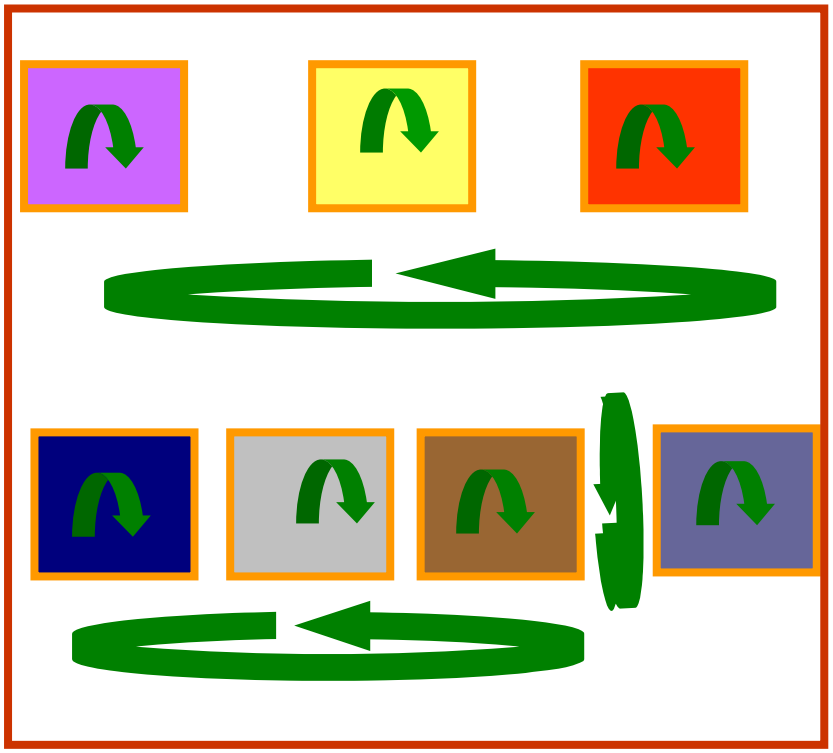
Component-based Design – Programming Styles

Thread-based programming



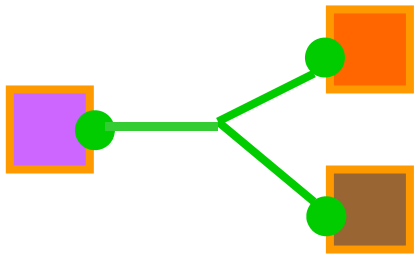
Software Engineering

Actor-based programming

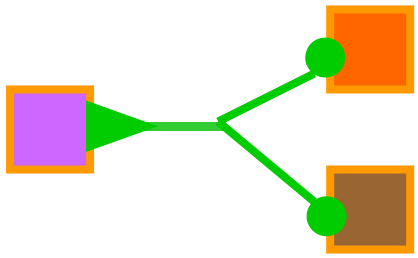


Systems Engineering

Component-based Design – Interaction Mechanisms



Rendezvous: atomic symmetric synchronization



Broadcast: asymmetric synchronization triggered by a Sender

Existing formalisms and theories are not expressive enough

- use variety of low-level coordination mechanisms including semaphores, monitors, message passing, function call
- encompass point-to-point interaction rather than multiparty interaction

Component-based Design – Composition

- Is it possible to express component coordination in terms of composition operators?

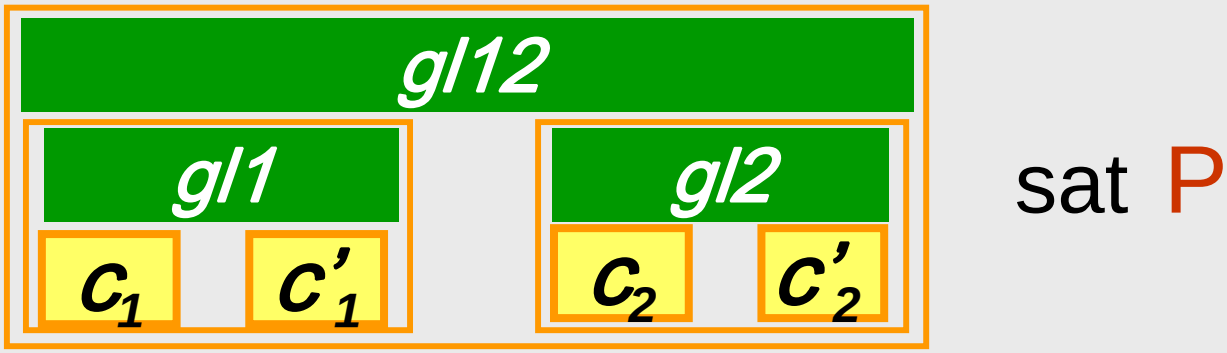
  We need a unified composition paradigm for describing and analyzing the coordination between components in terms of tangible, well-founded and organized concepts and characterized by
  - Orthogonality: clear separation between behavior and coordination constraints
  - Minimality: uses a minimal set of primitives
  - Expressiveness: achievement of a given coordination with a minimum of mechanism and a maximum of clarity

- Most component composition frameworks fail to meet these requirements
  - Process algebras e.g. CCS, CSP, pi-calculus do not distinguish between behavior and coordination
  - Most Architecture Description Languages (ADL) are ad hoc and lack rigorous semantics.

Component-based Design – The Concept of Glue

Build a component *C* satisfying a given property *P*, from

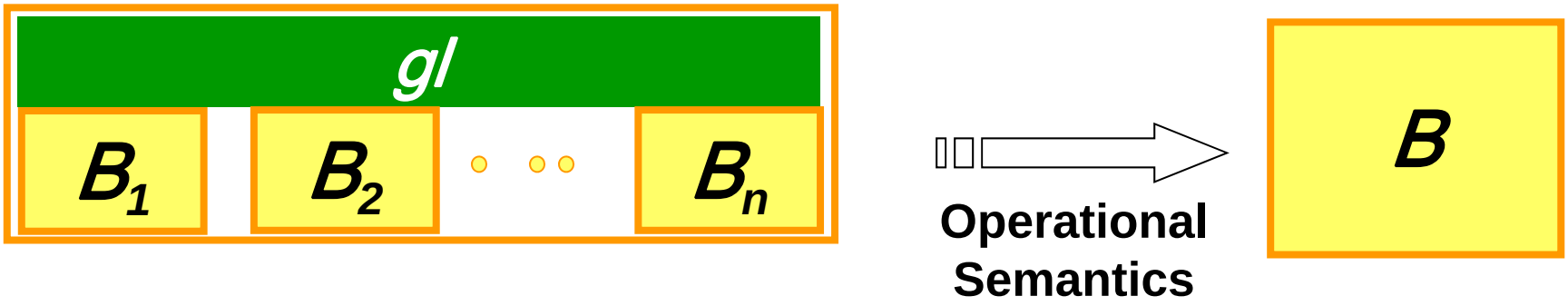
- $\mathcal{C}_0$ a set of **atomic** components described by their behavior
- $\mathcal{GL} = \{gl_1, \ldots, gl_i, \ldots\}$ a set of **glue *operators*** on components



Glue operators are stateless – separation of concerns between behavior and coordination

Component-based Design – Glue Operators

We use operational semantics to define the meaning of a composite component – glue operators are “behavior transformers”



Glue Operators

- build interactions of composite components from the actions of the atomic components e.g. parallel composition operators
- can be specified by using a family of operational semantics rules (the Universal Glue)

Component-based Design – Glue Operators

A **glue operator** defines **interactions** as a set of derivation rules of the form

$$\frac{\{q_i - a_i \rightarrow_i q'_i\}_{i \in I} \qquad C(q_k)_{k \in K}}{(q_1, .., q_n) - a \rightarrow (q'_1, .., q'_n)}$$

- $I, K \subseteq \{1, \ldots n\}$, $I \neq \emptyset$, $K \cap I = \emptyset$
- $a = \bigcup_{i \in I} a_I$ is an interaction
- $q'_i = q_i$ for $i \notin I$

Notice that, non deterministic choice and sequential composition are not glue operators

A **glue** is a set of glue operators

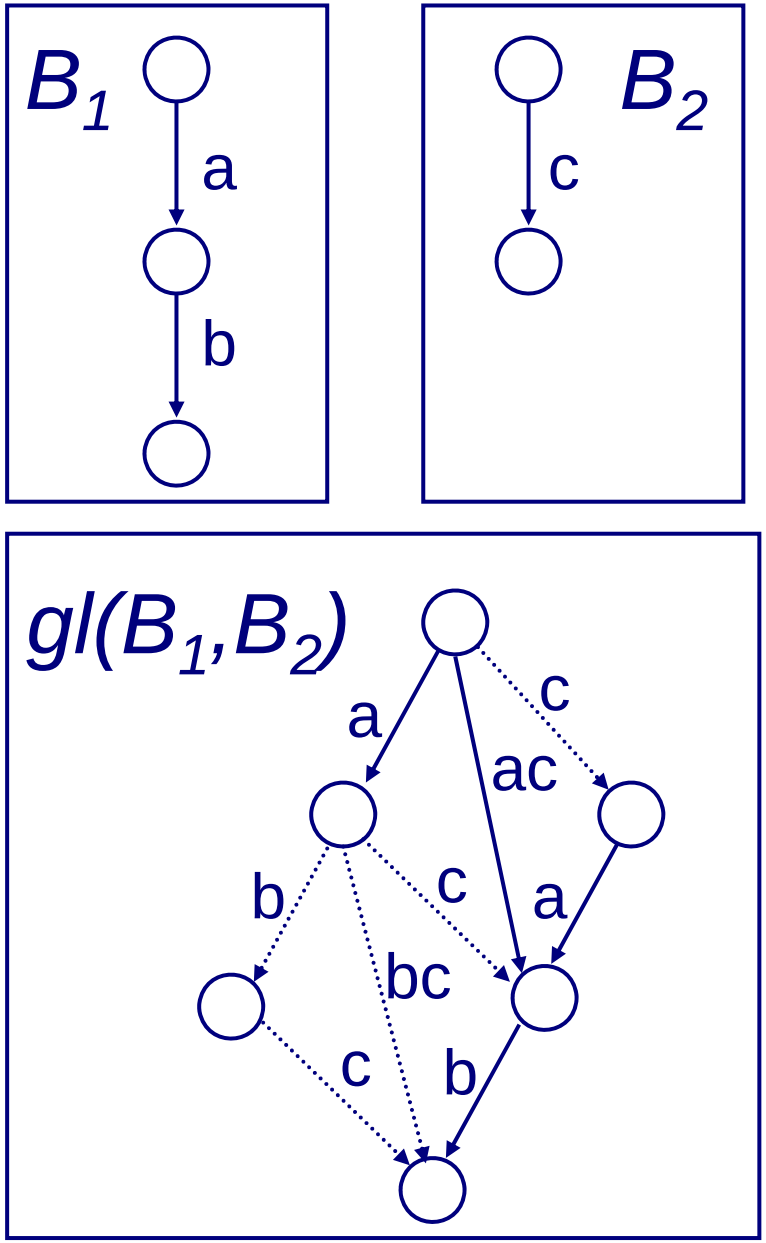
Component-based Design – Glue Operators: Example

gl is defined by

$$\frac{q_1 \text{ - } a \rightarrow q'_1}{q_1 q_2 \text{ - } a \rightarrow q'_1 q_2}$$

$$\frac{q_1 \text{ - } a \rightarrow q'_1 \quad q_2 \text{ - } c \rightarrow q'_2}{q_1 q_2 \text{ - } ac \rightarrow q'_1 q'_2}$$

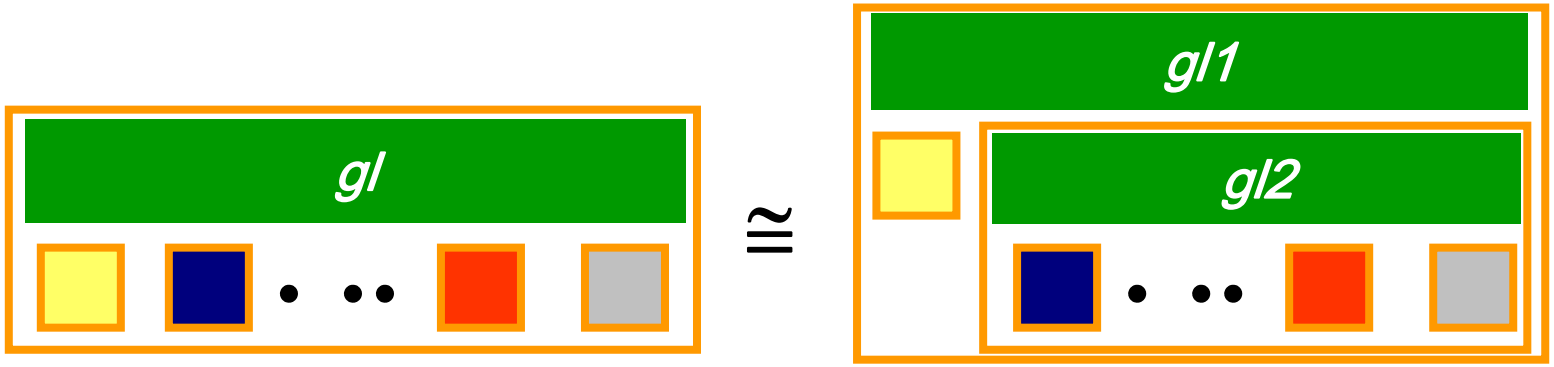
$$\frac{q_1 \text{ - } b \rightarrow q'_1 \quad \neg\, q_2 \text{ - } c \rightarrow}{q_1 q_2 \text{ - } b \rightarrow q'_1 q_2}$$



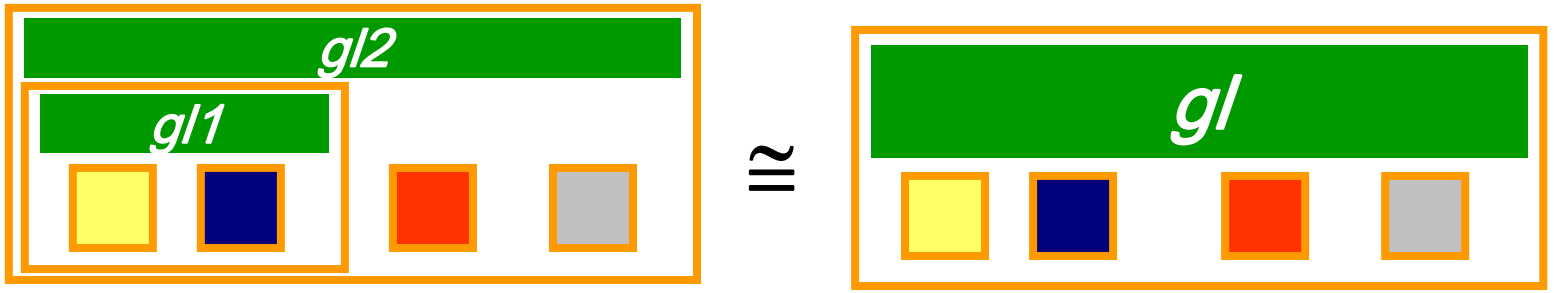
Component-based Design – Glue Operators: Properties

Glue is a first class entity independent from behavior that can be decomposed and composed

1. Incrementality



2. Flattening



Component-based Design – Glue Operators: Expressiveness

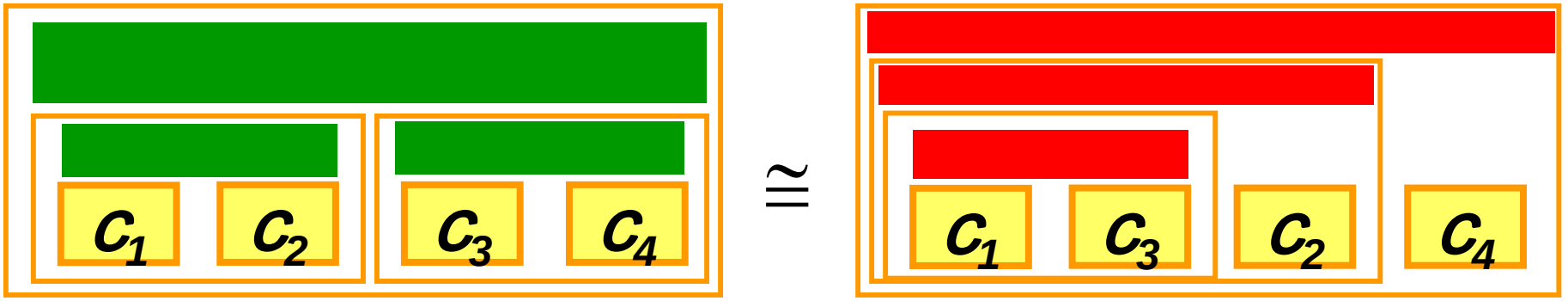
- Different from the usual notion of expressiveness!
- Based on strict separation between glue and behavior

Given two glues G_1 , G_2

G_2 ***is strongly more expressive than*** G_1

if for any component built by using G_1 and a set of components $\mathscr{C}_0$

there exists an equivalent component built by using G_2 and $\mathscr{C}_0$



Component-based Design – Glue Operators: Expressiveness

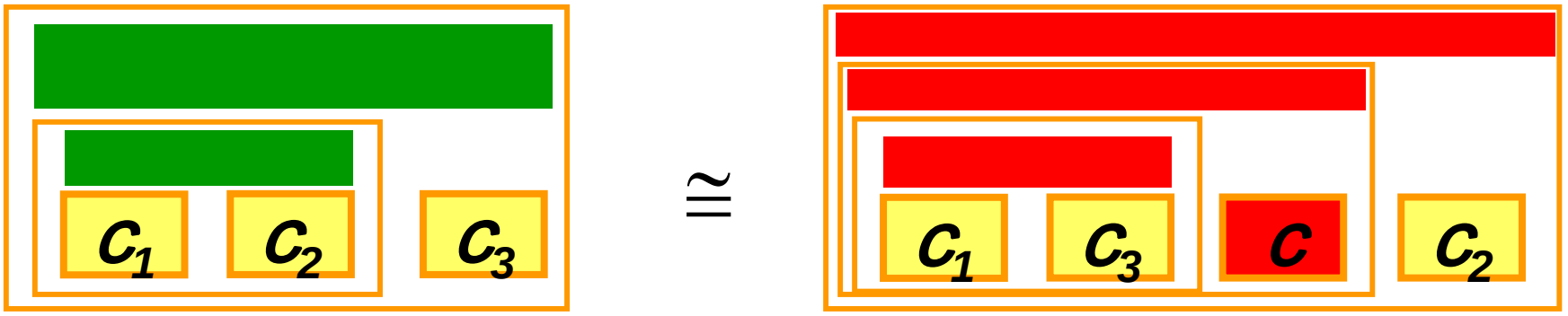
Given two glues G_1 , G_2

G_2 ***is weakly more expressive than*** G_1

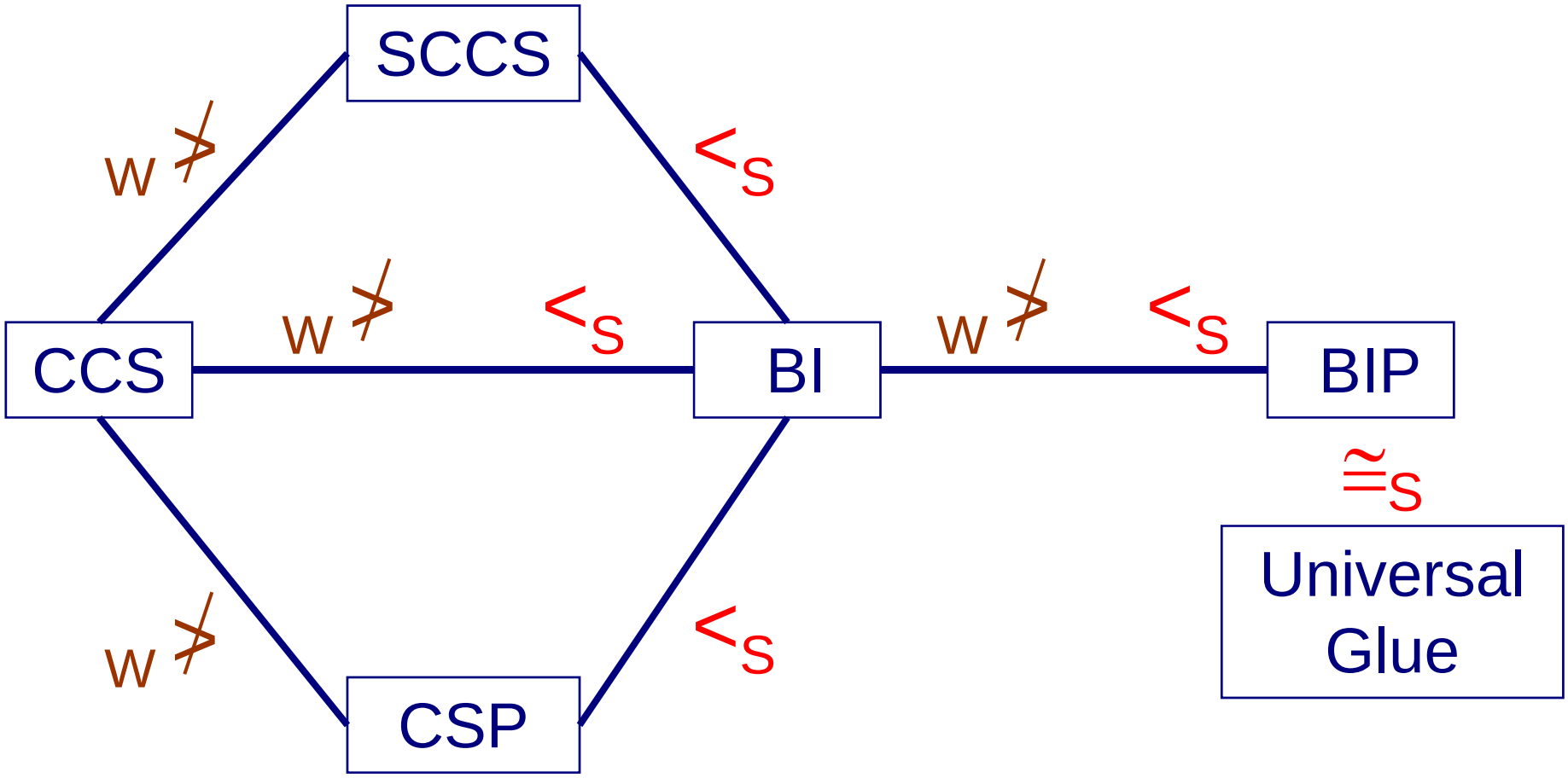
if for any component built by using G_1 and a set of components $\mathcal{C}_0$

there exists an equivalent component built by using G_2 and $\mathcal{C}_0 \cup \mathcal{C}$

where $\mathcal{C}$ is a finite set of coordinating components.



Component-based Design – Glue Operators: Expressiveness

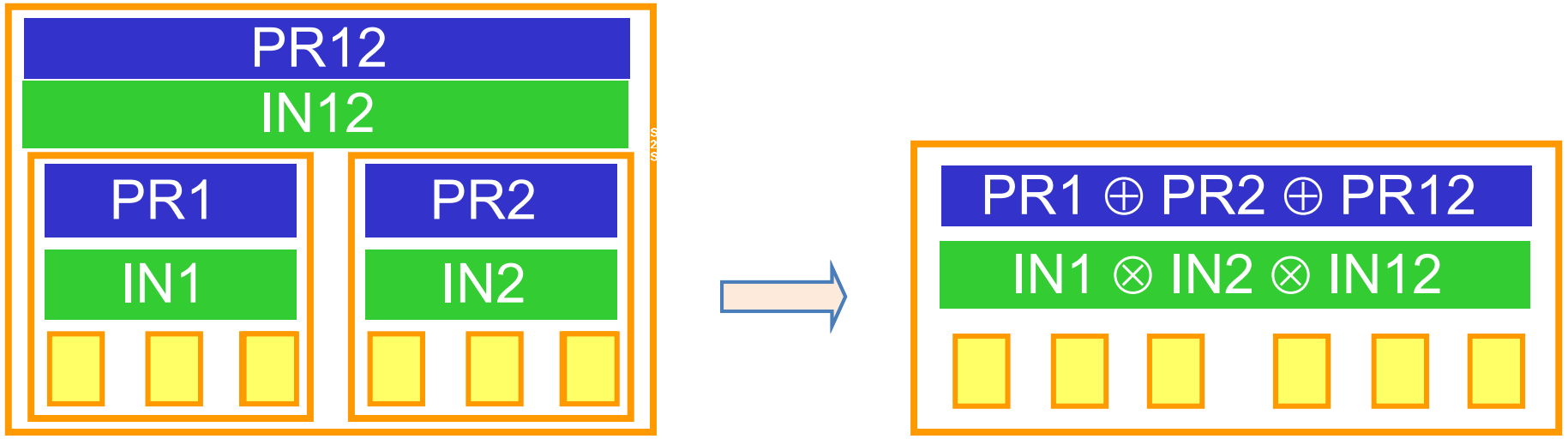


[Bliudze&Sifakis, Concur 08]

Component-based Design – Modeling in BIP

Layered component model

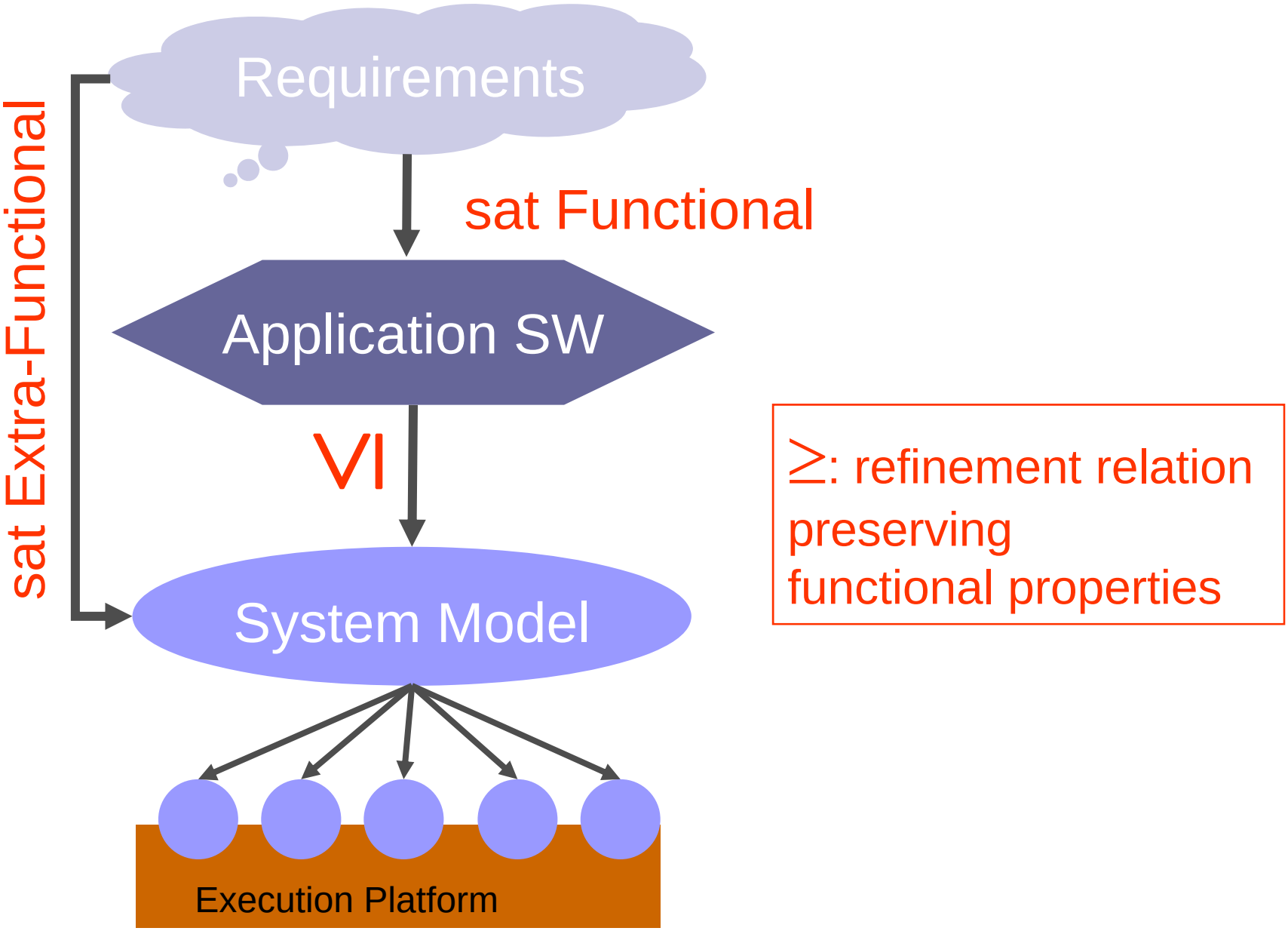
Composition operation parameterized by glue IN12, PR12



OVERVIEW

- System Design
- Rigorous System Design
  - Separation of Concerns
  - Component-based Design
  - Semantically Coherent Design
  - Correct-by-construction Design
- Discussion

Correct by Construction



Correct by Construction – Architectures

Architectures

- depict design principles, paradigms that can be understood by all, allow thinking on a higher plane and avoiding low-level mistakes

- are a means for ensuring global properties characterizing the coordination between components – correctness for free

- Using architectures is key to ensuring trustworthiness and optimization in networks, OS, middleware, HW devices etc.

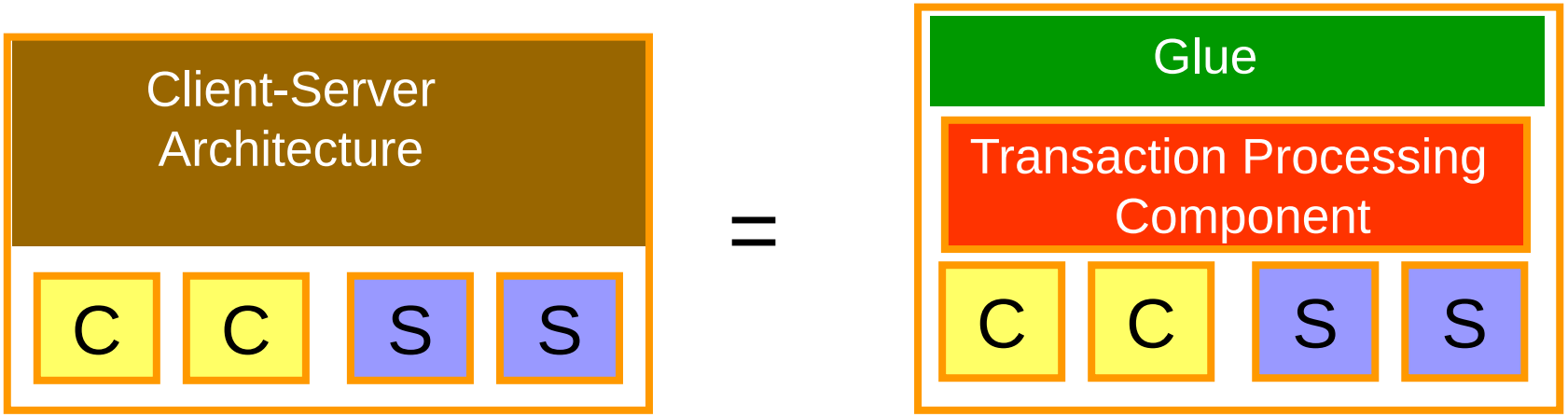
System developers extensively use libraries of standard reference architectures -- nobody ever again have to design from scratch a banking system, an avionics system, a satellite ground system, a web-based e-commerce system, or a host of other varieties of systems.

- Time-triggered architectures
- Security architectures
- Fault-tolerant architectures
- Adaptive Architectures
- SOAP-based architecture, RESTful architecture

Correct by Construction – Architecture Definition

An architecture is a family of operators A(n)[X] parameterized by their arity n and a family of characteristic properties P(n)

- A(n)[B1,..,Bn] = gl(n)(B1,..,Bn, C(n)), where C(n) is a set of coordinators
- A(n)[B1,..,Bn] meets the characteristic property P(n).



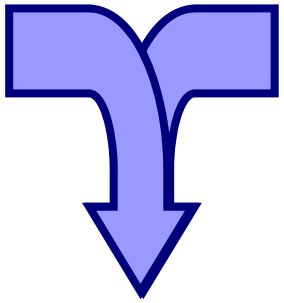
Characteristic property: atomicity of transactions, fault-tolerance ….

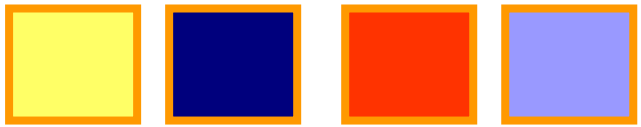
Note that the characteristic property need not be formalized!

Correct by Construction – Architectures

Rule1: Property Enforcement

Architecture
for Mutual Exclusion



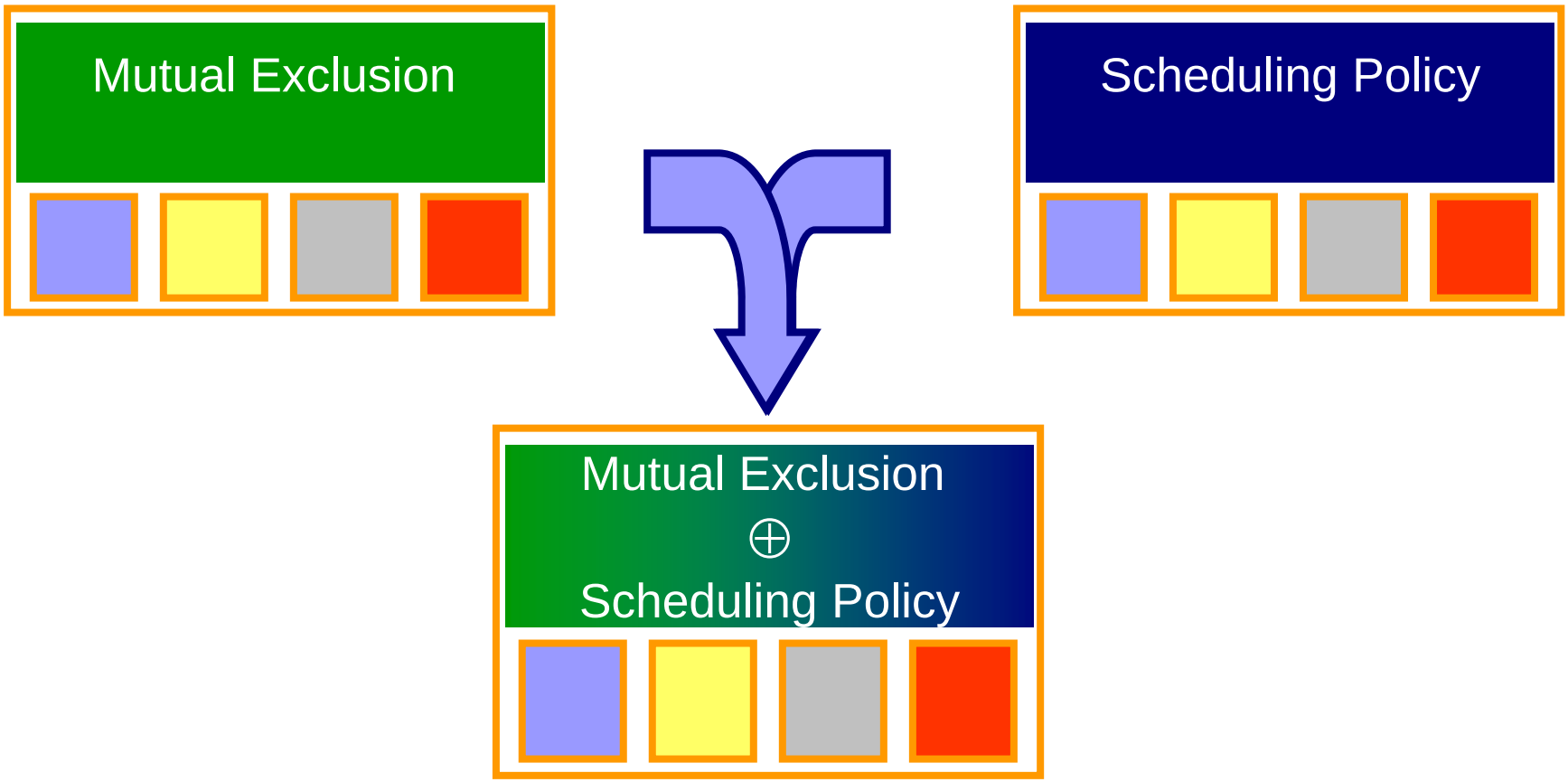


Components

satisfies Mutex

Correct by Construction – Architectures: Composability

Rule2: Property Composability



Feature interaction in telecommunication systems, interference among web services and interference in aspect programming are all manifestations of lack of composability

Sifakis et al "A General Framework for Architecture Composability" SEFM 2014

Components – Correctness-by-Construction

Fully customizable smartphones *: The design for Project Ara consists of what we call an endoskeleton (endo) and modules. The endo is the structural frame that holds all the modules in place. A module can be anything, from a new application processor to a new display or keyboard, an extra battery, a pulse oximeter or something not yet thought of!*

TECH INNOVATION

Project Ara: Inside Google's Bold Gambit to Make Smartphones Modular

Tech Guru Daily

WILL PROJECT ARA WORK? THE GOOD AND BAD OF GOOGLE'S PLAN TO TURN PHONES INTO LEGOS

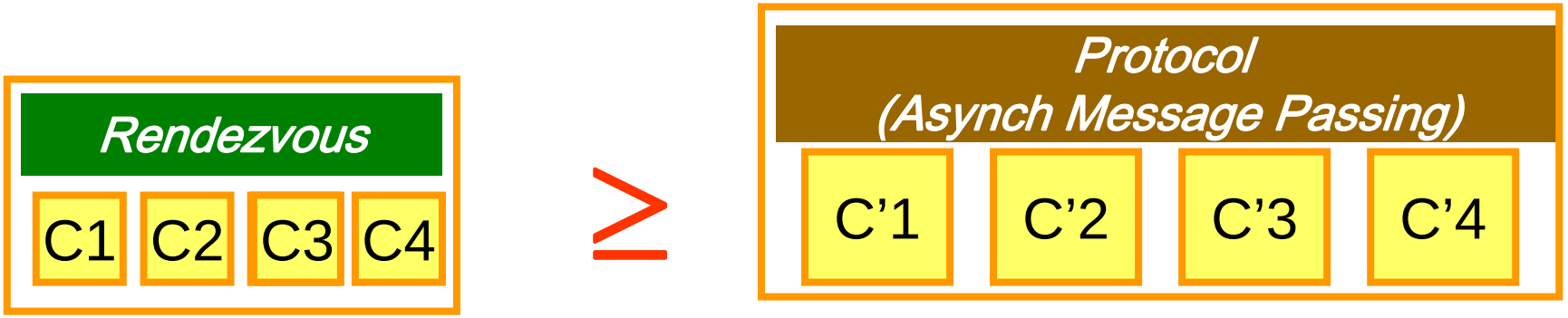
By **Simon Hill** — April 26, 2014

Debated: Google's Project Ara Will Likely Fail

May 2, 2014 - 00:31 by Rob Enderle

Correct by Construction – Refinement

The Refinement Relation ≥



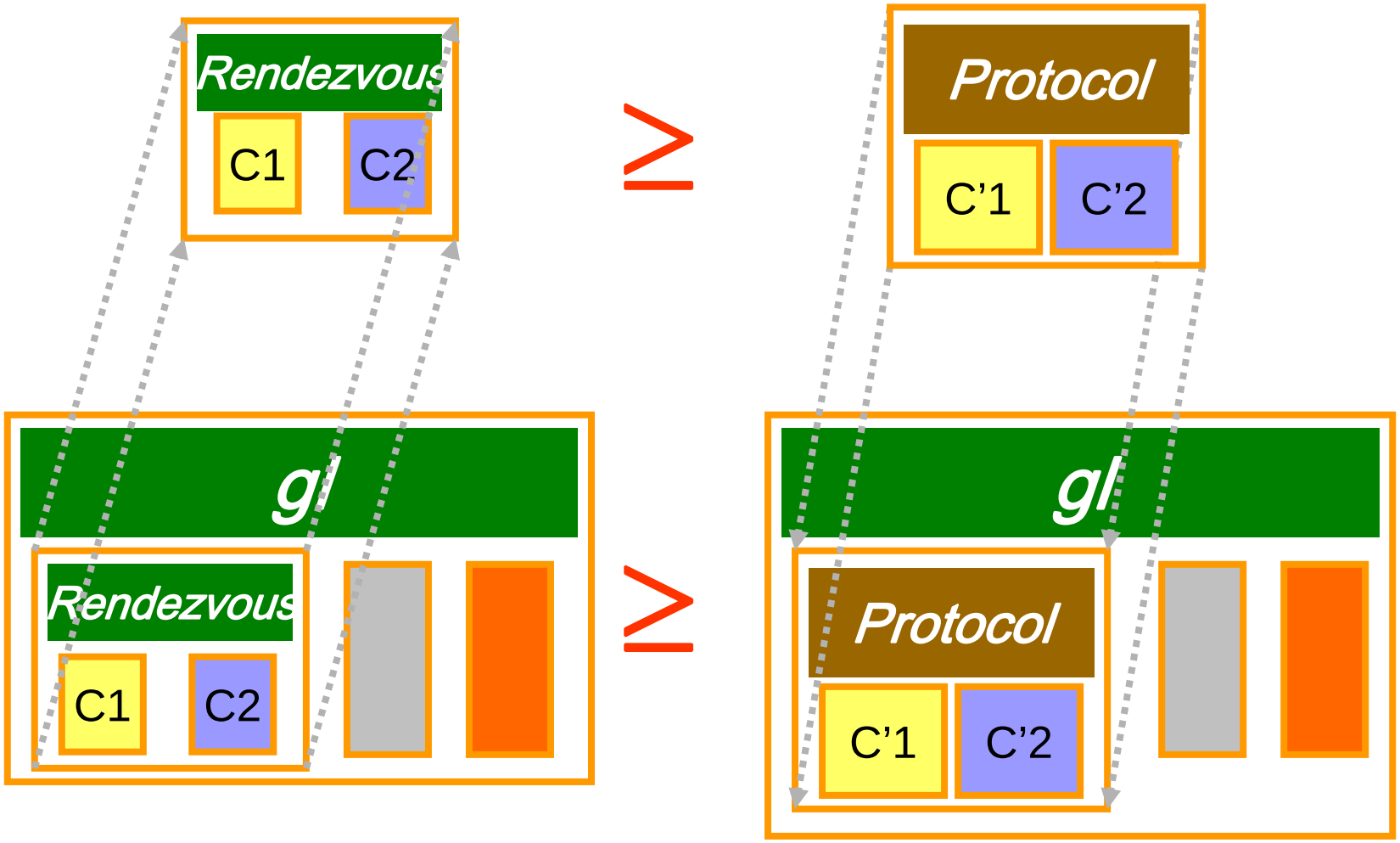
S1 S2

S1 ≥ S2 (S2 refines S1) if

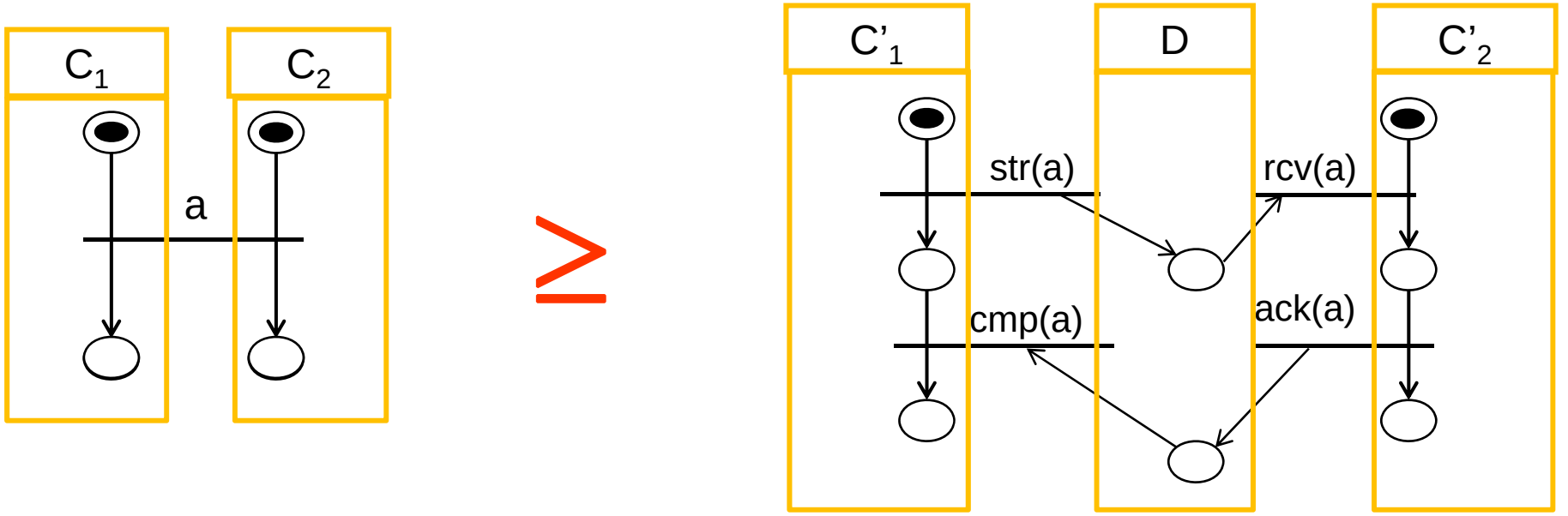
- all traces of S2 are traces of S1(modulo some observation criterion)
- if S1 is deadlock-free then S2 is deadlock-free too
- ≥ is preserved by substitution

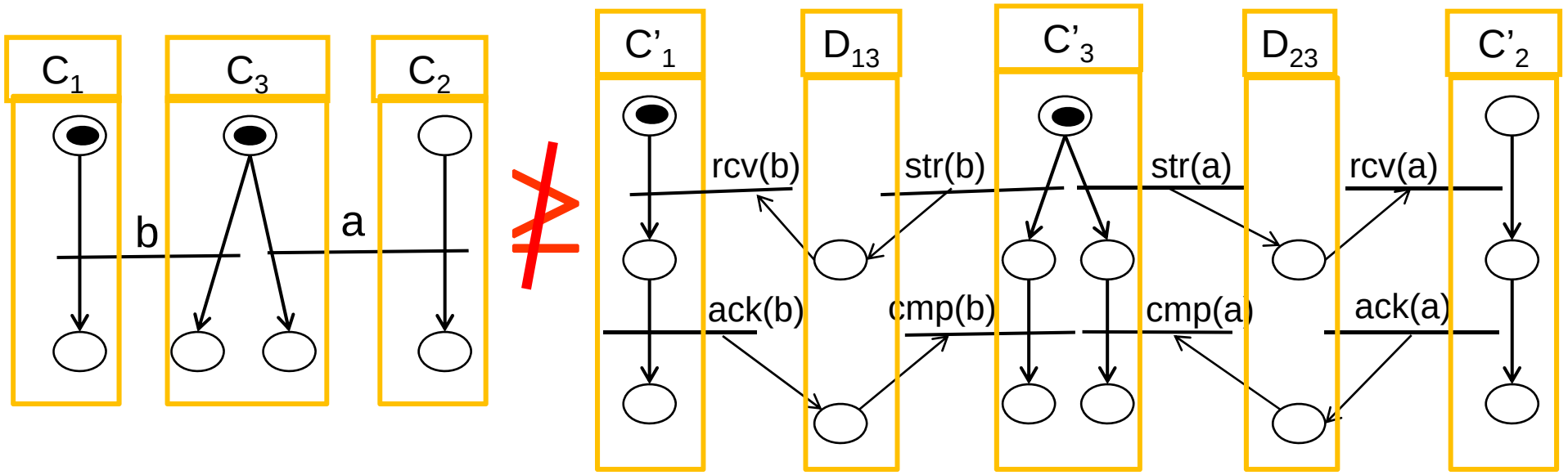
Correct by Construction – Refinement

Preservation of ≥ by substitution

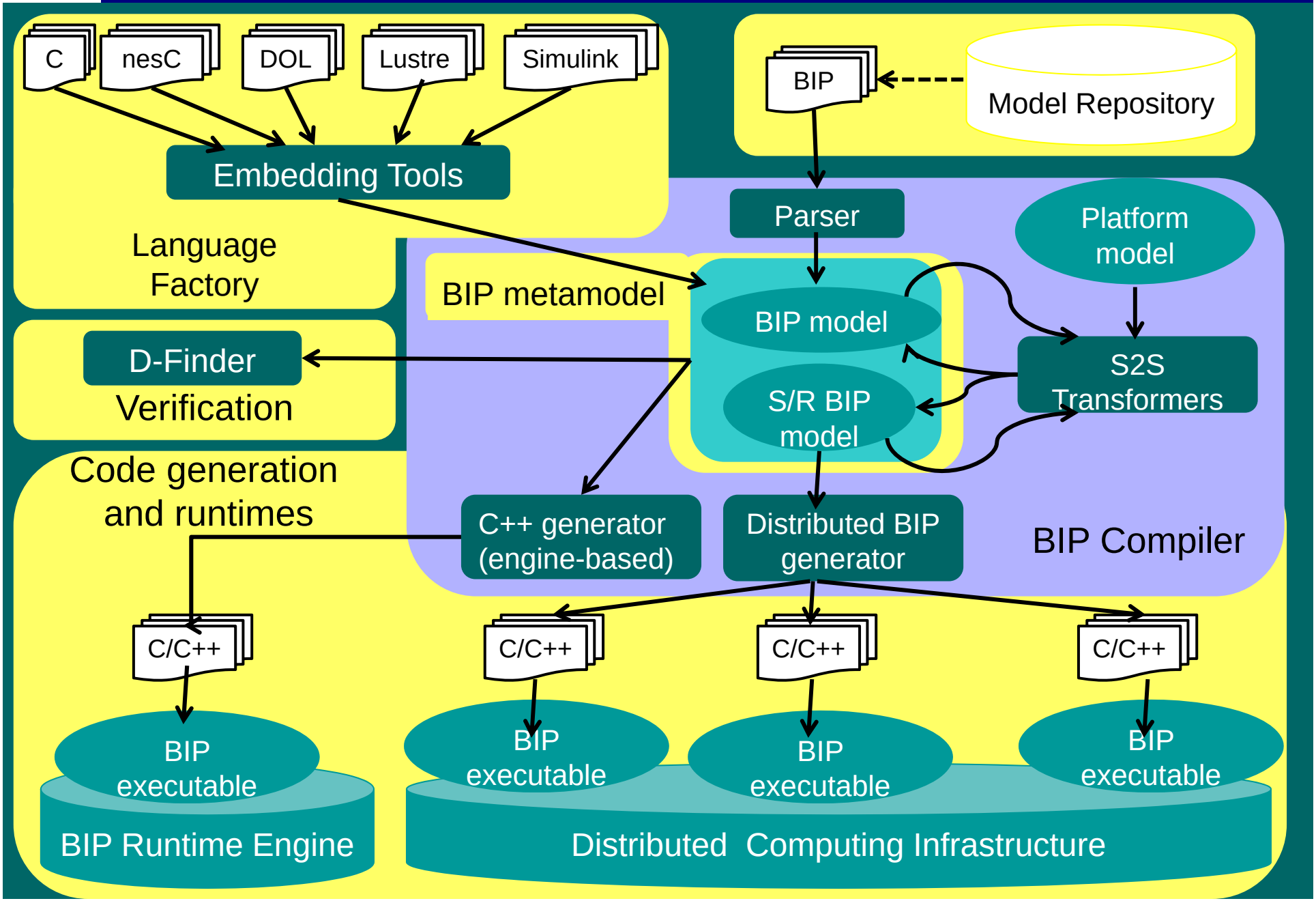


Correct by Construction – Refinement Preservation





Correct by Construction – The BIP Toolset



OVERVIEW

- System Design
- Rigorous System Design
  - Separation of Concerns
  - Component-based Design
  - Semantically Coherent Design
  - Correct-by-construction Design
- Discussion

Discussion – The Way Forward

Design formalization raises a multitude of deep theoretical problems related to the conceptualization of needs in a given area and their effective transformation into correct artifacts. Two key issues are

Languages: Move from thread-based programming to actor-based programming for component-based systems

- as close as possible to the declarative style so as to simplify reasoning and relegate software generation to tools
- supporting synchronous and asynchronous execution as well as the main programming paradigms
- allowing description of architectures and high-level coordination mechanisms

Constructivity: There is a huge body of not yet well-formalized solutions to problems in the form of algorithms, protocols, hardware and software architectures. The challenge is to

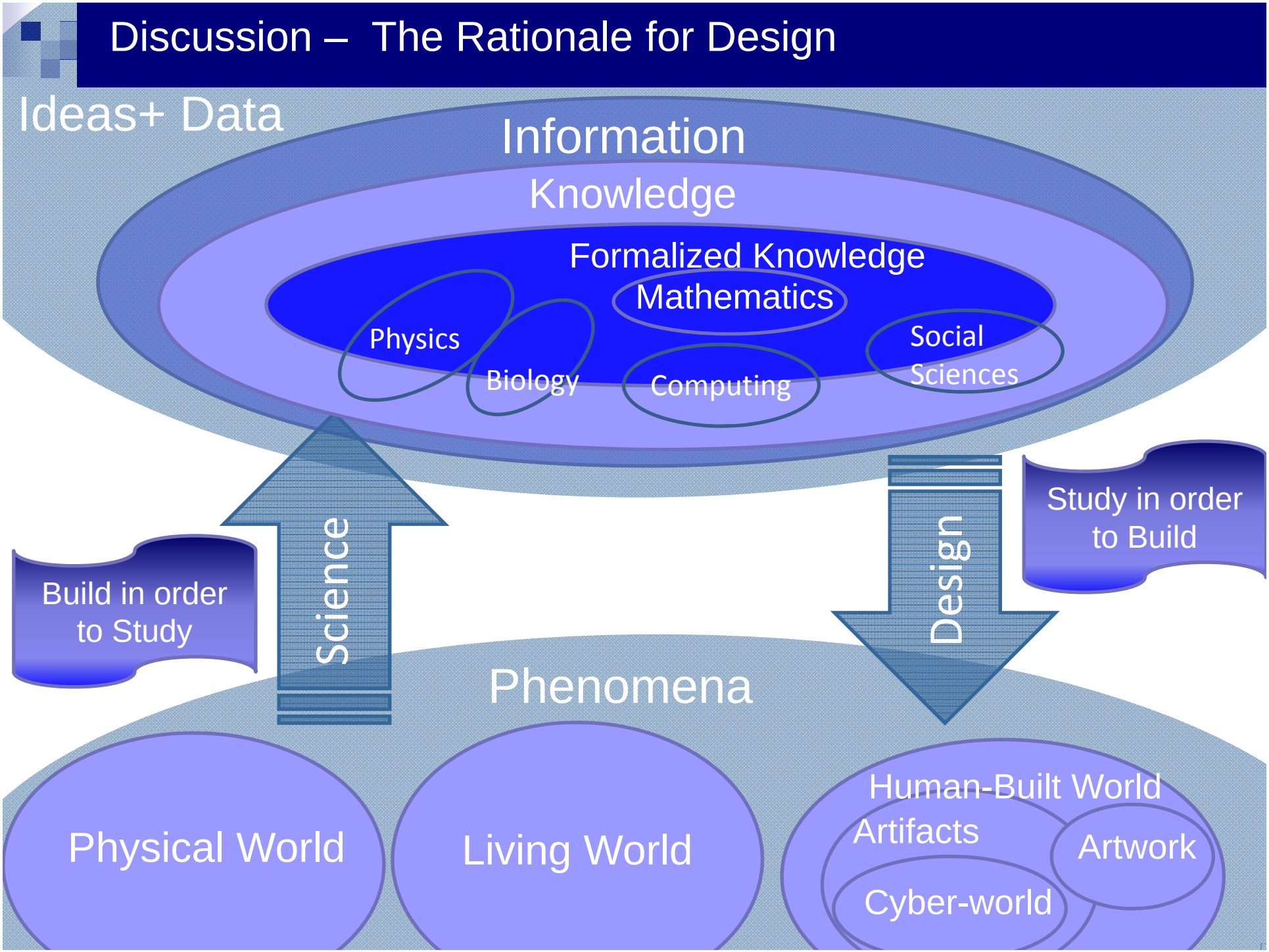
- formalize these solutions as architectures and prove their correctness
- provide a taxonomy of the architectures and their characteristic properties
- decompose any coordination property as the conjunction of predefined characteristic properties enforced by predefined architectures

Discussion – Why Is It So Hard?

The Physics Hierarchy	The Computing Hierarchy	The Bio-Hierarchy
The Universe	The Cyber-world	Ecosystem
Galaxy	Networked System	Organism
Solar System	Reactive System	Organ
Electro-mechanical System	Virtual Machine	Tissue
Crystals-Fluids-Gases	Instruction Set Architecture	Cell
Molecules	Register Transfer Level	Protein and RNA networks
Atoms	Logical Gate	Protein and RNA
Particles	Transistor	Genes

We need theory, methods and tools for climbing
up-and-down abstraction hierarchies

Discussion – The Rationale for Design



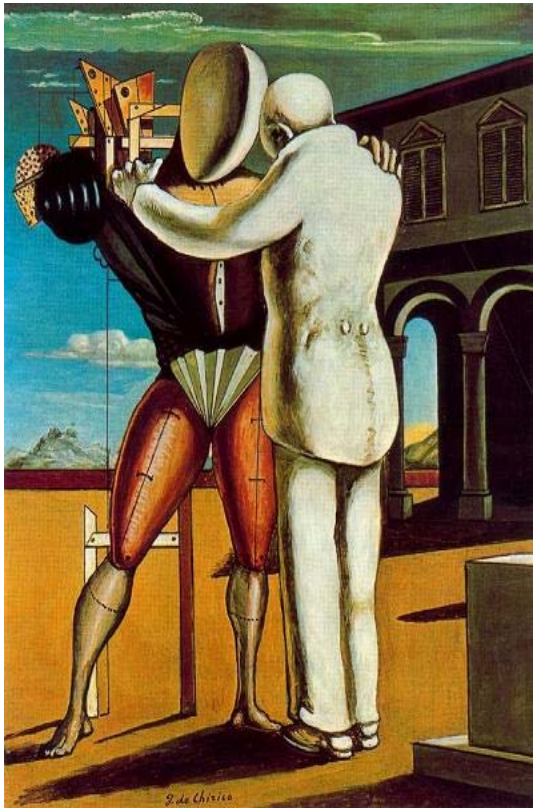
Discussion – For a System Design Discipline

Achieving this goal for systems engineering is both an intellectually challenging and culturally enlightening endeavor – it nicely complements the quest for scientific discovery in natural sciences

Failure in this endeavor would

- seriously limit our capability to master the techno-structure

- also mean that designing is a definitely a-scientific activity driven by predominant subjective factors that preclude rational treatment

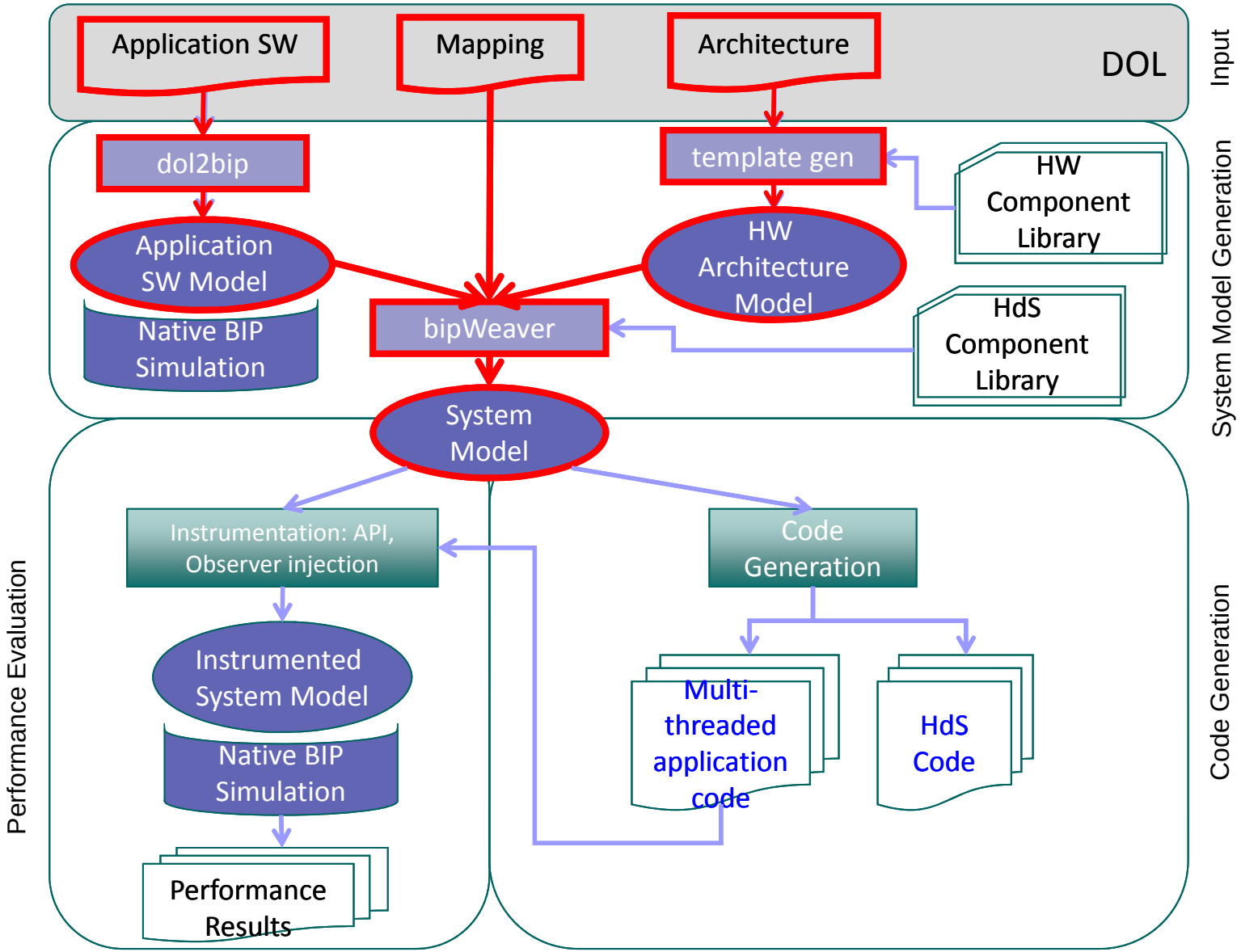


Is everything for the best in the best of all possible cyber-worlds ?
- The toughest uphill battles are still in front of us

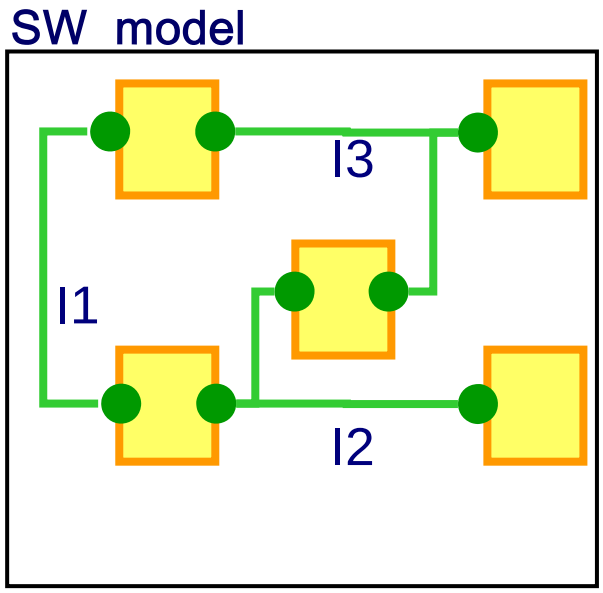
Discussion

Joseph Sifakis (2013), "Rigorous System Design",
Foundations and Trends® in Electronic Design Automation: Vol. 6: No. 4, pp 293-362.

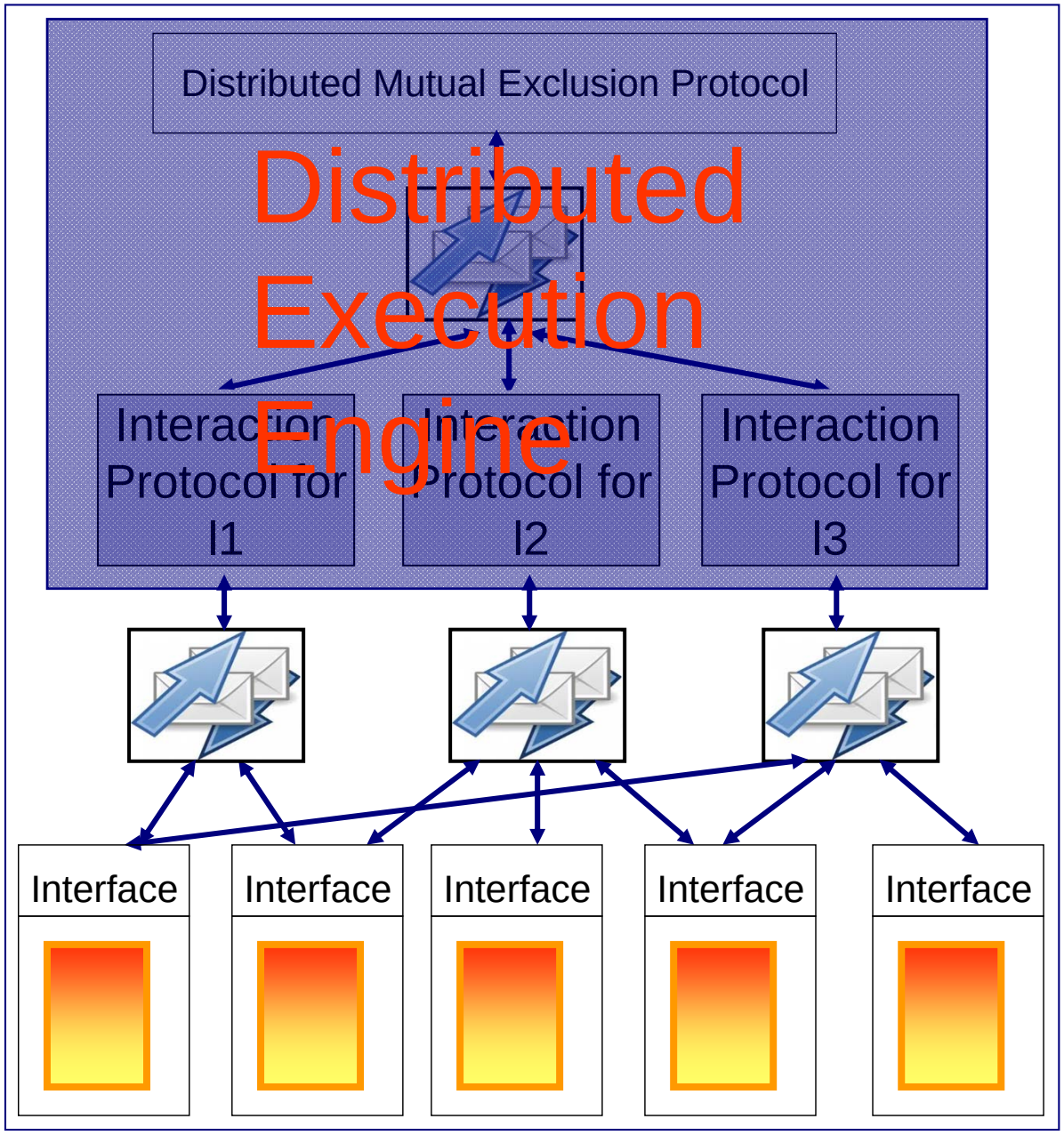
Correct by Construction – HW-driven refinement



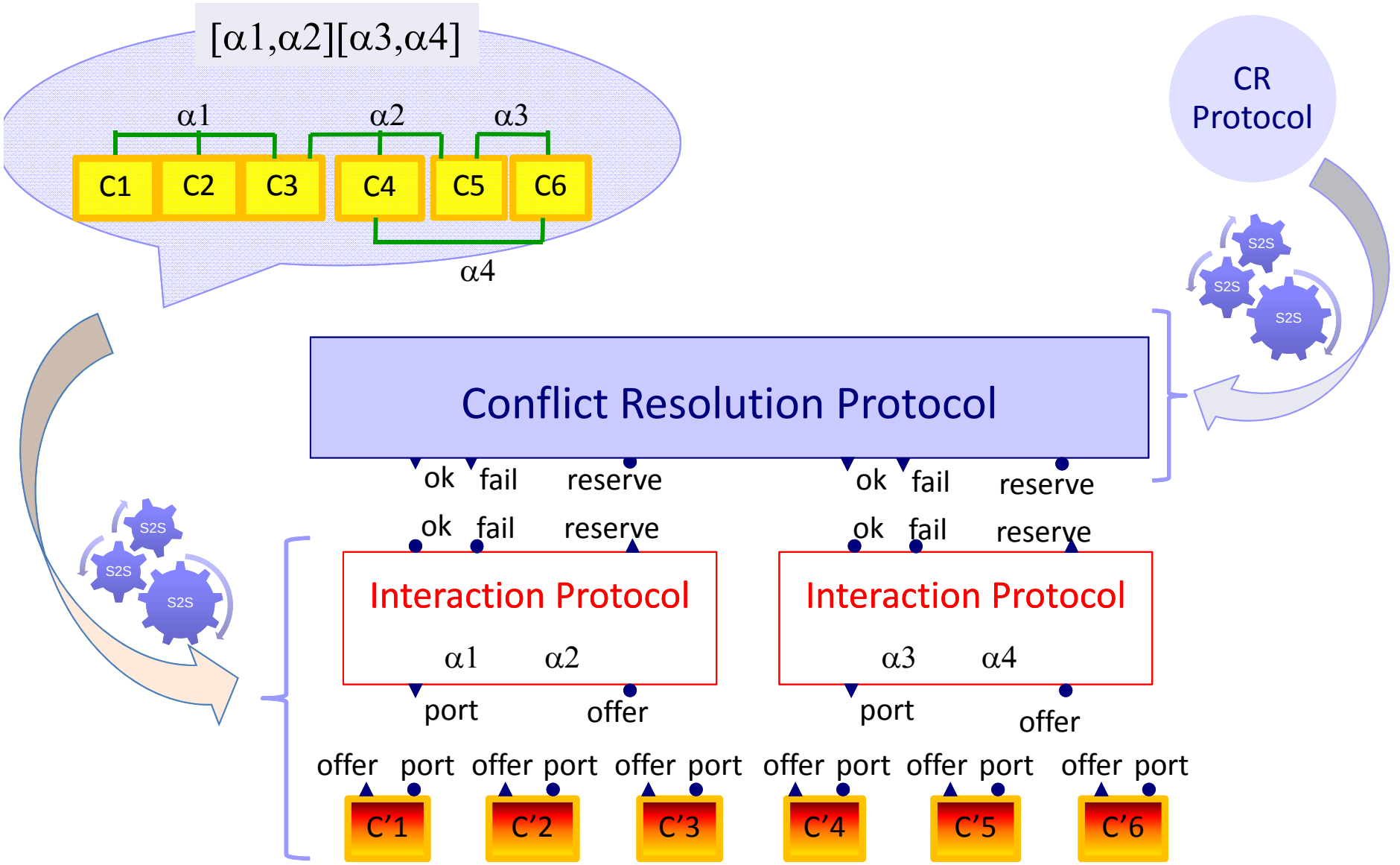
Correct by Construction – Distributed Implementation



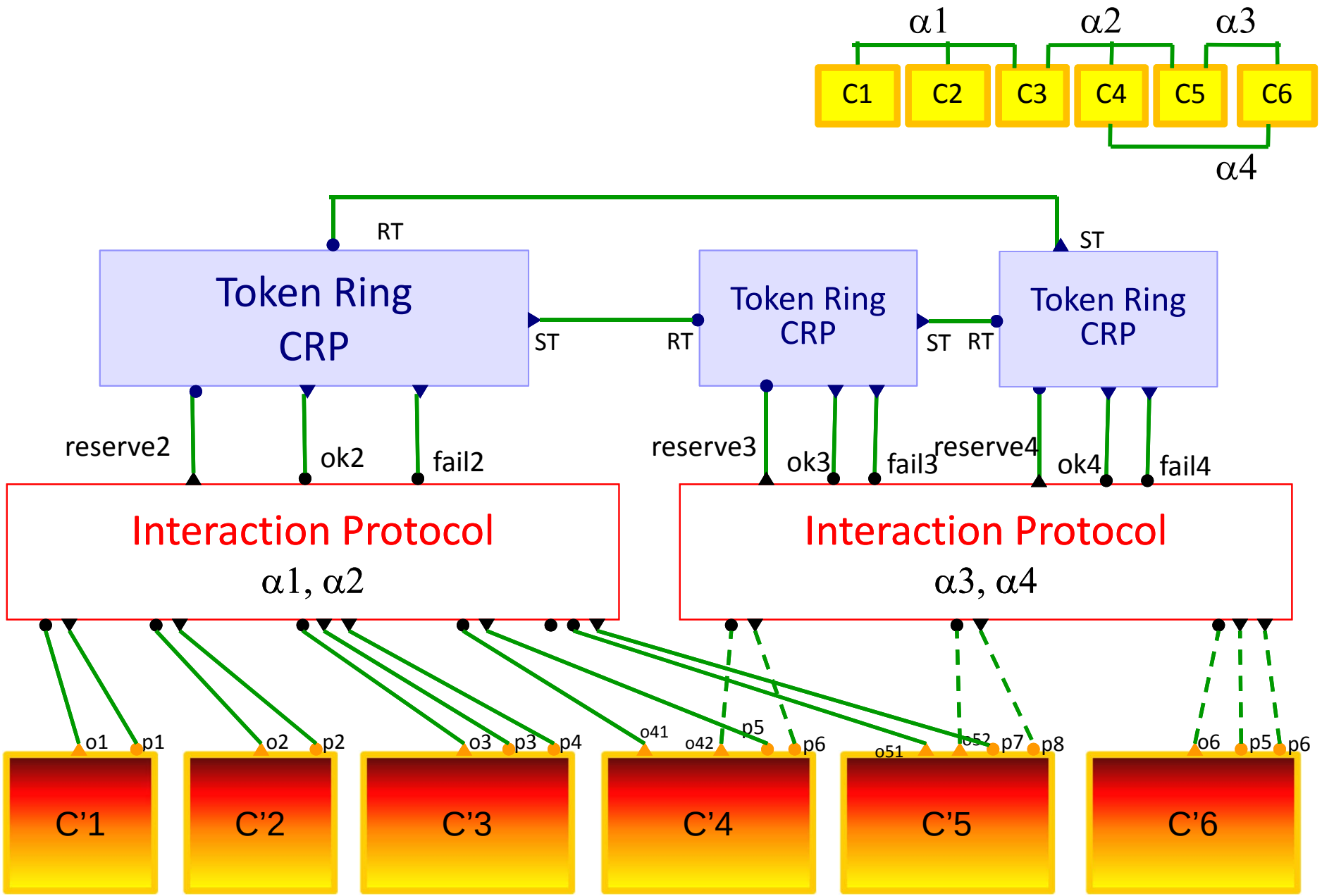
Distributed Implementation



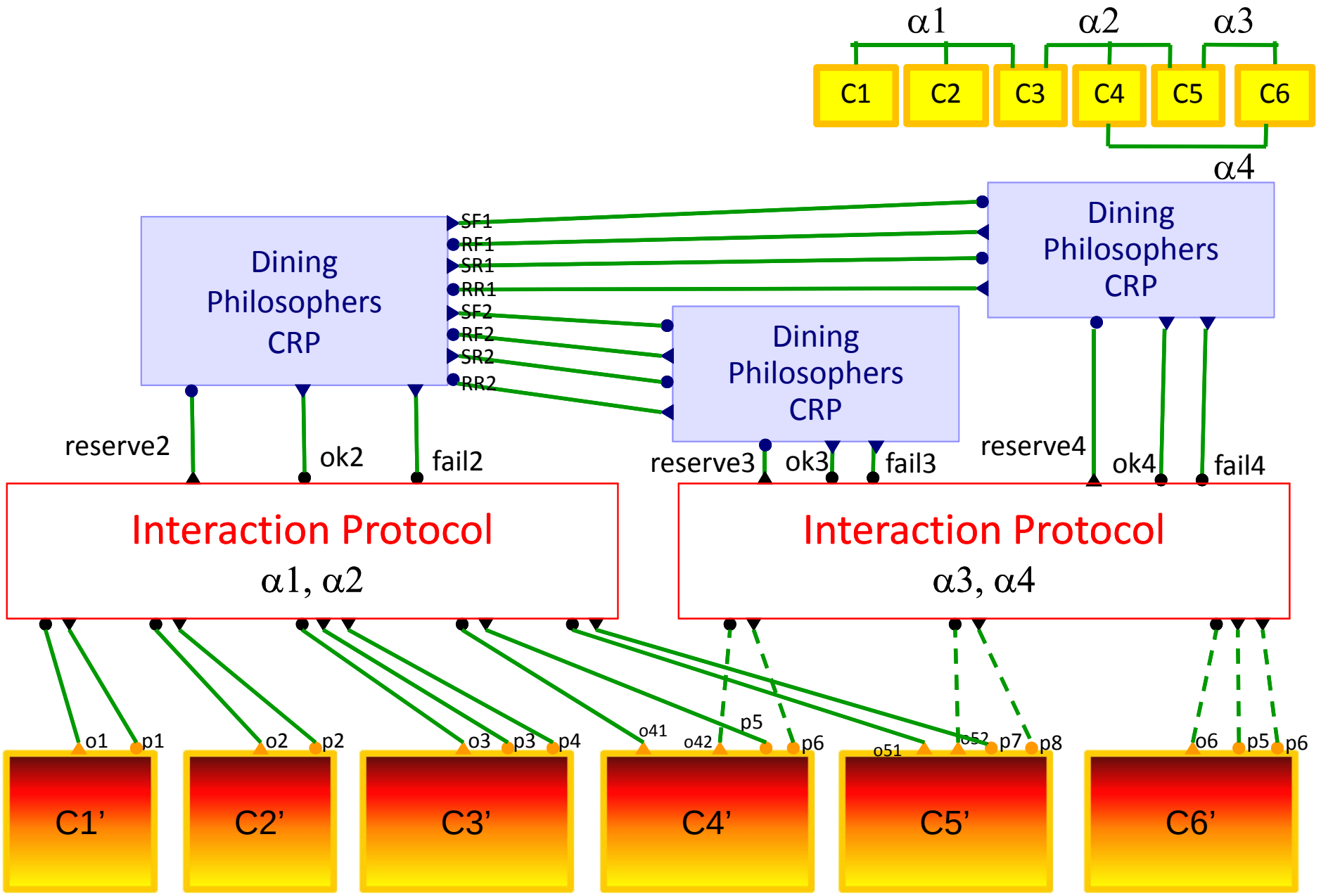
Correct by Construction – Distributed Implementation



Correct by Construction – Distributed Implementation



Correct by Construction – Distributed Implementation



Correct by Construction – Distributed Implementation