Rigorous Component-based System Design in BIP

Tutorial
Embedded Systems Week 2011

Taipei, October 9, 2011

Joseph Sifakis
VERIMAG Laboratory and EPFL
in collaboration with
A. Basu, S. Bensalem, B. Bonakdarpour, M. Bozga, J. Combaz, H. Nguyen, M. Jaber, J. Quilbeuf

OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

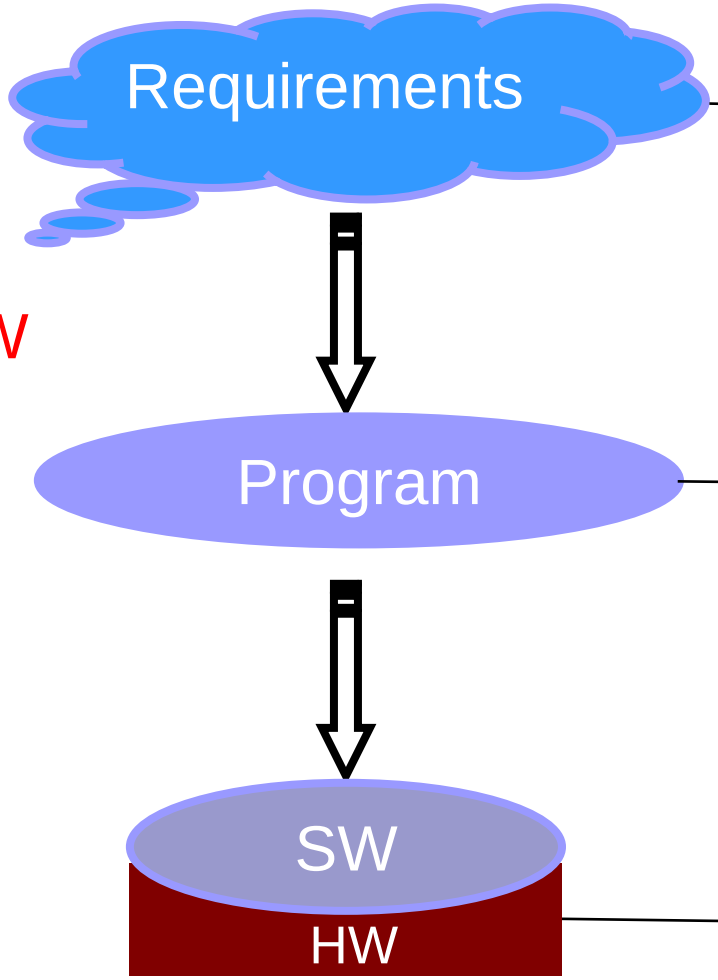
System Design – An Increasing Gap

System Design – An Increasing Gap

System Design – Simplified View

System design is the process leading to a mixed software-hardware system meeting given requirements

Different from pure SW or pure HW design!



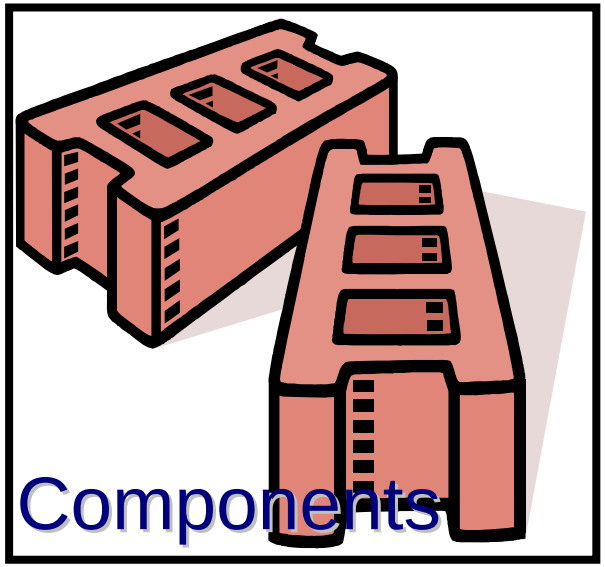
The expected behavior of the system to be designed with respect to its potential users and its environment

Executable platform-independent model meeting the requirements

System composed of HW and SW – the HW platform may be given

System Design – Quest for Productivity

Efficiency of the design process



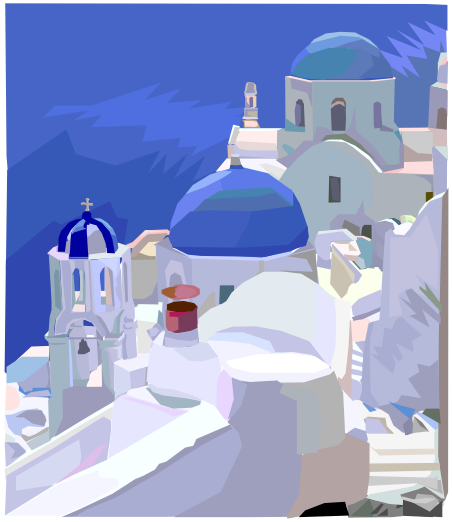
System Design – Quest for Trustworthiness

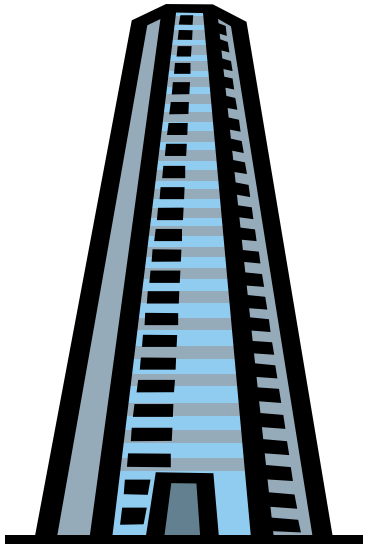
Assurance that the designed system can be trusted that it will perform as expected despite

System Design – Quest for Optimality

- Languages for describing feasible (trustworthy) design solutions
  Don't program multimedia applications in plain C!

- Optimal use of resources through design space exploration to resolve choices such as
  - reducing parallelism (through mapping on the same processor)
  - reducing non determinism (through scheduling)
  - fixing parameters (quality, frequency, voltage)

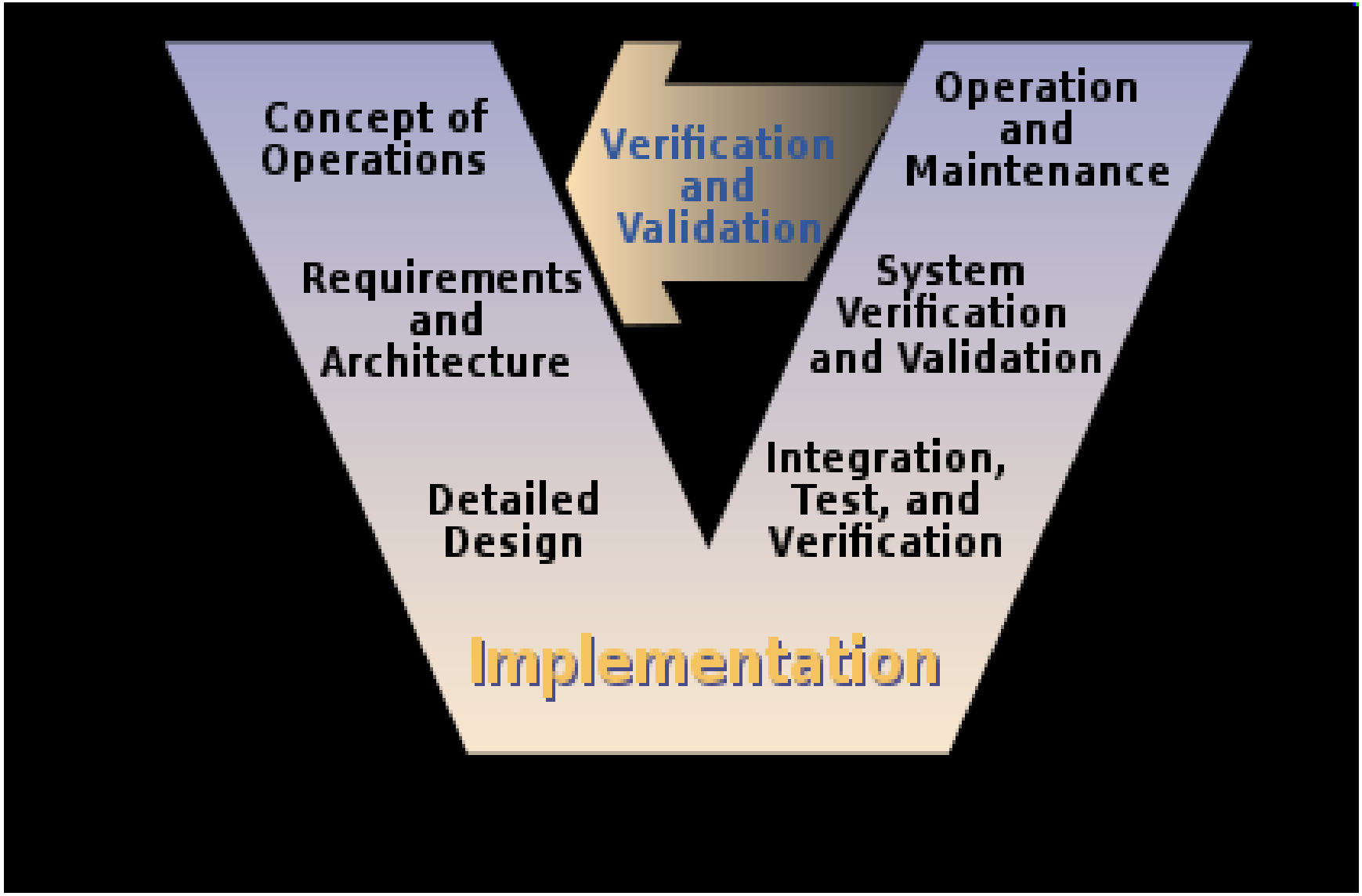




OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Correctness-by-Checking – The V-model



The V-model of the Systems Engineering Process, Source: Wikipedia

Correctness-by-Checking

- consists in comparing a system model against requirements
- is a relative judgment "Are we building the system right?"
- would be an answer to the question "Are we building the right system" if
  - Requirements could be correctly formalized, sound and complete
  - Models could represent faithfully the detailed behavior of a mixed HS/SW system interacting with its environment
  - We had effective and efficient checking techniques

- its value has been largely overestimated:
  - it only partially contributes to trustworthiness as it is limited to systems and properties that can be formalized and checked efficiently
  - it is of limited help for achieving optimality often characterized by quantitative properties

Any hype about mathematical rigor is pointless if we cannot provide supporting methods and tools !

The Limits of Correctness-by-Checking – Requirements

Requirements should be written in a formal language easy to understand and use by engineers. But

- for some types of requirements we do not have adequate languages e.g. to describe security properties
- even if some languages are expressive enough, their effective use by engineers seems to be problematic e.g. SUGAR

Requirements should be <u>sound</u>. But

- checking soundness, even for decidable requirements specification languages, is problematic due to intrinsic complexity of decision algorithms

Requirements should be <u>complete</u> that is they tightly characterize the system's behavior. But

- there is no criterion characterizing completeness for declarative languages – writing requirements may be an endless game!

The Limits of Correctness-by-Checking – Models

Models should be

- faithful e.g. whatever property is satisfied for the model holds for the real system
- generated automatically from system descriptions

For mixed HW/SW systems modeling

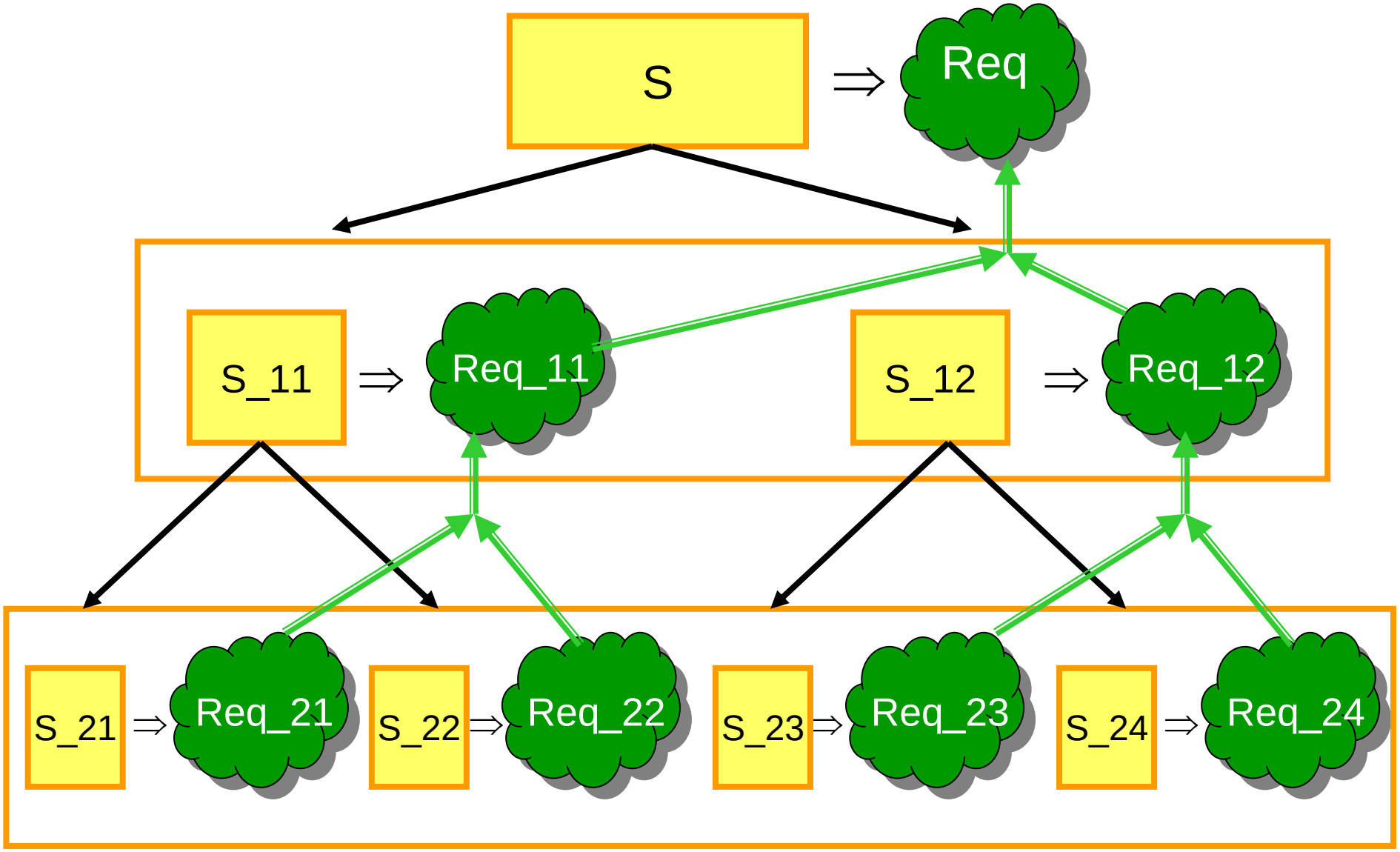
- Interaction between application software and the underlying execution platform (hardware-dependent software and hardware) is intricate and hard to understand
- Hardware execution platforms have “unpredictable” behavior
  - Variability due to manufacturing errors or aging
  - Varying execution times due to layering, caches, speculative execution
- Detailed models are intrinsically complex

The Limits of Correctness-by-Checking – Checking

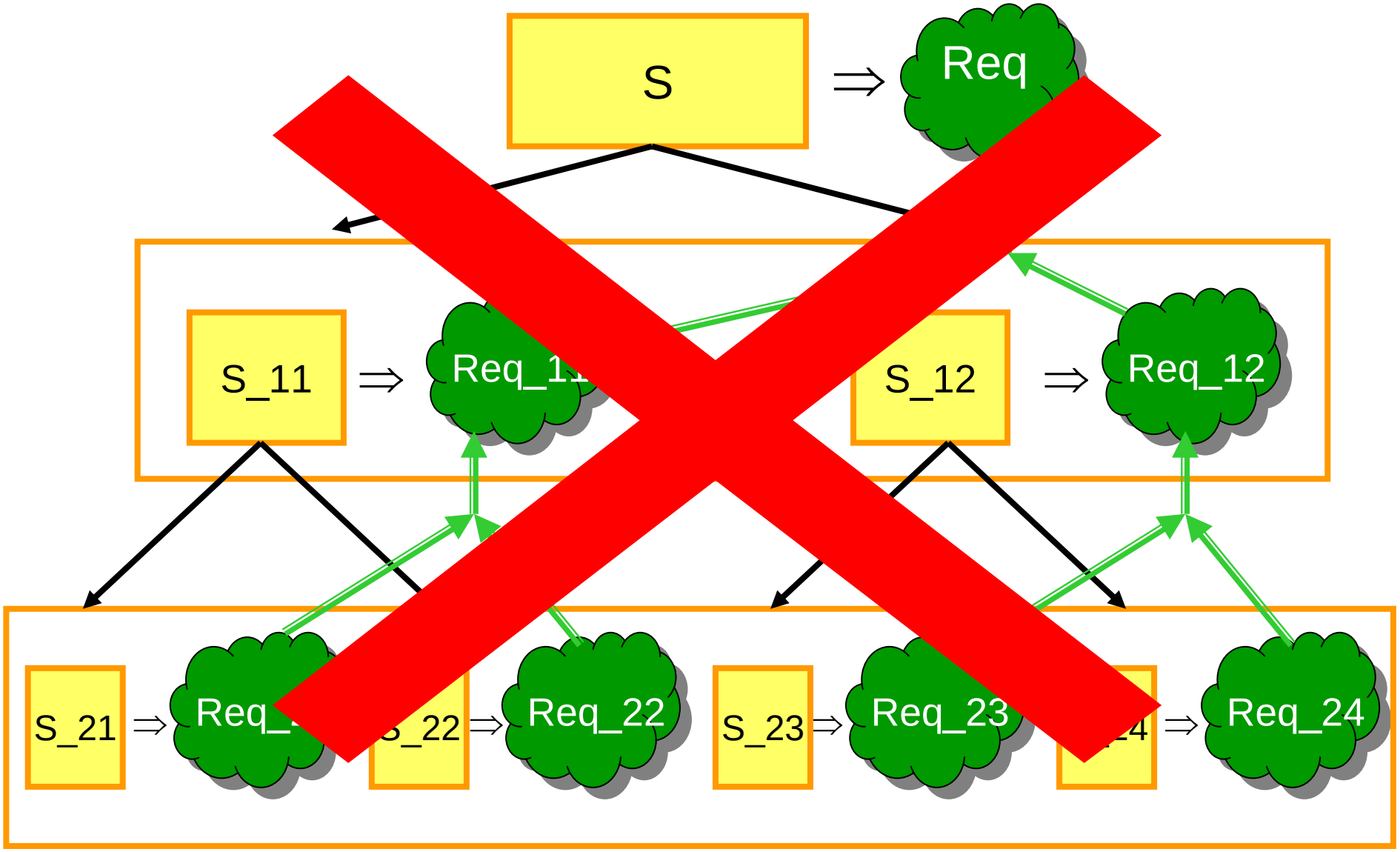
Checking correctness with respect to given requirements

- Ad hoc techniques e.g. by simulation allow error detection only
- Formal verification allows exhaustive validation, but
  - Monolithic verification is limited to small or medium size systems and to specific of properties
  - Attempts to apply <u>compositional top-down verification</u> to composite systems and thus cope with complexity of monolithic verification, have failed e.g. assume/guarantee techniques: the main obstacle is decomposition of a global property into a set of properties met by the constituent components

The Limits of Correctness-by-Checking – Top-Down Verification



The Limits of Correctness-by-Checking – Top-Down Verification



OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Rigorous Design

Trustworthy and optimal design raises complex issues crosscutting all aspects of system design and lifecycle that cannot be addressed through correctness-by-checking

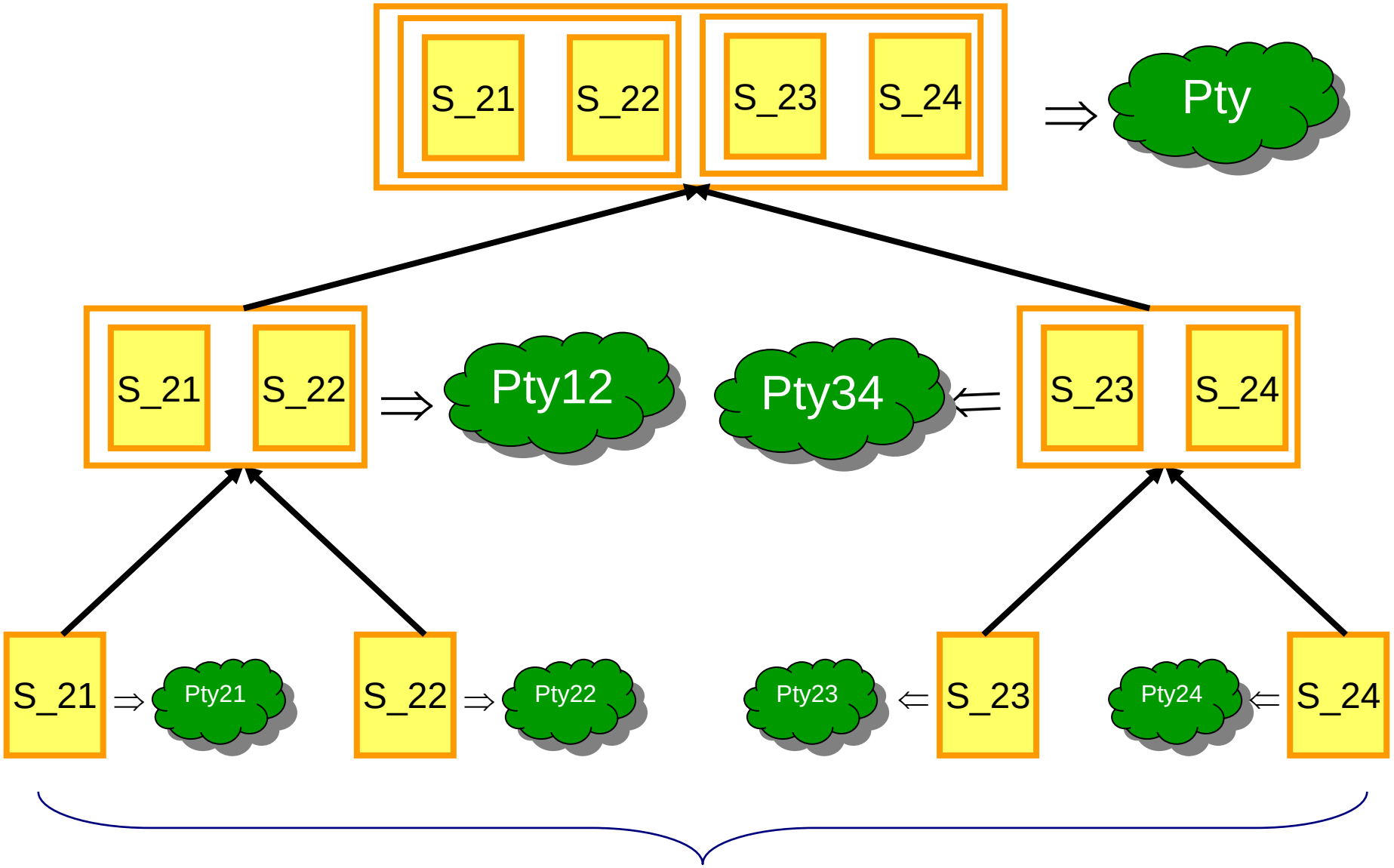
Two principles:

- Systems are built bottom-up by composing components incrementally
- Essential properties are enforced by construction

Rigorous design flow is

- Accountable: at each step we know which essential properties hold and why
- Has no gaps: all system descriptions are based on a common host language – this allows in particular, guaranteeing property preservation between steps
- Scalable: overcomes difficulties of monolithic correct-by-checking techniques

Rigorous Design – Bottom-up construction



Base elements e.g. atomic features

OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Multi-language Frameworks

System designers deal with a large variety of components, with different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system

TSpaces Concurrent Fortran Java C

Softbench NesC

SWbus BPEL,

Corba SES/Workbench

MPI SysML

Javabeans AADL

.NET Statecharts

Fractal Matlab/Simulink,

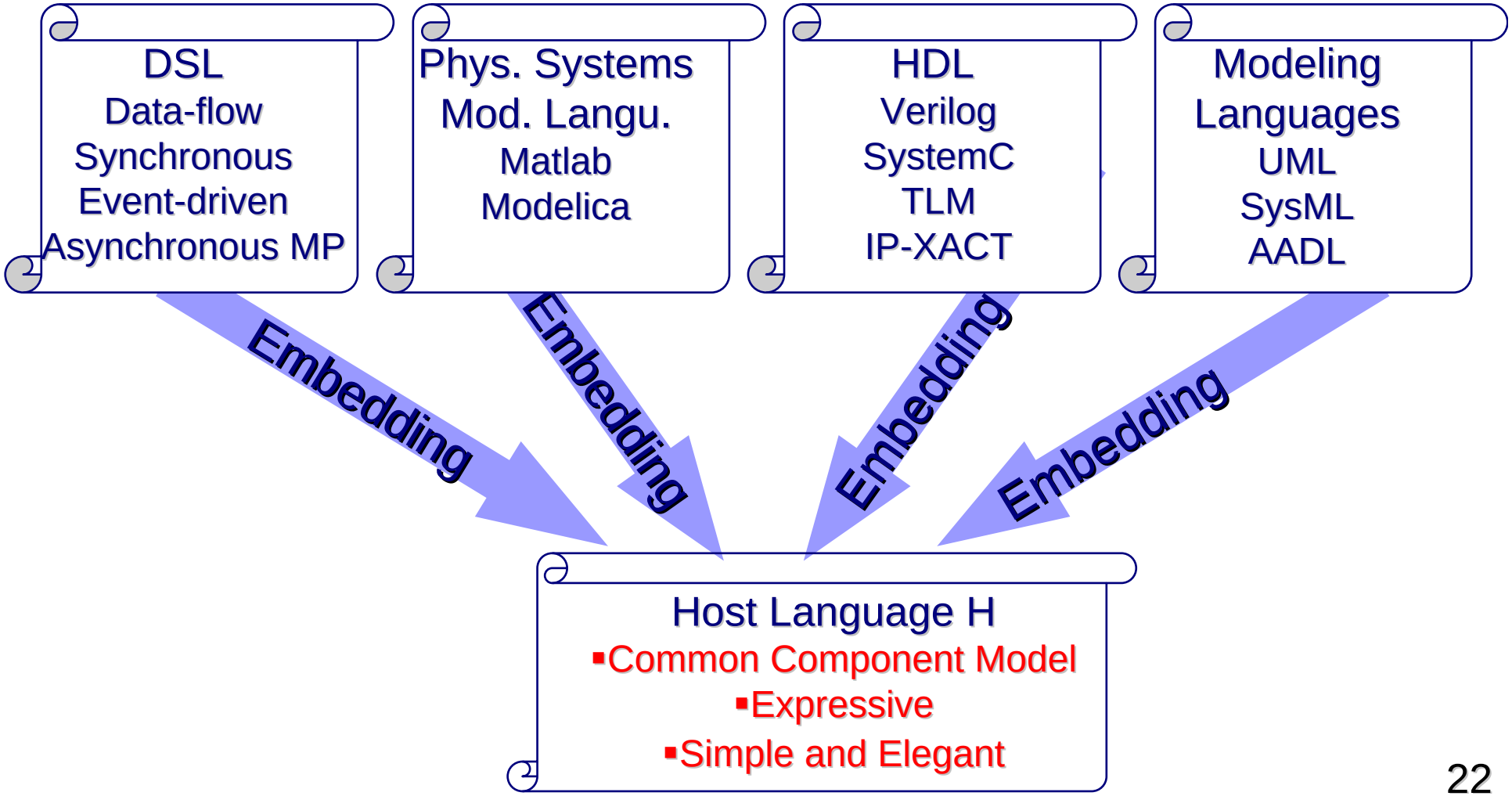
Verilog VHDL SystemC TLM

<u>Consequences</u>:

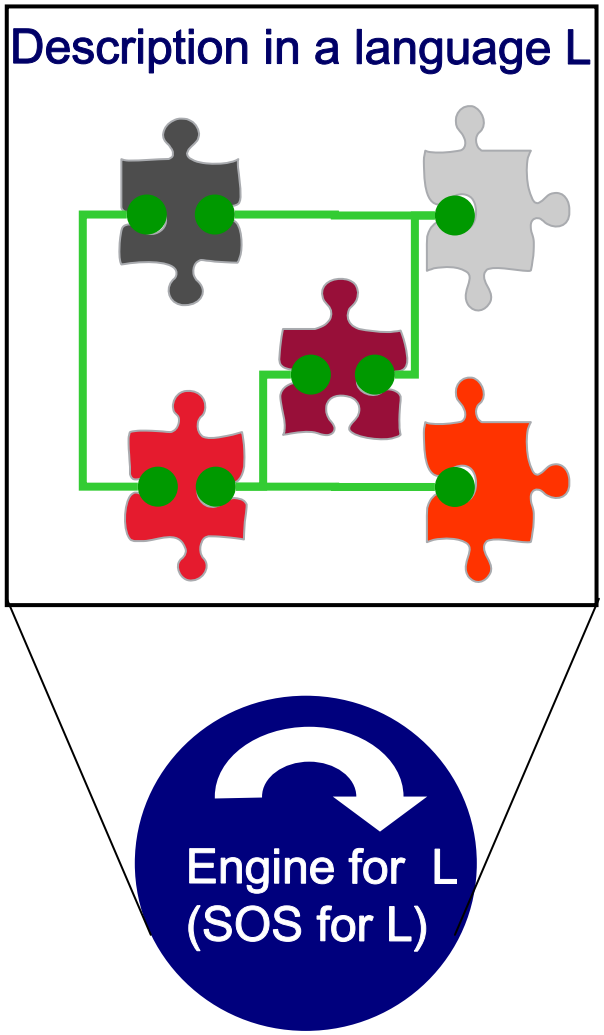
- Using semantically unrelated formalisms e.g. for programming, HW description and simulation, breaks continuity of the design flow and jeopardizes its coherency
- System development is often decoupled from validation and evaluation.

Multi-language Frameworks – Interoperability

Semantic interoperability: To ensure global consistency of the design flow we need to express the semantics of the various languages in terms of an all encompassing host language



Multi-language Frameworks – Interoperability

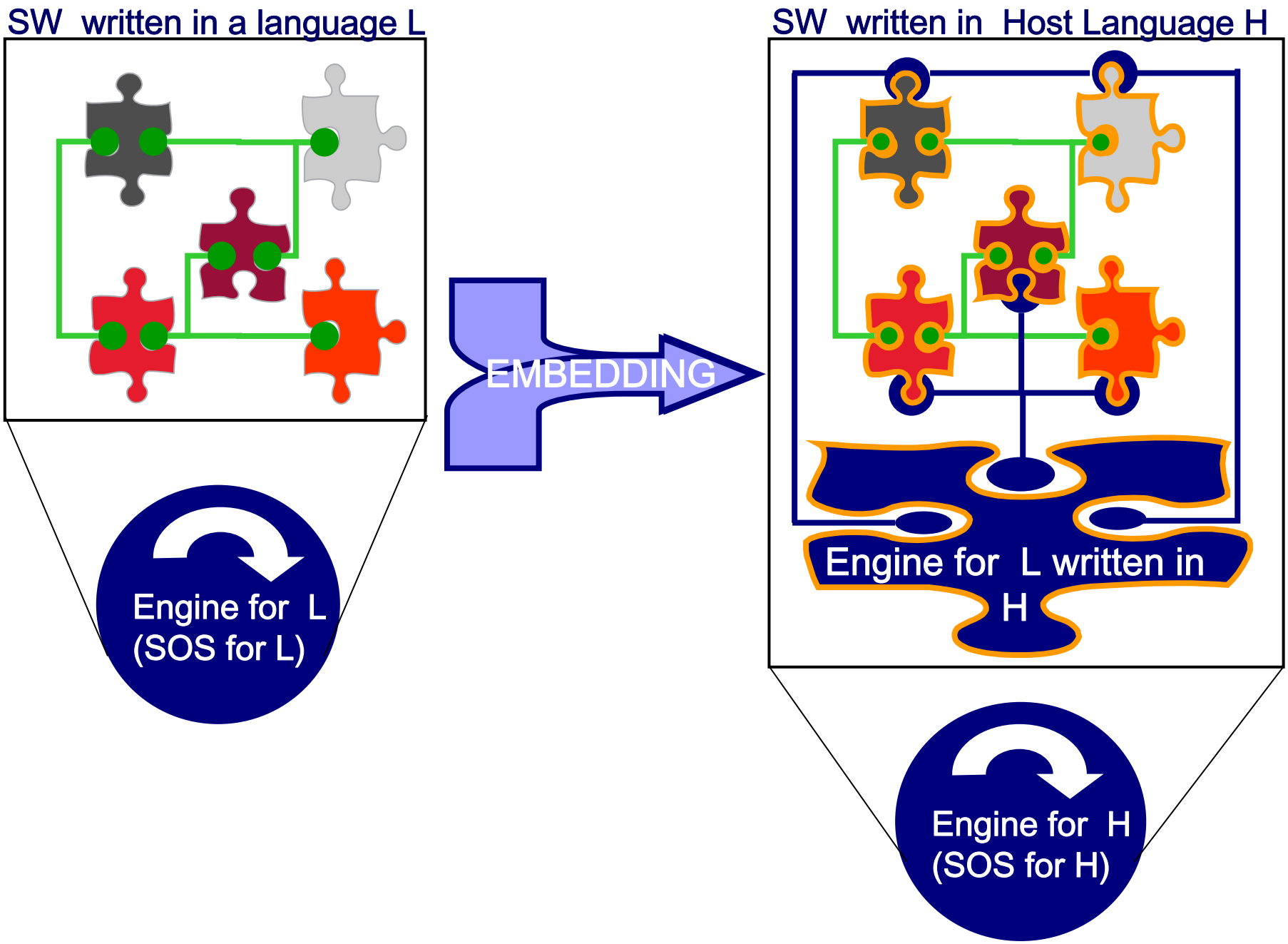


Structured Operational Semantics for L is implemented by an Engine which cyclically executes a two-phase protocol:

1. Monitors components and determines enabled connections

2.Chooses one enabled connection and executes the corresponding interaction – the latter may modify the states of the involved components

Multi-language Frameworks – Interoperability



Multi-language Frameworks – The Host Language

What we need?

- Overcome the limitations of existing theoretical frameworks based on a single composition operator e.g. function call, asynchronous message passing, rendez-vous

- A Common Component Model based on a unified composition paradigm for describing and analyzing the coordination between components. in terms of tangible, well-founded and organized concepts
  - expressive – coordination glue between components e.g. protocols, schedulers, buses, architectures can be expressed as the combination of composition operators

  - simple and elegant - achievement of a given functionality with a minimum of mechanism and a maximum of clarity

OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Architectures

Architectures

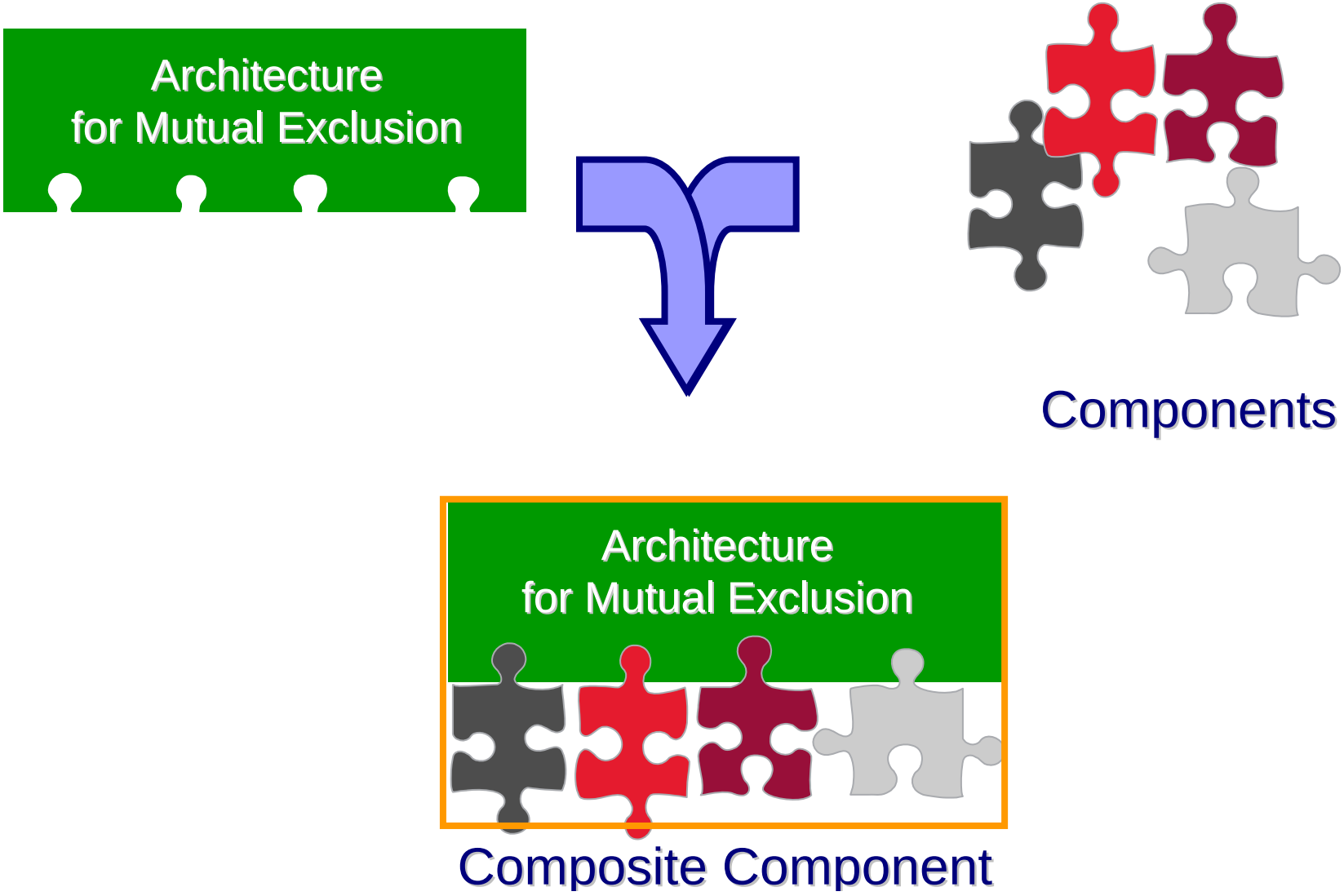
- depict design principles, paradigms that can be understood by all, allow thinking on a higher plane and avoiding low-level mistakes
- are a means for enforcing global properties characterizing the coordination between components – correctness for free
- Using architectures is key to ensuring trustworthiness and optimality in in networks, OS, middleware, HW devices etc.

System developers extensively use libraries of <u>reference architectures</u> ensuring both functional and non functional properties e.g.

- Fault-tolerant architectures
- Resource management and QoS control
- Time-triggered architectures
- Security architectures
- Adaptive Architectures

Architectures – Definition

An architecture is as operator that constraints the behavior of the composed components to enforce a given property – correctness for free!



Architectures – Property Enforcement

Postulate: An architecture ensuring a given property can be obtained as the combination of a set of architectures ensuring basic properties:

1. Identification of an expressive and minimal set of basic architectures/properties

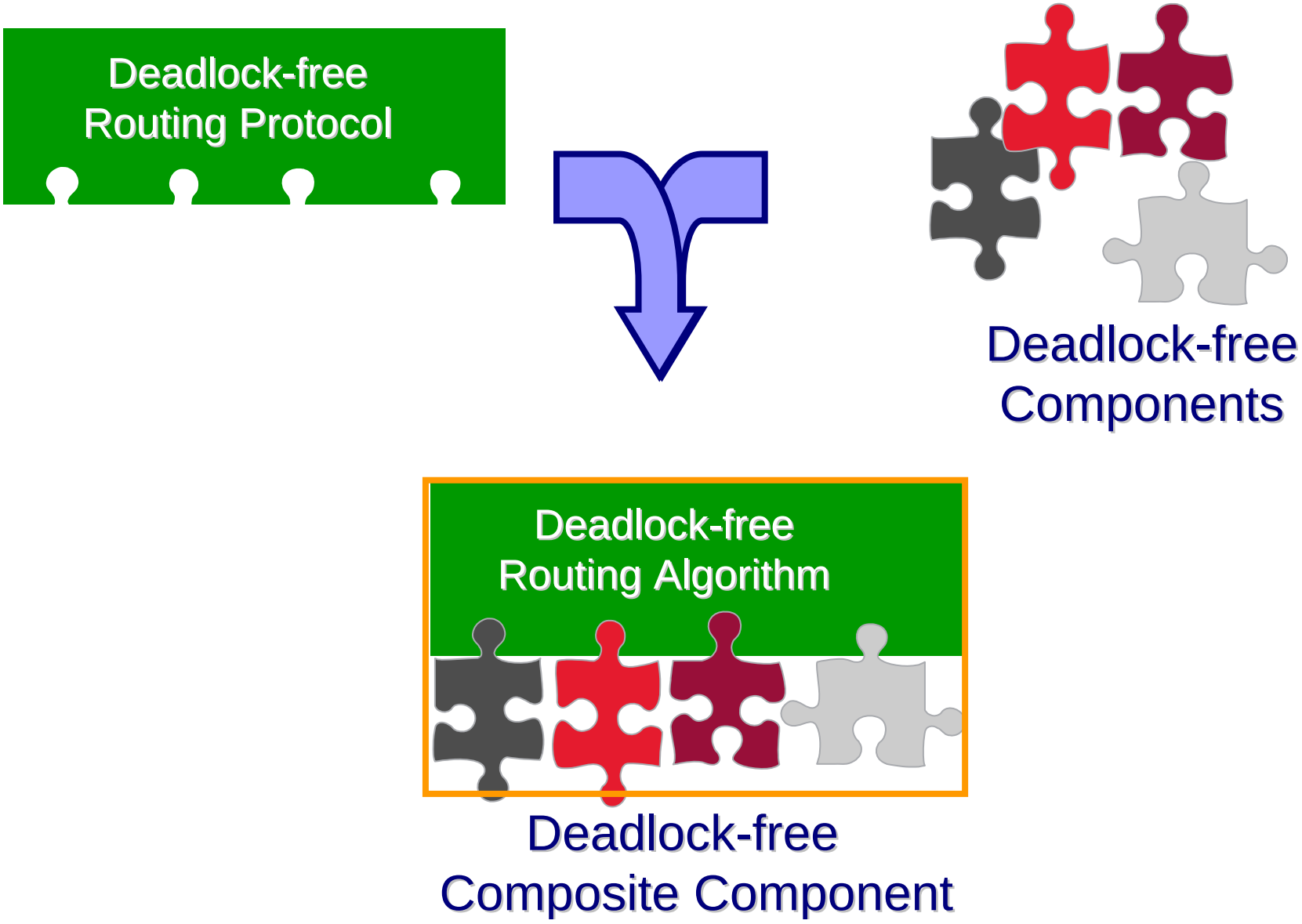
Security architectures are obtained by composition of architectures ensuring

- Antivirus protection
- Intrusion Detection System, Intrusion Protection System
- Sampling
- Monitoring
- Watermarking
- Embedded cryptography
- Integrity checking

2. Correct-by-construction theory for combining architectures and their characteristic properties

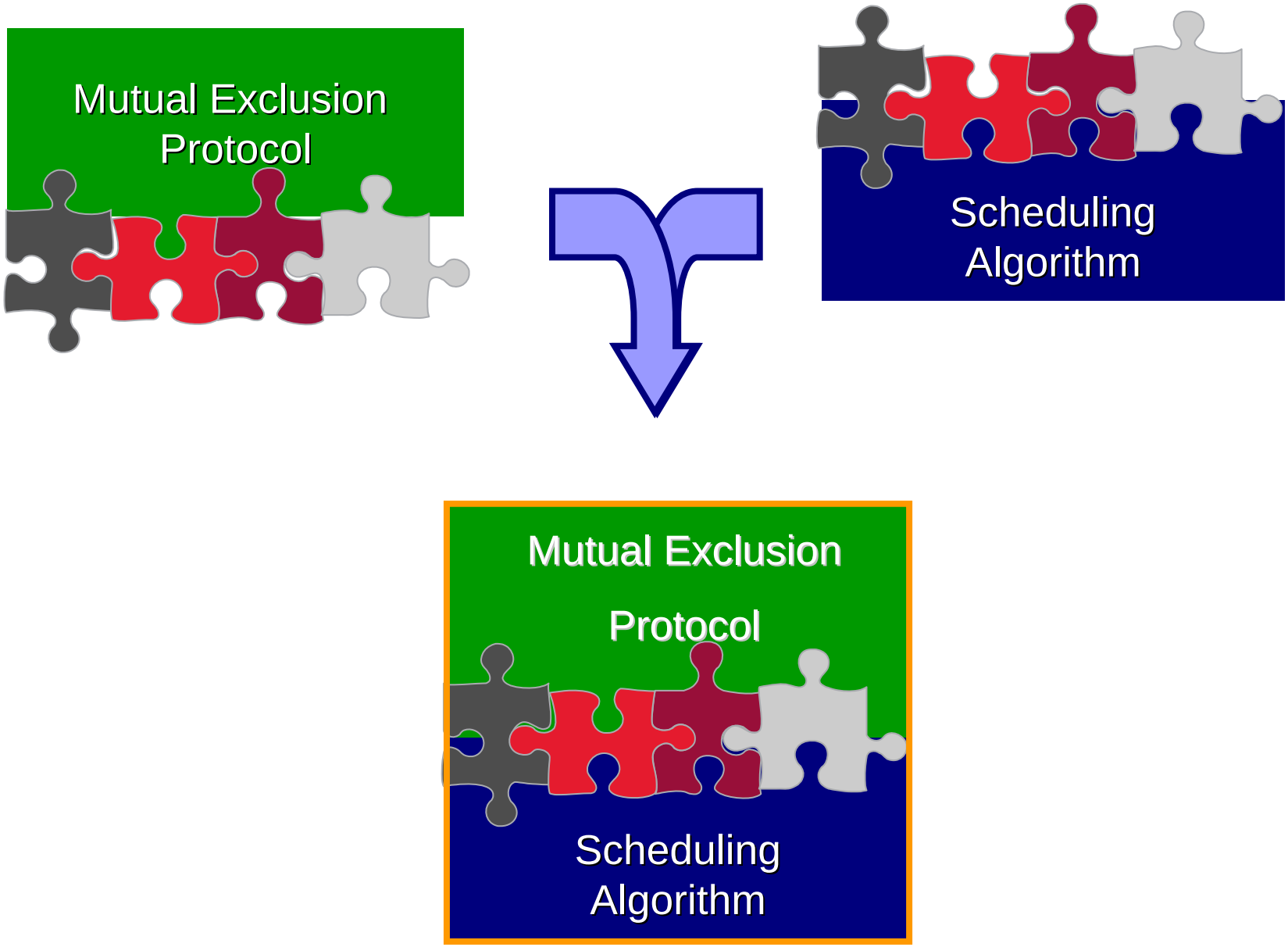
Architectures – Correctness-by-Construction

Property Preservation



Architectures – Correctness-by-Construction

Property Composability



OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Automatic Implementation

Problem: Given an application software and an execution platform generate trustworthy and optimal implementations

Application Software is written in high level languages supporting abstractions such as

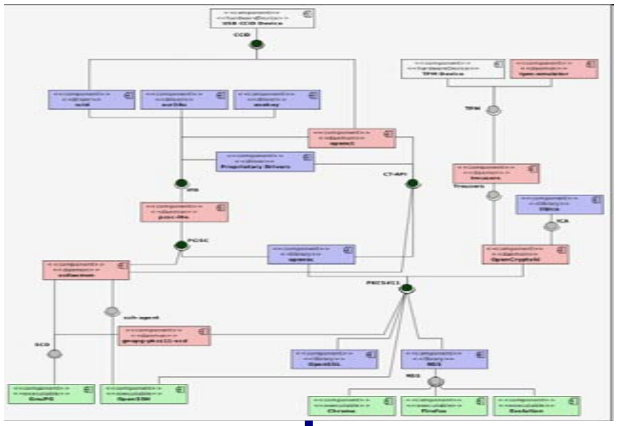
- Atomicity of primitives and interactions between components – in particular multiparty interaction
- A logical notion of time assuming zero-time actions and synchrony of execution with respect to the physical environment

The generated implementation should be

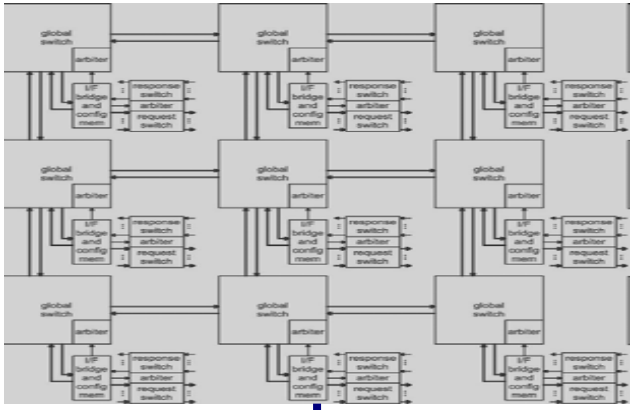
- functionally equivalent to the application software (correctness)
- generated automatically for a given mapping associating
  - Processes of the ASW → processors of the platform
  - Data of the ASW → memories of the platform
  - Interactions → execution paths or protocols

Automatic Implementation – Methods and Tools

Application SW



Execution Platform Model



Mapping

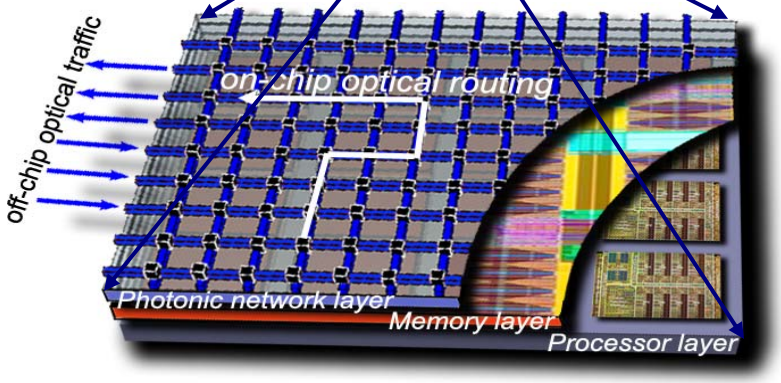
Process → Processor

Data → Memory

Code Generation and Deployment

Optimal Design

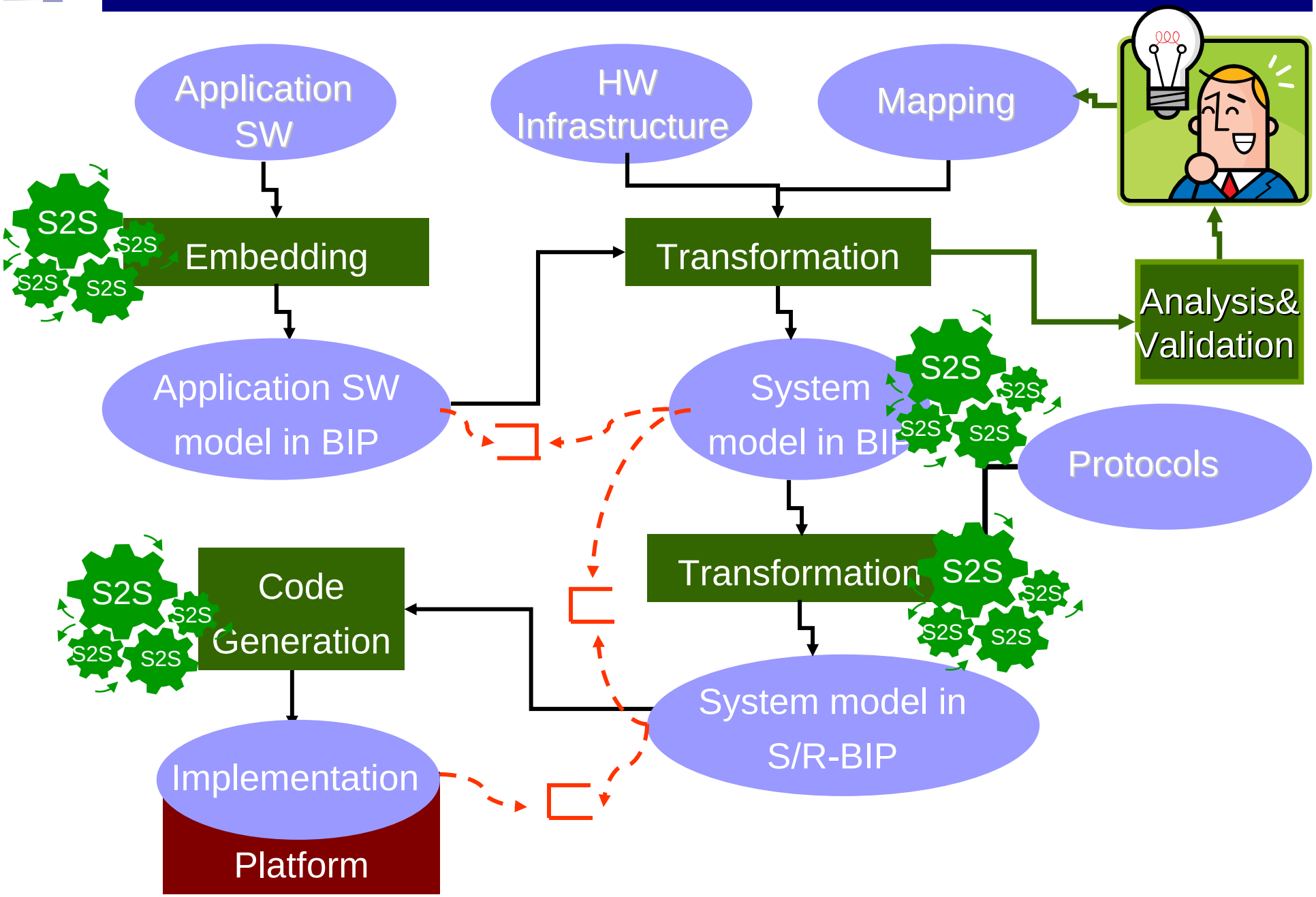
Analysis and Validation Tools



OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Putting RSD into Practice – System Design Flow in BIP

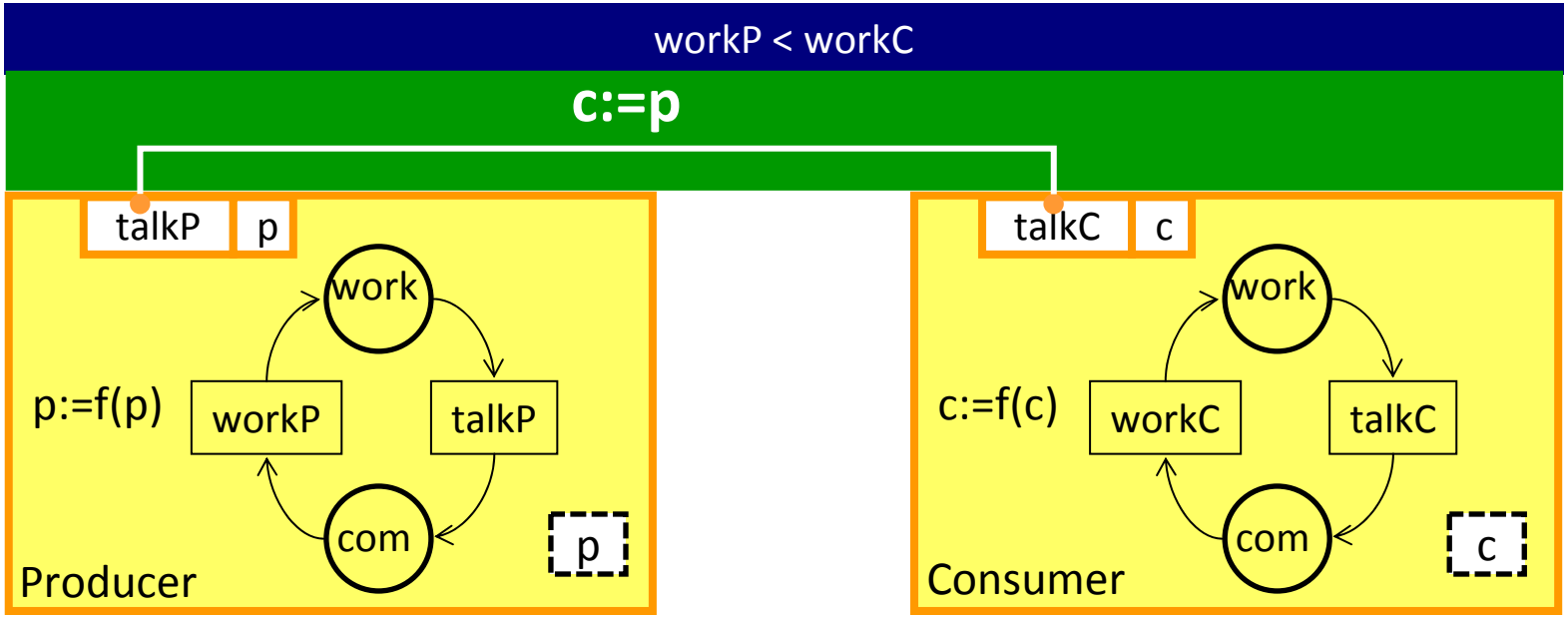


Putting RSD into Practice – Using BIP as a Host Language

BIP is a component framework including

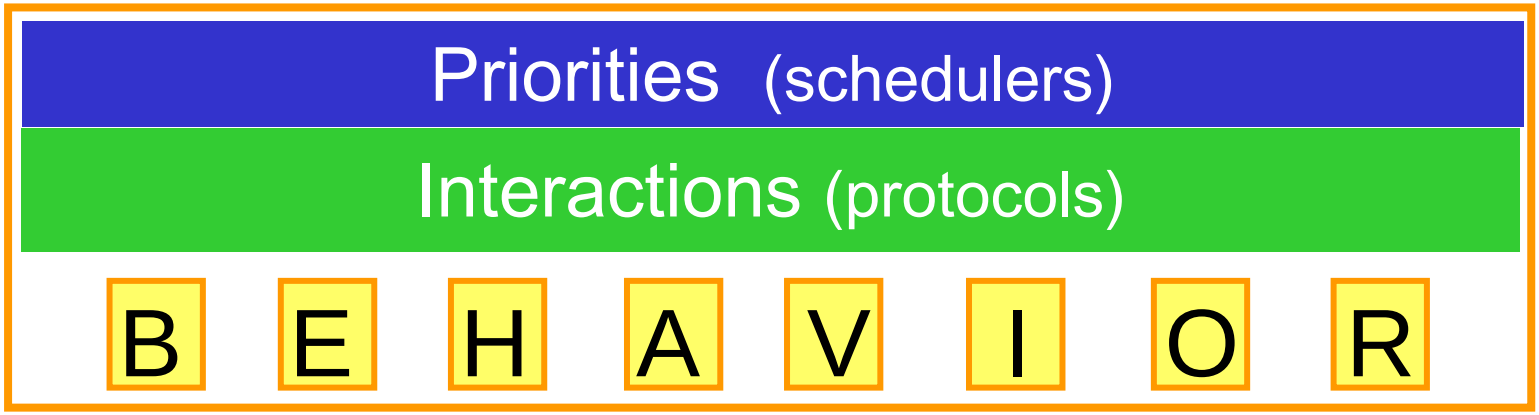
1. a language for hierarchical construction of composite components

2. associated tools
   - Translators from domain specific languages
   - Run-time Engines
   - Validation and analysis tools
   - Code generation tools

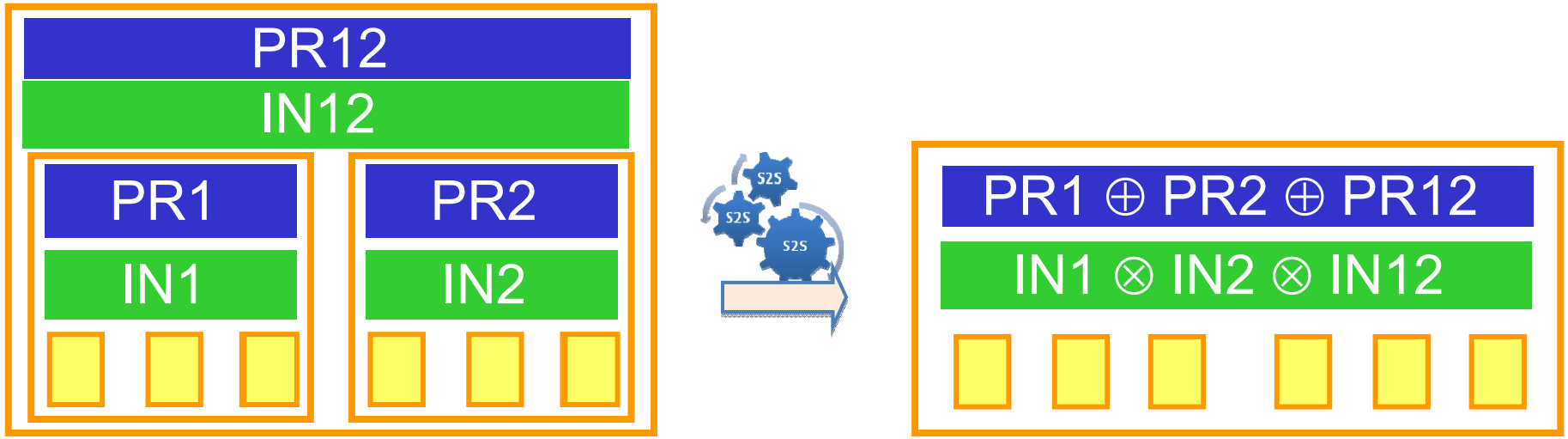


Putting RSD into Practice – Basic Concepts of BIP

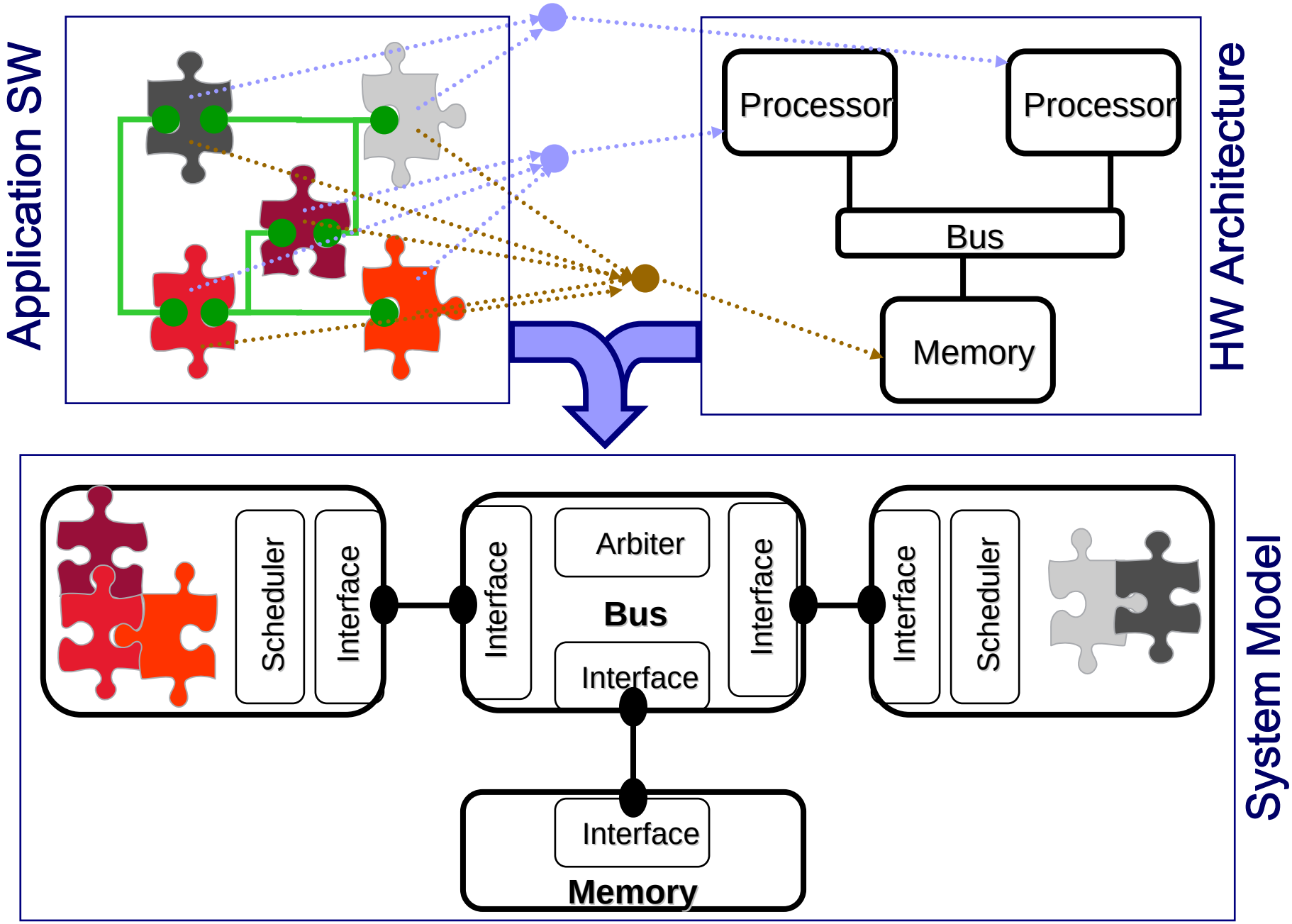
Layered component model



Composition operation parameterized by glue IN12, PR12



Putting RSD into Practice – Integrating HW constraints

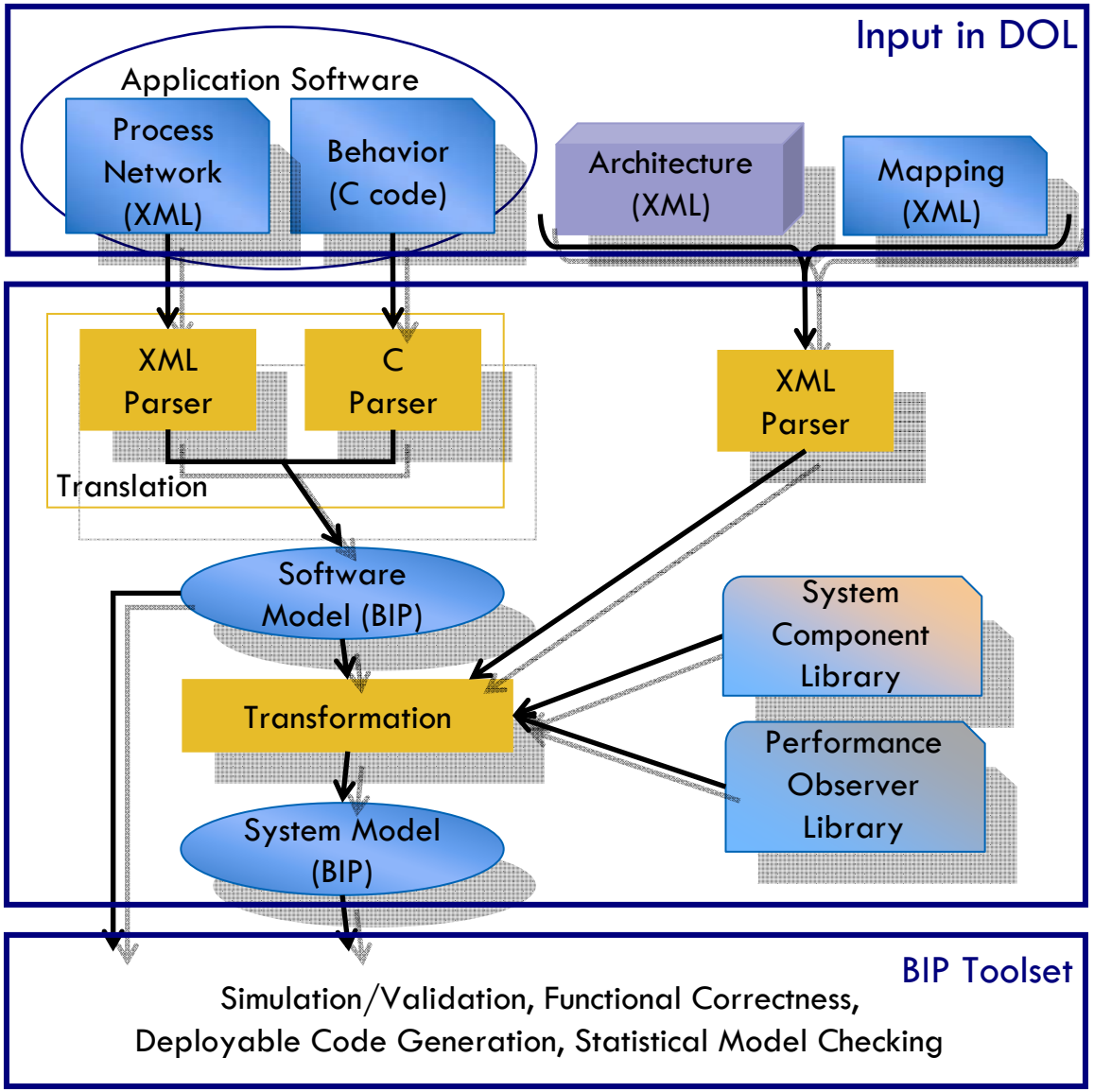


Putting RSD into Practice – Integrating HW constraints

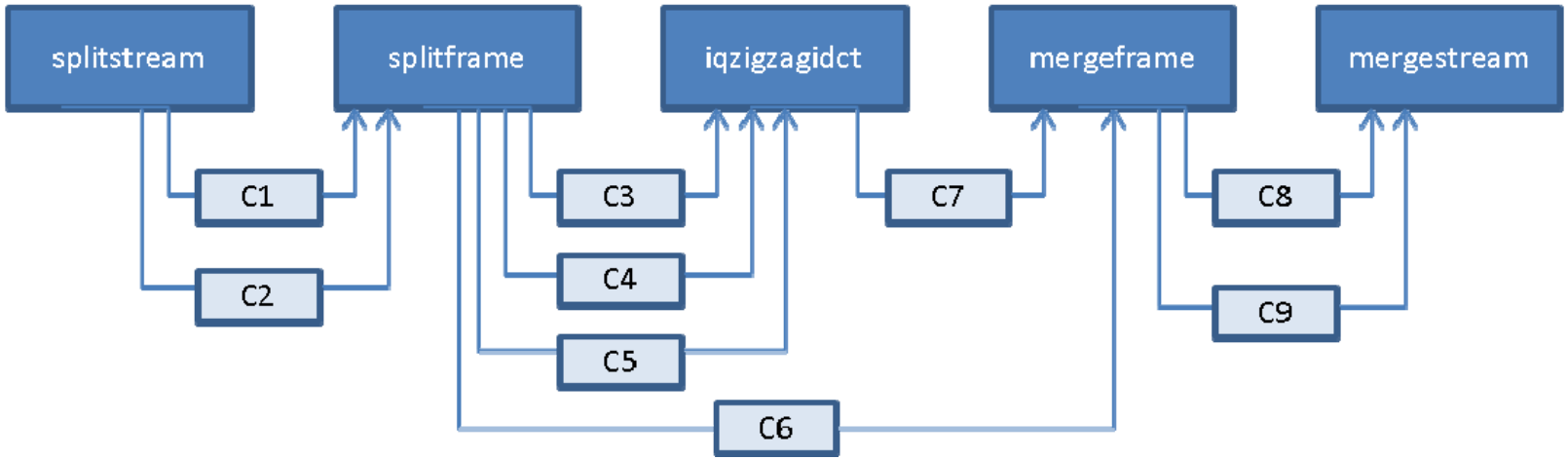
The ***frontend*** transforms networks of processes written in DOL into BIP models

The ***backend*** applies a series of source-to-source correct-by-construction transformations over the BIP software model.

The generated system model includes can be validated and analyzed



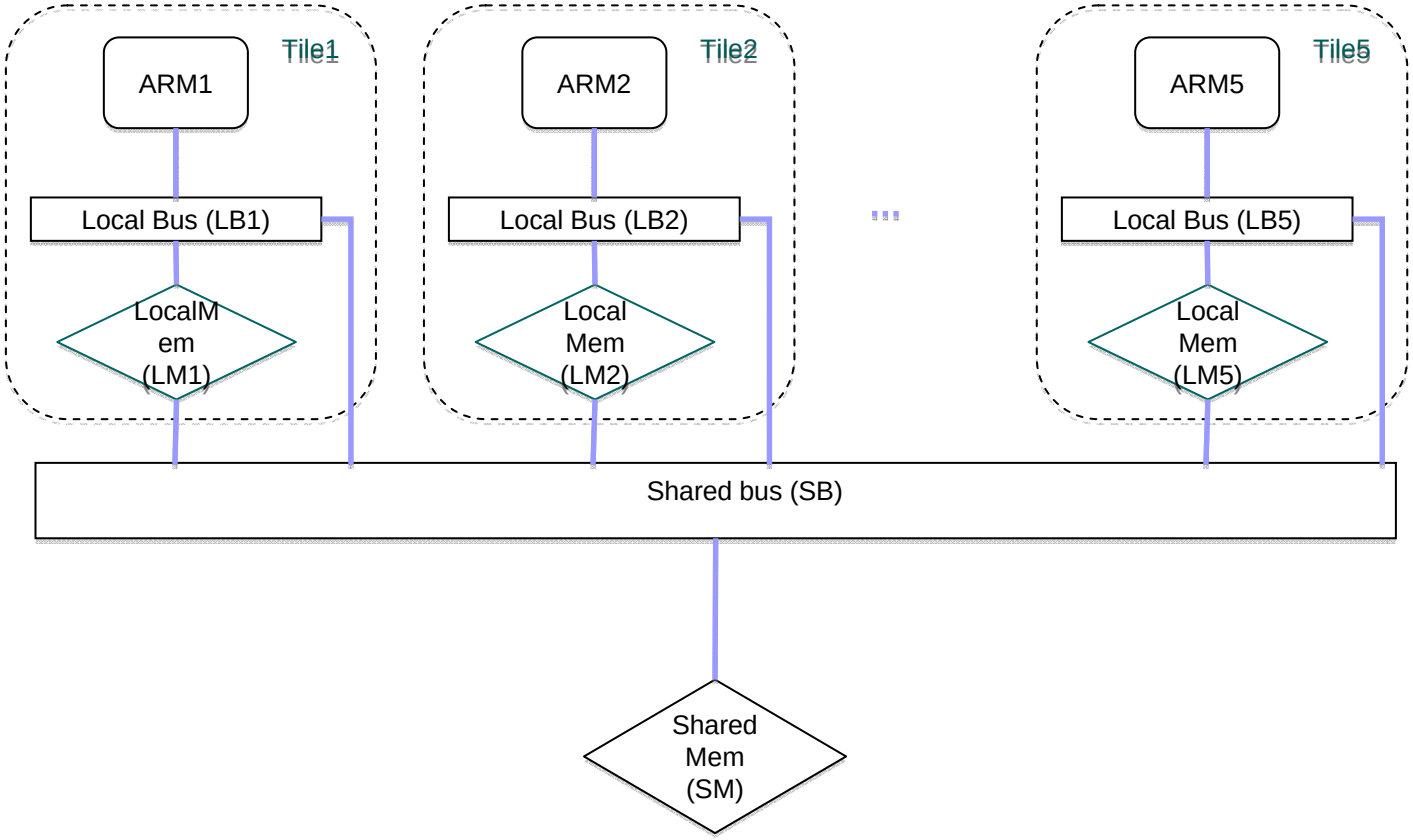
MJPEG decoder



The MJPEG decoder

- reads a sequence of frames and displays the decompressed frames
- is described as a process network with five processes and nine communication channels

MJPEG decoder – MPARM Architecture



A simplified Multi-Processor ARM (MPARM)

- Five identical tiles and a Shared Memory connected via a Shared Bus.
- Tiles contain a CPU connected to its Local Memory via a Local Bus.
- CPU frequency: 200 Mhz.
- Access times: 2 CPU cycles for local memory
  6 CPU cycles for shared memory

MJPEG decoder – Mapping

Process mapping table

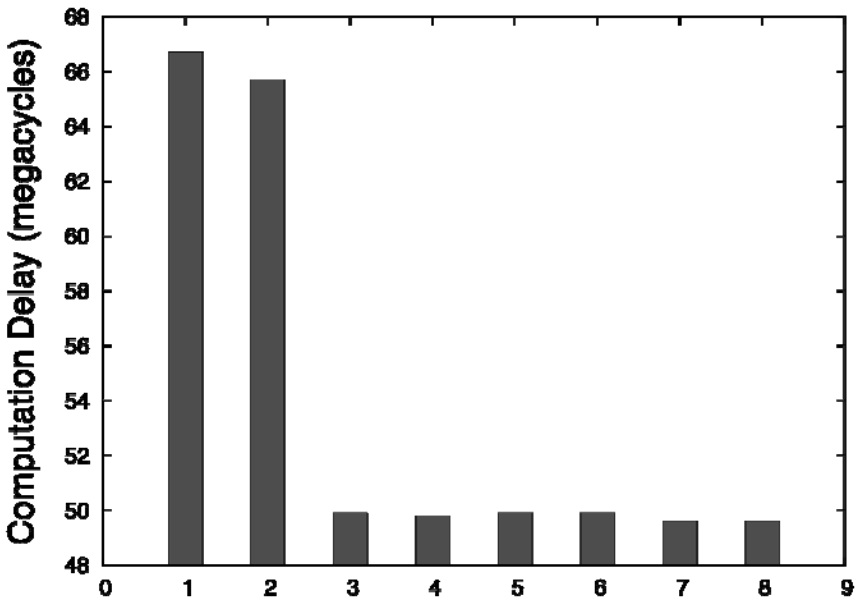
	ARM1	ARM2	ARM3	ARM 4	ARM5
Mapping1	all				
Mapping2	SS, SF, IQ	MF, MS			
Mapping3	SS, SF	IQ,MF,MS			
Mapping4	SS, SF	IQ	MF, MS		
Mapping5	SS, MS	SF	IQ	MF	
Mapping6	SS	SF	IQ	MF	MS
Mapping7	SS, SF	IQ	MF, MS		
Mapping8	SS	SF	IQ	MF	MS

SW-Channel mapping table

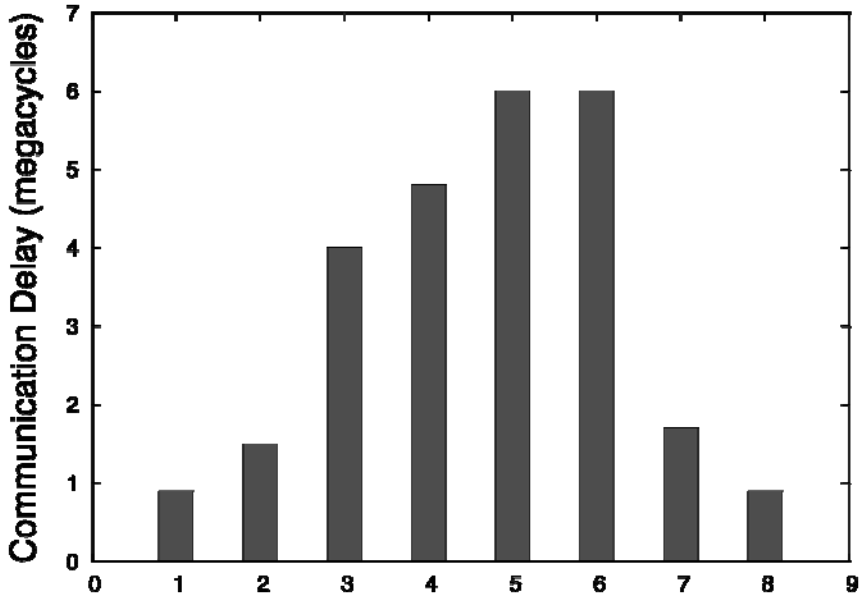
	Shared	LM1	LM2	LM3	LM4
Mapping1		all			
Mapping2	C6, C7	C1, C2, C3, C4, C5	C8, C9		
Mapping3	C3, C4, C5, C6	C1, C2	C7, C8, C9		
Mapping4	C3, C4, C5, C6, C7	C1, C2		C8, C9	
Mapping5	all				
Mapping6	all				
Mapping7	C6, C7	C1, C2, C3, C4, C5		C8, C9	
Mapping8		C1, C2	C3, C4, C5, C6	C7	C8, C9

MJPEG decoder – Results

Computation delay



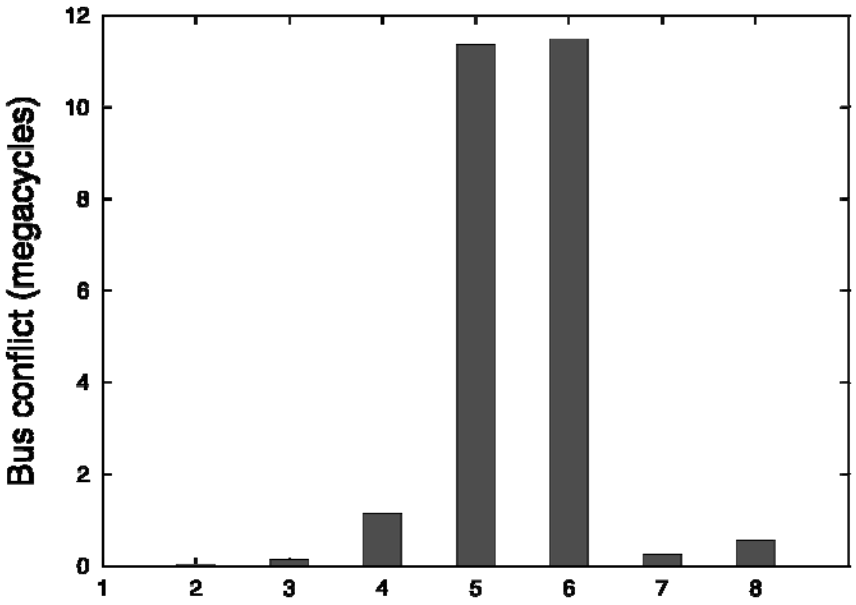
Communication delay

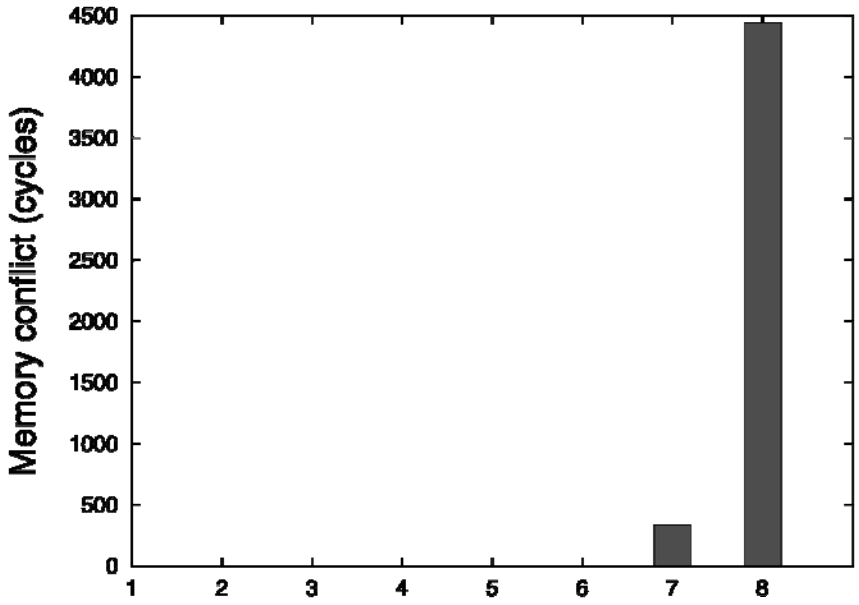


- Mapping (1) gives the worst computation time as all processes are mapped to a single processor.
- The communication overhead is reduced if we distribute sw-channels to the local memories of the processors.

MJPEG decoder – Results

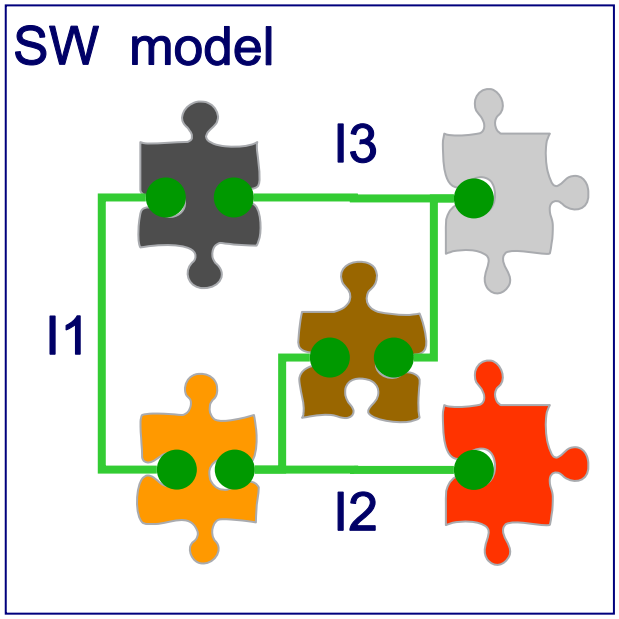
Delay (waiting time) due to bus and memory conflict

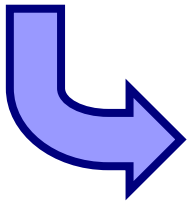




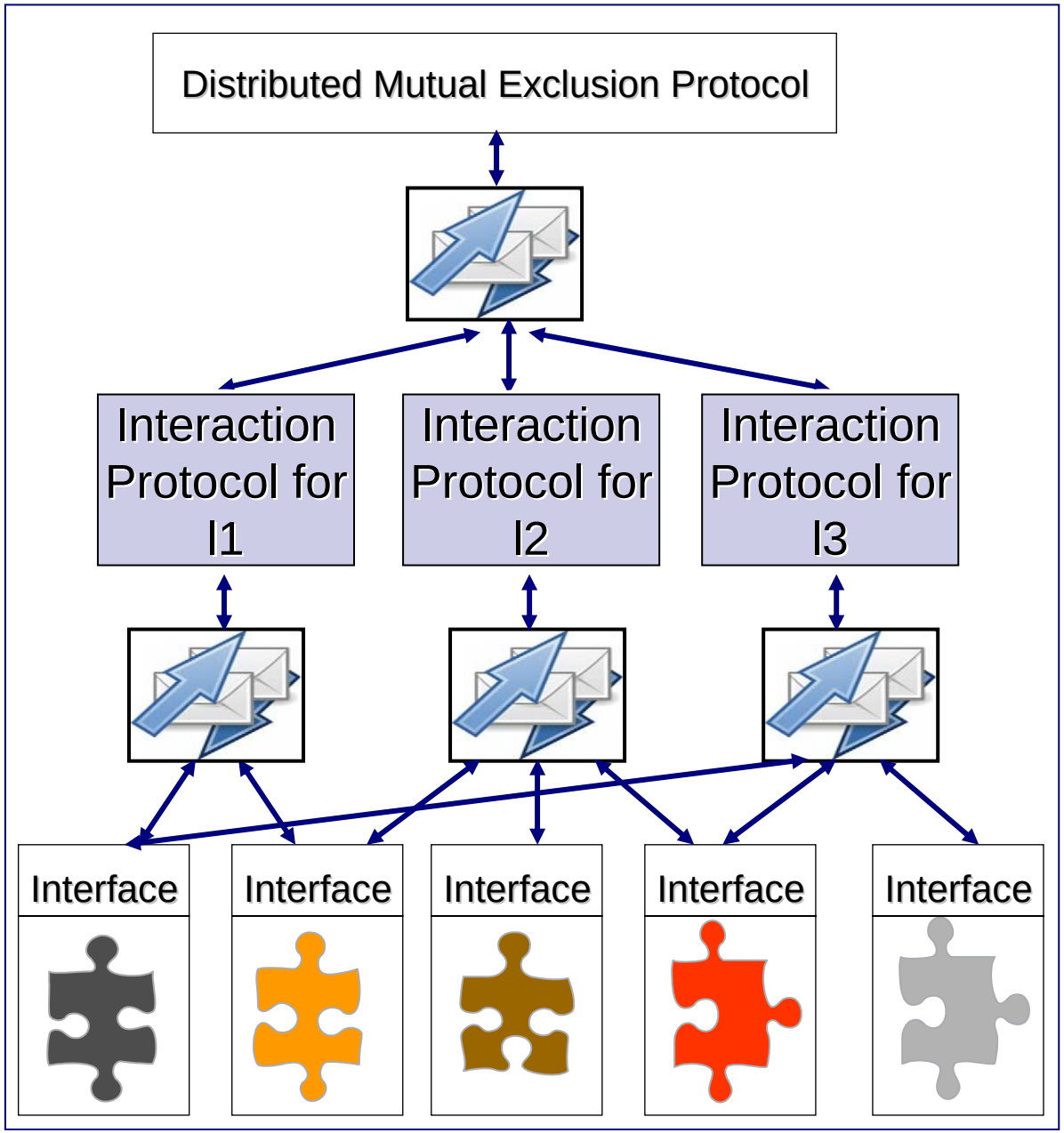
As more channels are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as is evident for mapping (8).

Putting RSD into Practice – Distributed Implementation

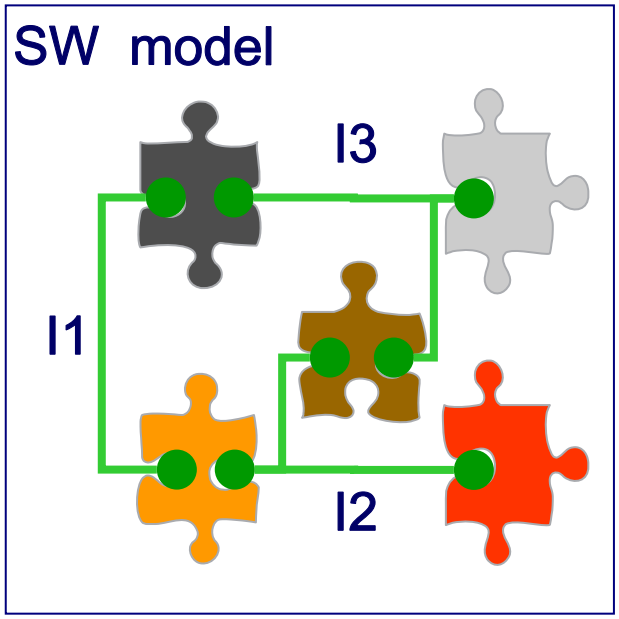




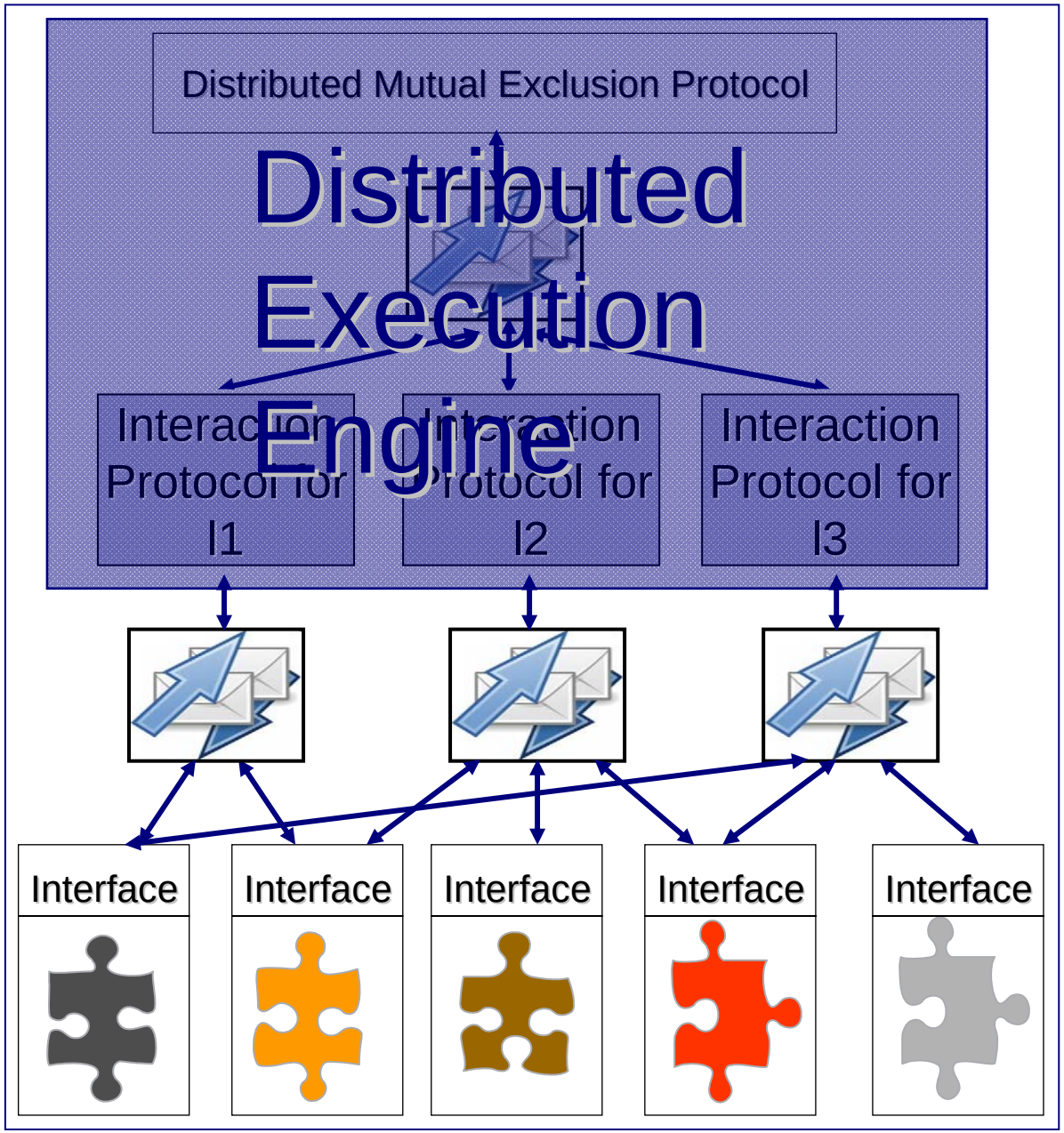
Distributed Implementation



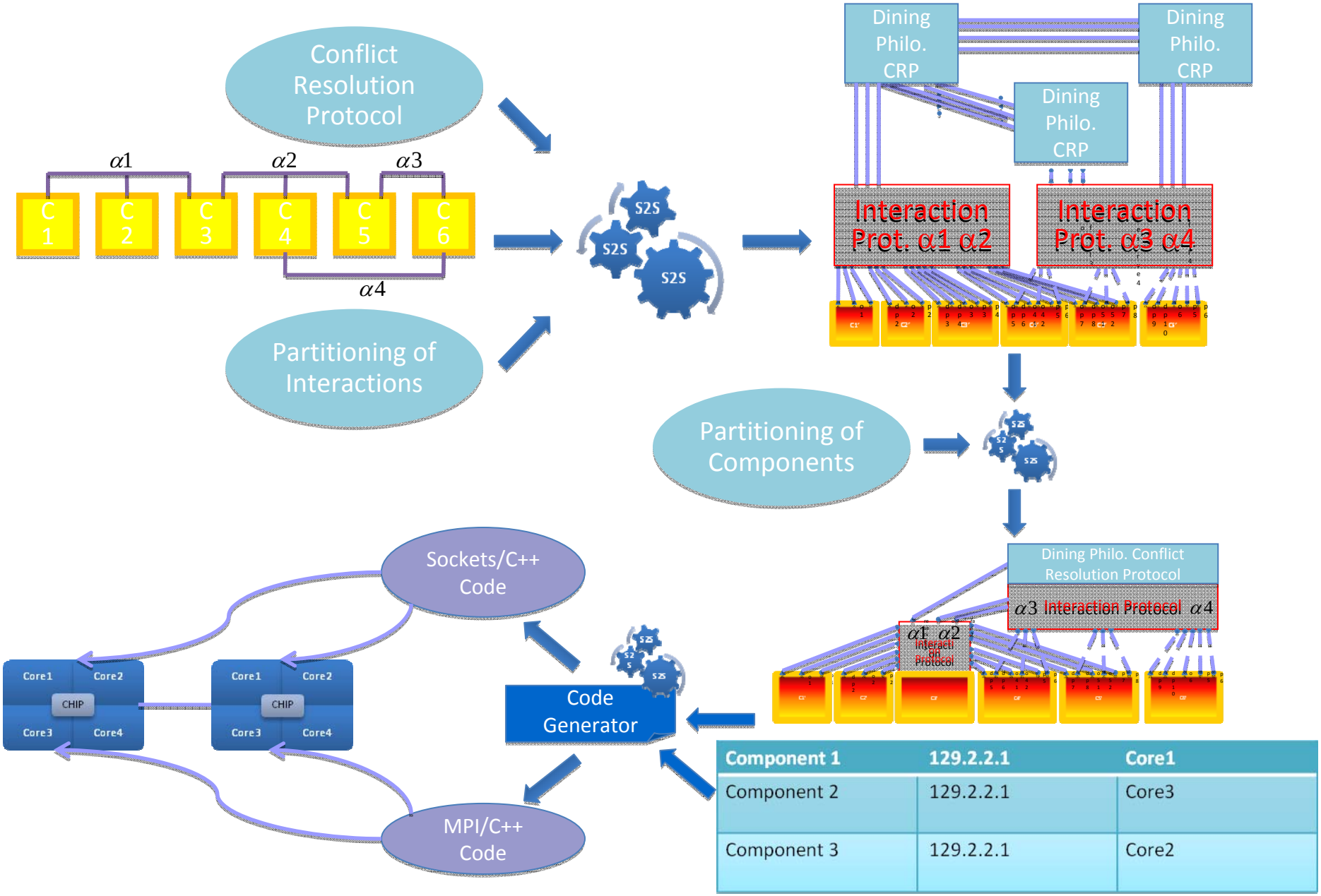
Putting RSD into Practice – Distributed Implementation



Distributed Implementation

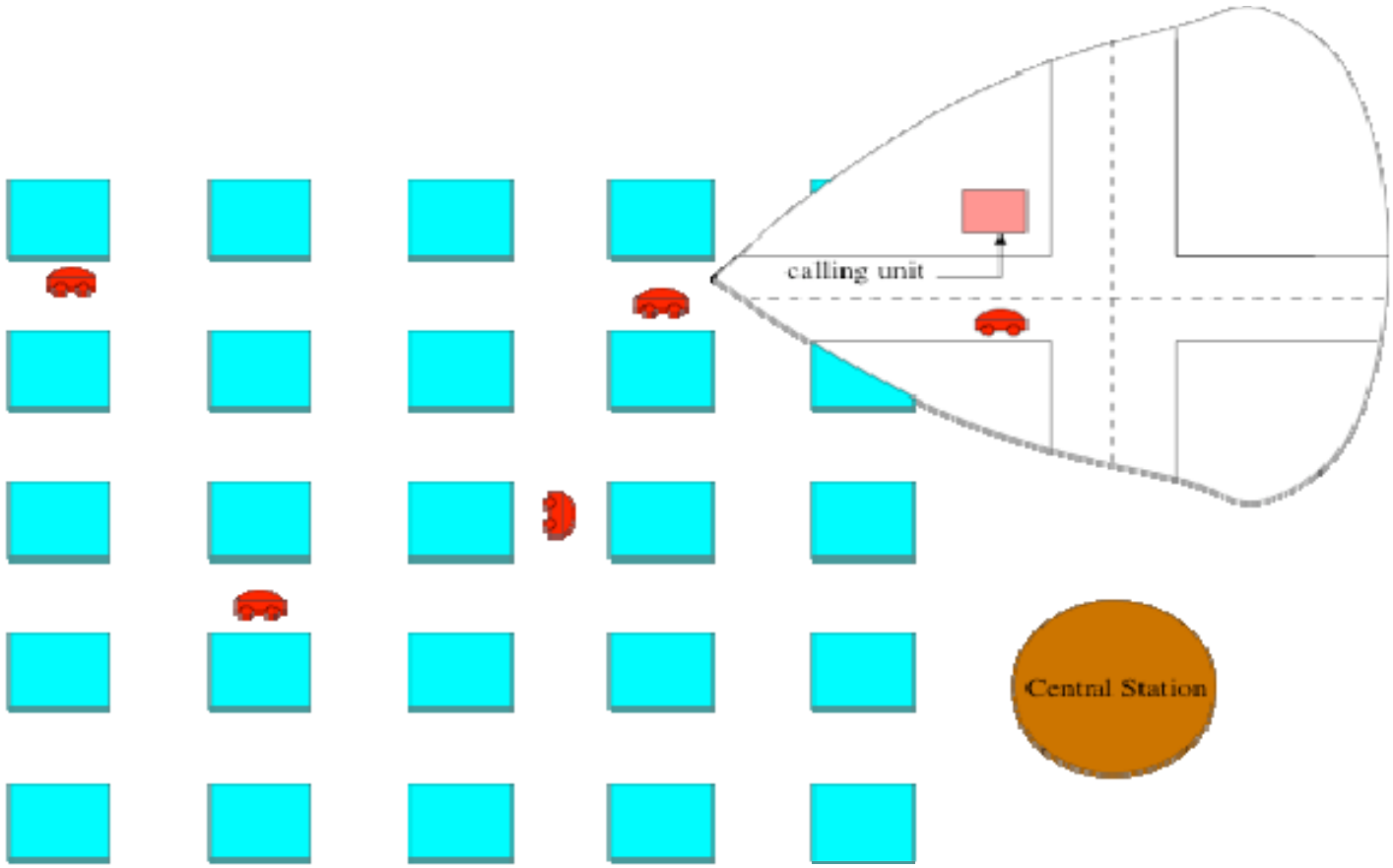


Putting RSD into Practice – Distributed Implementation

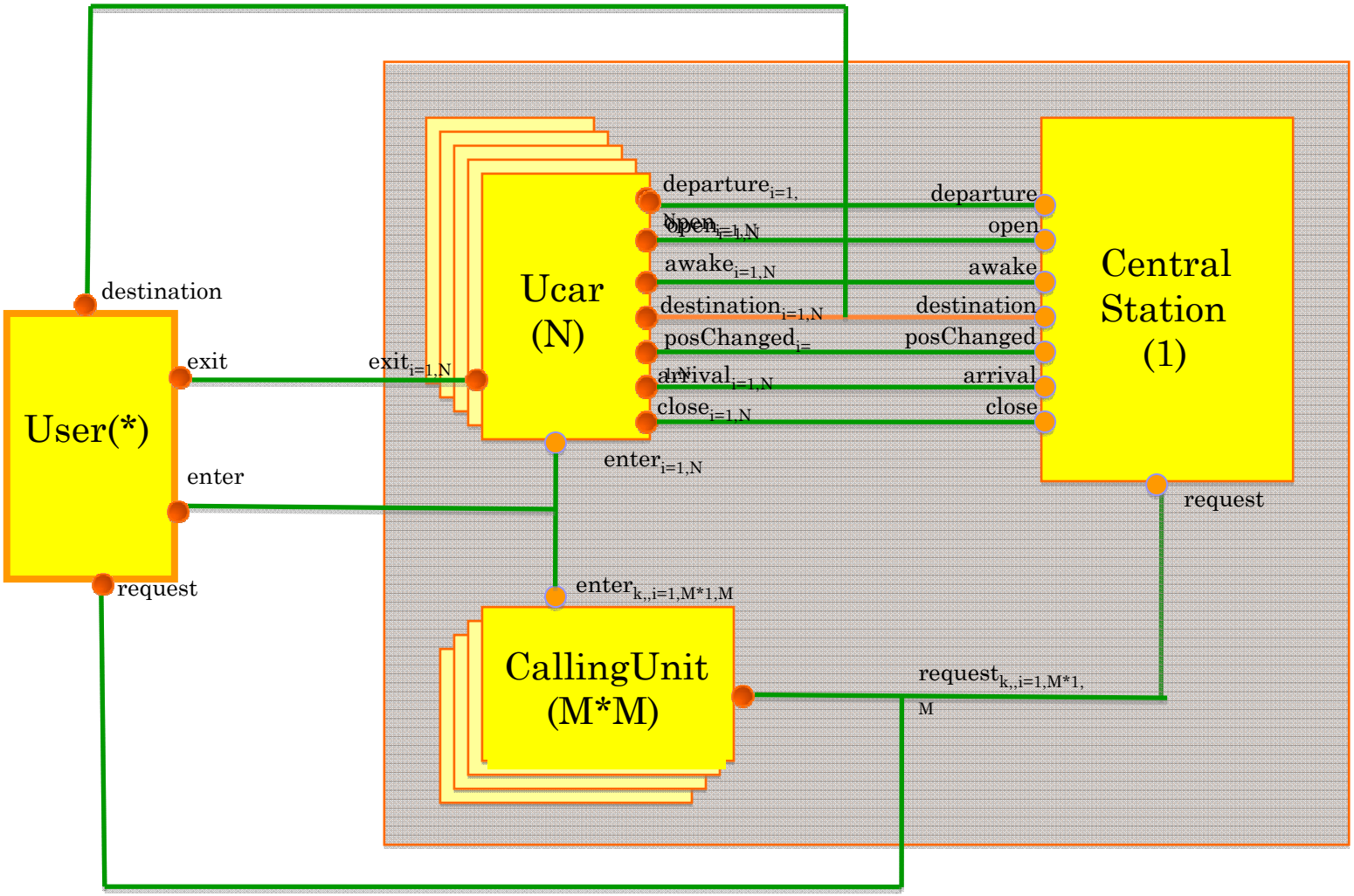


Putting RSD into Practice – Distributed Implementation

The UTOPAR system is an automated transportation system proposed by Israel Aircraft Industries in the COMBEST EU project



Putting RSD into Practice – Distributed Implementation

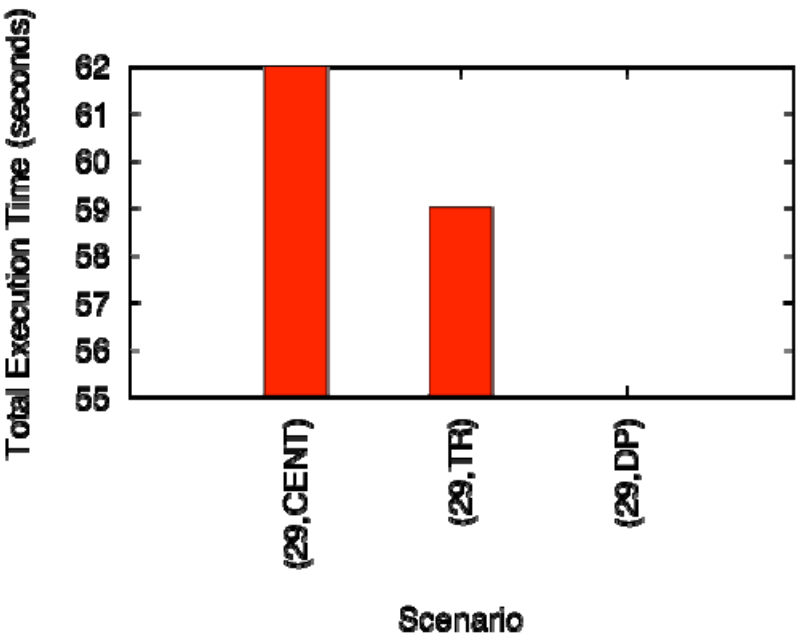


UTOPAR model in BIP

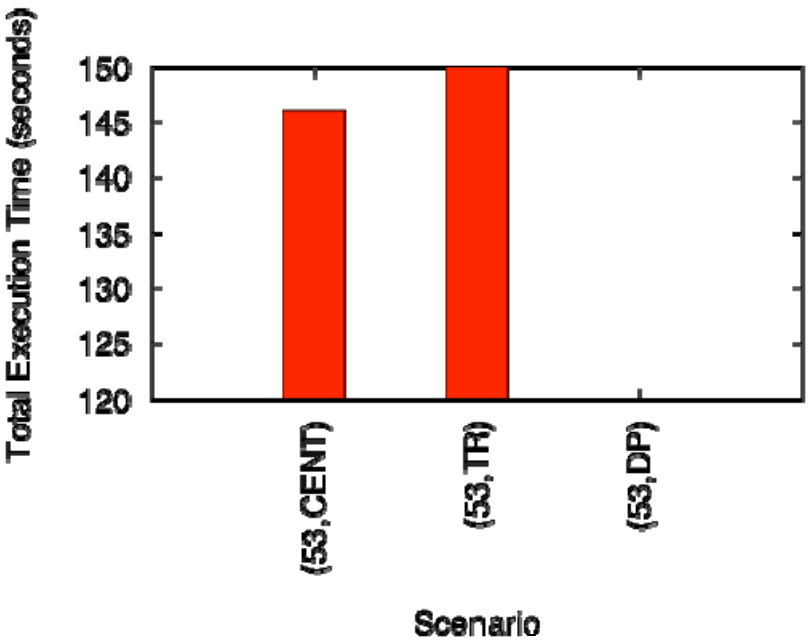
Putting RSD into Practice – Distributed Implementation

Benchmarks for fully automated generation of distributed C++ code for Linux sockets

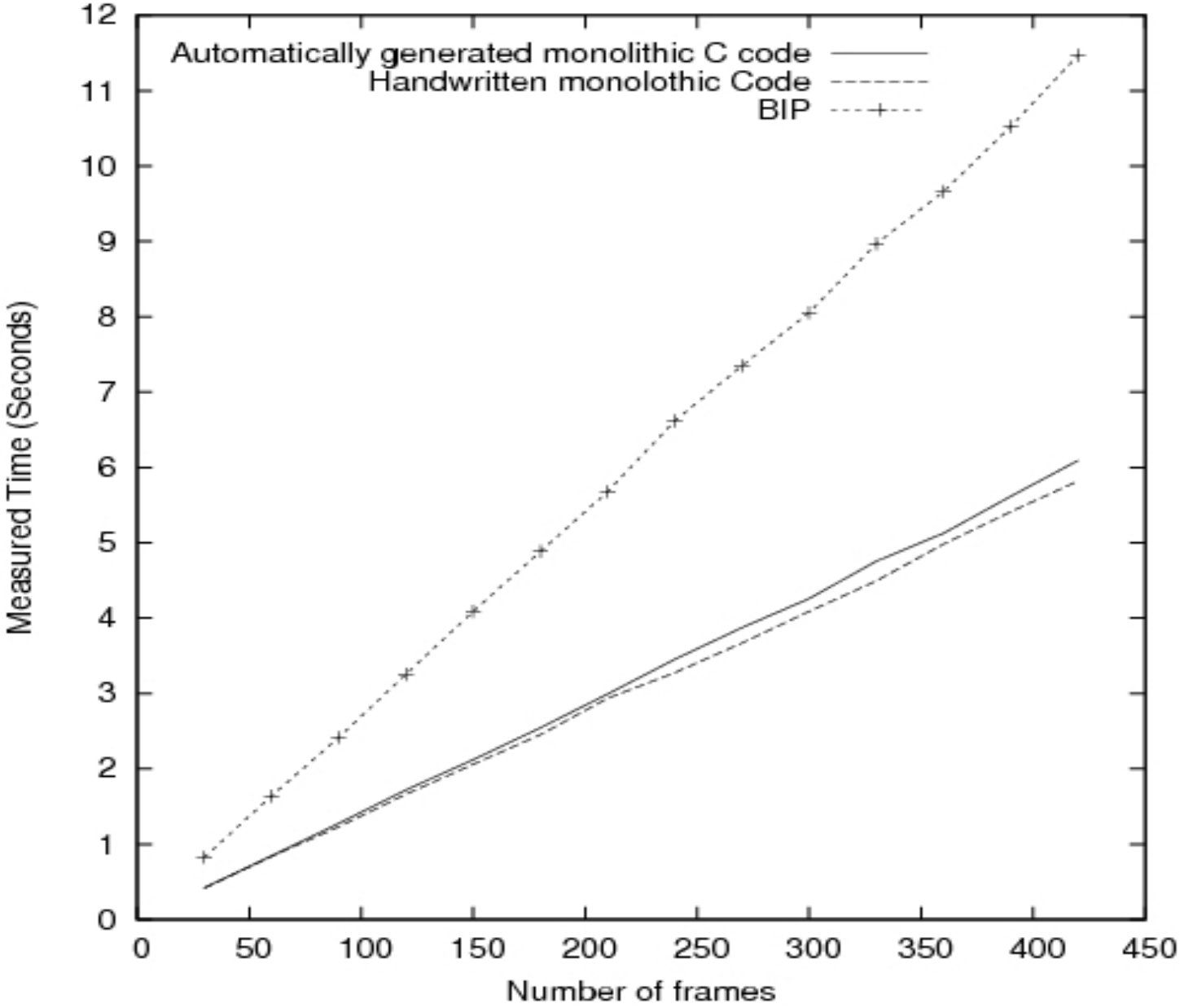
25 = 5 * 5 calling units and 4 cars – 29 Interaction Protocols



49 = 7 * 7 calling units and 4 cars – 53 Interaction Protocols



Putting RSD into Practice – Monolithic Code Generation

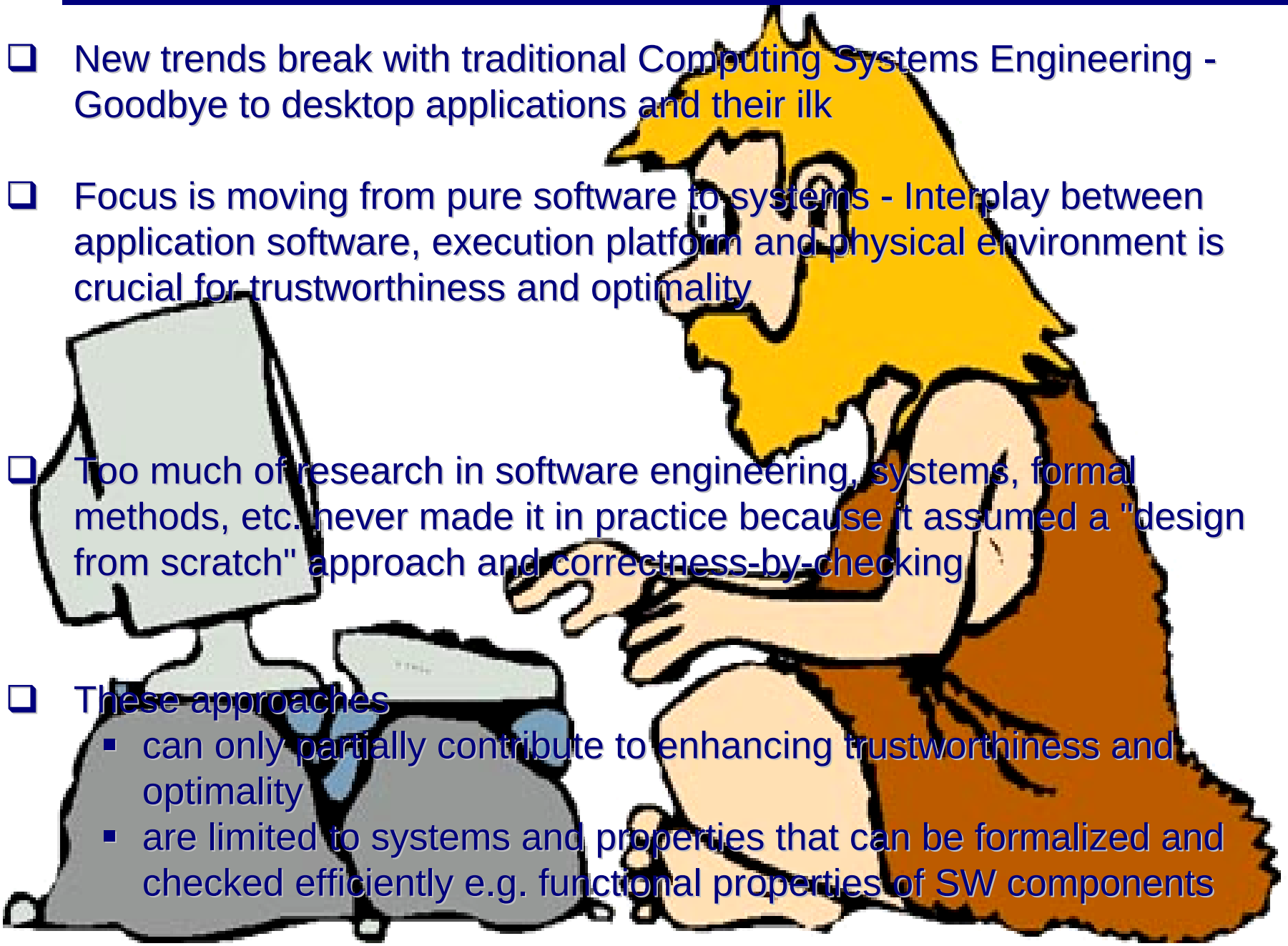


OVERVIEW

- System Design
- The Limits of Correctness-by-Checking
- Rigorous System Design
  - Multi-language Frameworks
  - Architectures
  - Automatic Implementation
- Putting RSD into Practice in BIP
- Discussion

Conclusion – Breaking with Outdated Ideas

- New trends break with traditional Computing Systems Engineering - Goodbye to desktop applications and their ilk
- Focus is moving from pure software to systems - Interplay between application software, execution platform and physical environment is crucial for trustworthiness and optimality
- Too much of research in software engineering, systems, formal methods, etc. never made it in practice because it assumed a "design from scratch" approach and correctness-by-checking
- These approaches
  - can only partially contribute to enhancing trustworthiness and optimality
  - are limited to systems and properties that can be formalized and checked efficiently e.g. functional properties of SW components



Conclusion – For a Discipline of Design

New approach for the design of trustworthy and optimal systems

- Assumes that
  - systems are built/modified incrementally by integrating components bottom-up
  - correctness can be effectively enforced by construction
  - the Software IS the Model
- Raises challenging and relevant fundamental problems with high potential for innovation and impact on languages, architectures, implementation techniques
- Endeavors unification through formalization of design as a process
  - for deriving trustworthy and optimal implementations from an application software and models of its execution platform and physical environment
  - which is semantically sound, scalable and accountable
- Opens the way for moving from empirical design techniques to a well-founded design discipline

Discussion – Putting RSD into Practice in BIP

The BIP component framework has been developed for more than 10 years, with Rigorous Design in mind

Main achievements:

- Translation of several DSL (Simulink, Lustre, DOL, nesC) into BIP
- Rigorous design flow based on source-to-source transformations proven correct-by-construction, in particular for
  - taking into account HW resources
  - generating distributed implementations for several platforms
  - code optimization
- Run-times for centralized execution/simulation, distributed execution, real-time execution
- Validation and analysis tools
  - Incremental checking for Deadlock-freedom: D-Finder tool
  - Statistical Model Checking
- Successful application in many industrial projects
  - software componentization for robotic systems (DALA Space Robot for Astrium)
  - programming multi-core systems (P2012 for ST Microelectronics, MPPA for Kalray)
  - complex systems modeling (AFDX and IMA for Airbus)

Thank You