Generating Distributed Implementations for BIP

WFCD 10

Scottsdale, October 24, 2010

Joseph Sifakis

VERIMAG Laboratory

in collaboration with

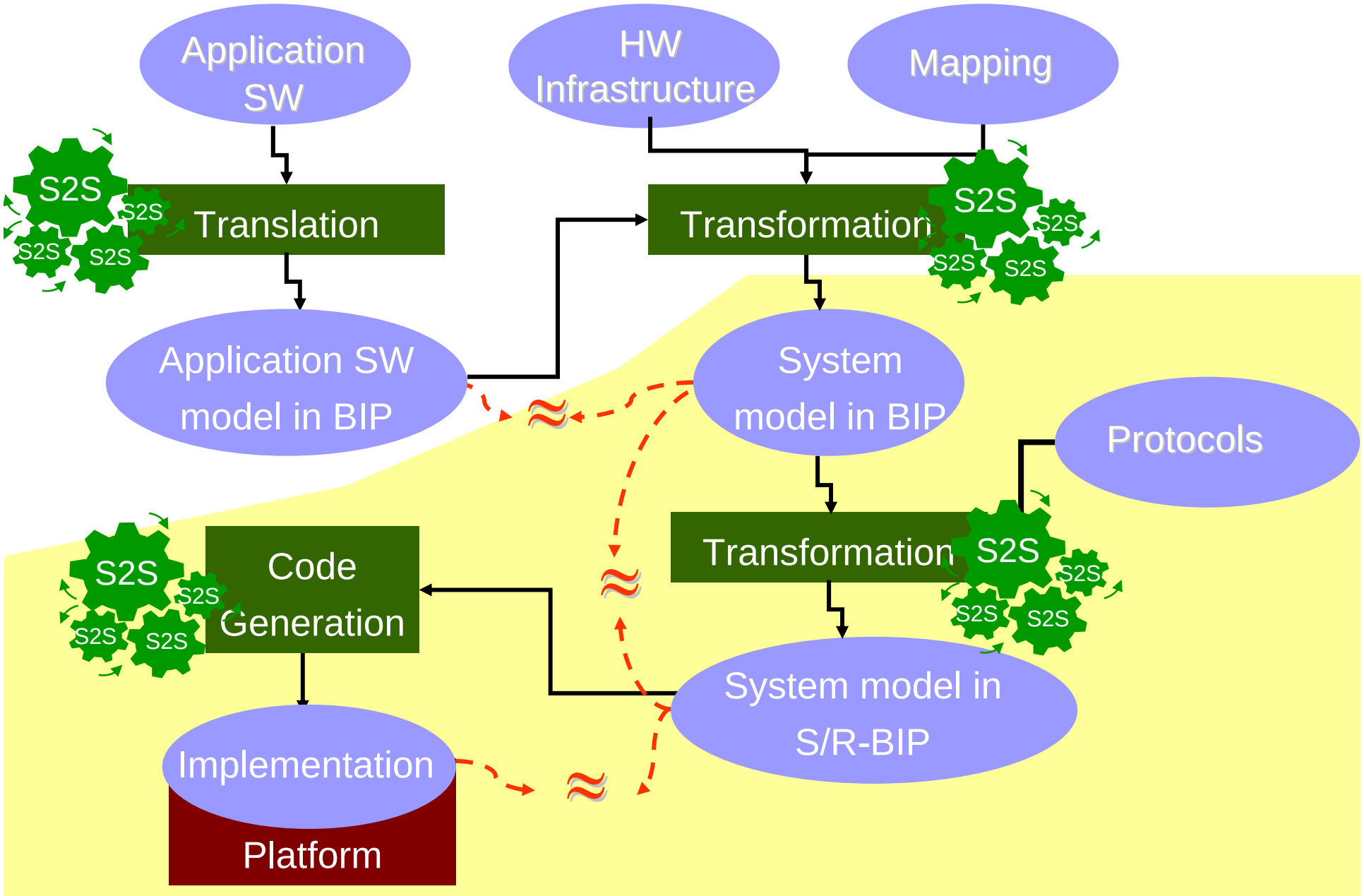
A. Basu, B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf

Rigorous System Design

A rigorous system design flow allows guaranteeing that the designed system meets some essential requirements. It is

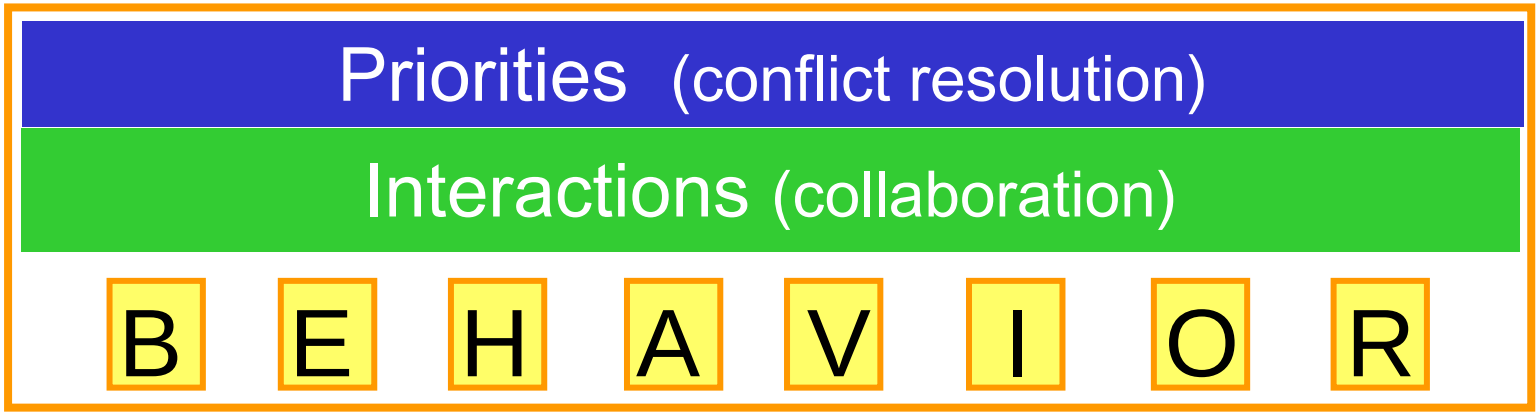
- **model-based**: the software and system descriptions used along the design flow should be based on a single semantic model. This is essential for the overall coherency and efficiency
  - Relate system descriptions and their properties for different abstraction levels and purposes (validation, performance evaluation, code generation)
- **component-based**: it provides primitives for building composite components as the composition of simpler heterogeneous components.
- **correct-by-construction**: it rely on tractable theory for guaranteeing at design time essential properties so as to avoid as much as possible monolithic *a posteriori* validation

Rigorous System Design in BIP– Design Flow

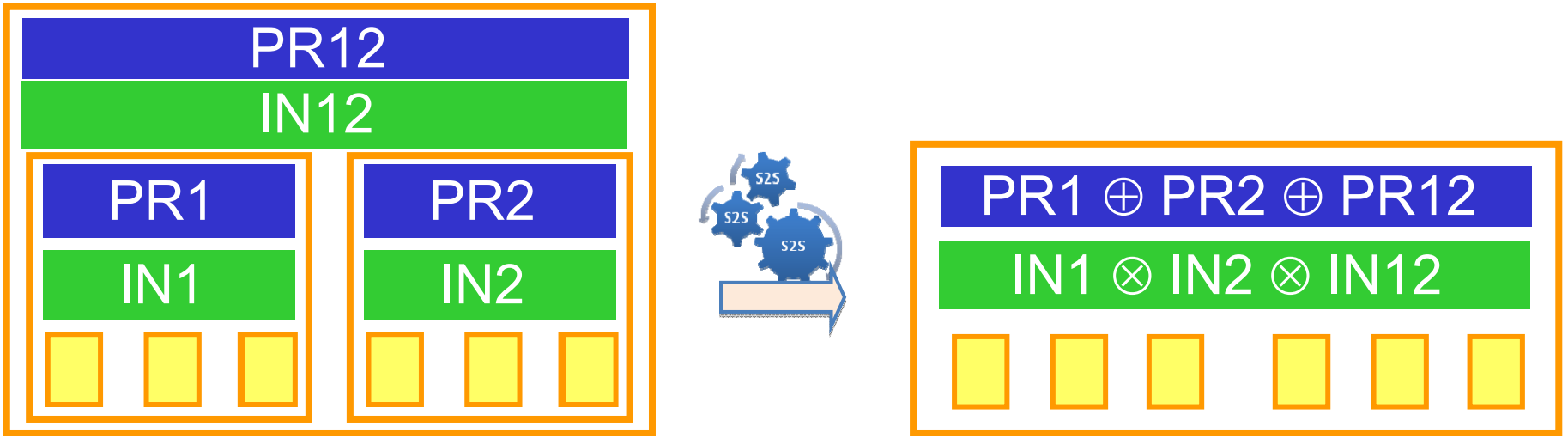


BIP – Basic Concepts

Layered component model



Composition operation parameterized by glue IN12, PR12

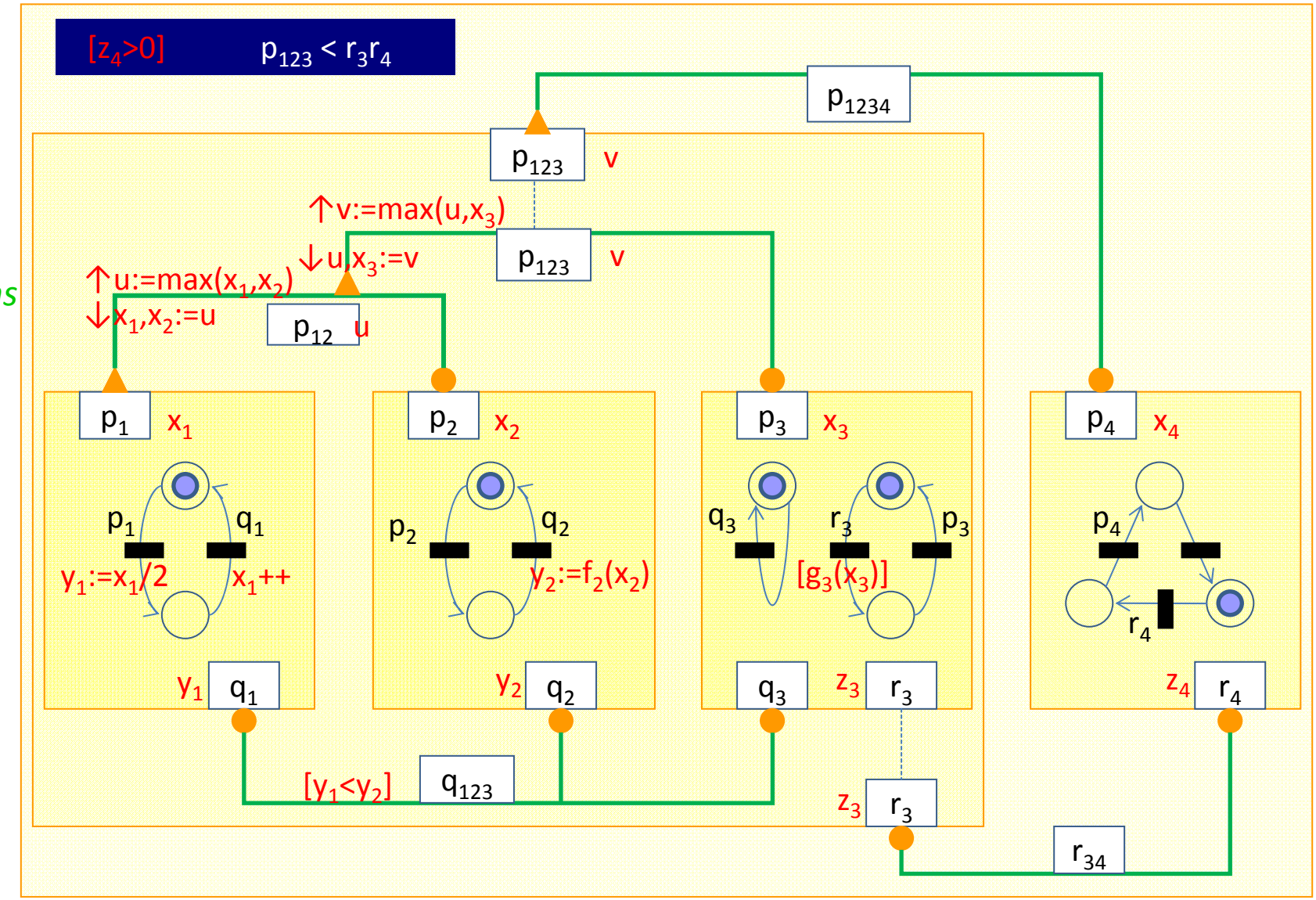


BIP in a Nutshell

Priorities

Interactions

Behavior



OVERVIEW

- Distributed Model Generation
- Monolithic code generation
- Examples
- Discussion

The S/R-BIP model

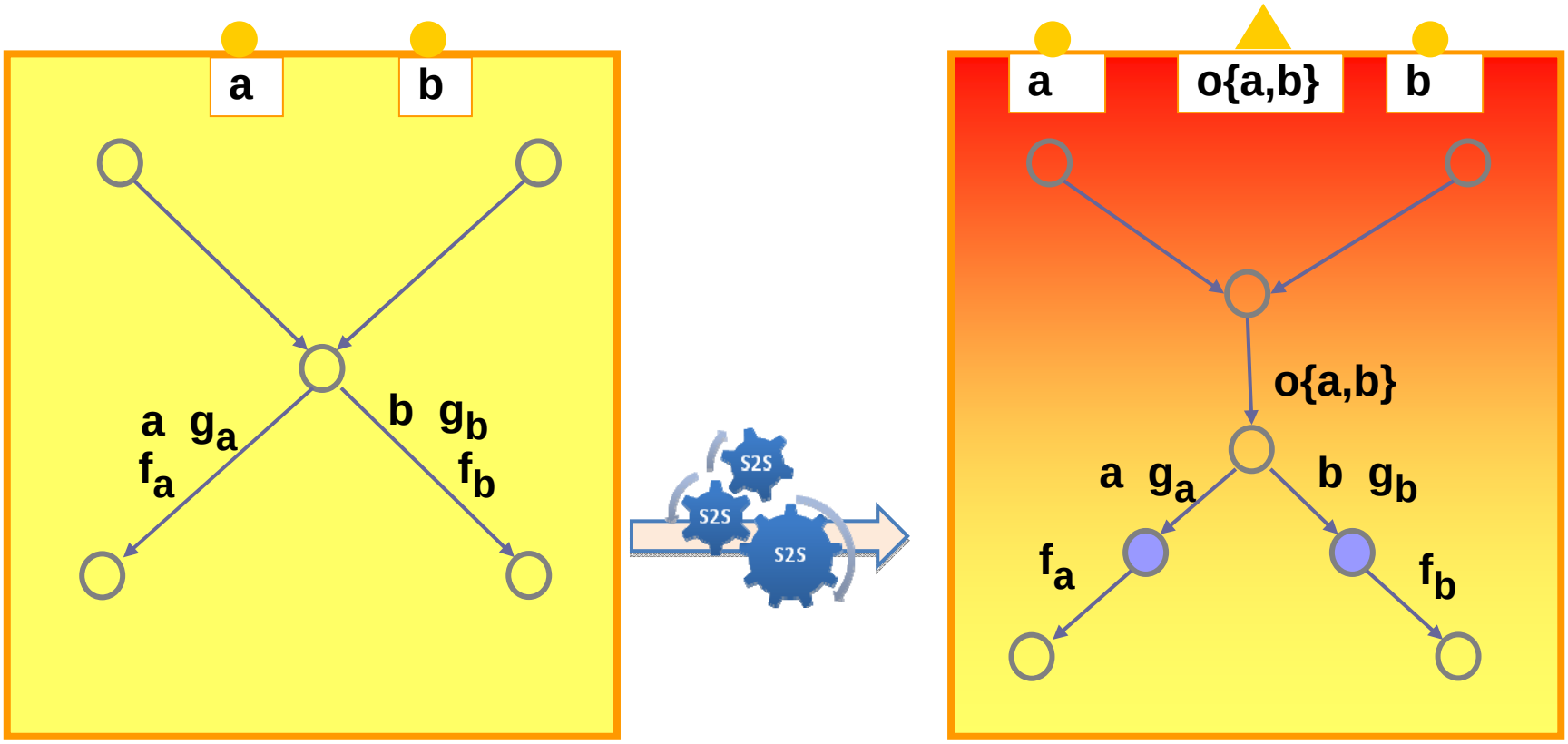
BIP is based on:

- Global state semantics, defined by operational semantics rules, implemented by the Engine
- Atomic multiparty interactions, e.g. by rendezvous or broadcast

Translate BIP models into observationally equivalent S/R-BIP models

- Collection of independent components intrinsically concurrent - No global state
  - Atomicity of transitions is broken by separating interaction from internal computation
- Point to point communication by asynchronous message passing
- Translation is correct-by-construction

From BIP to S/R-BIP – Atomic Components



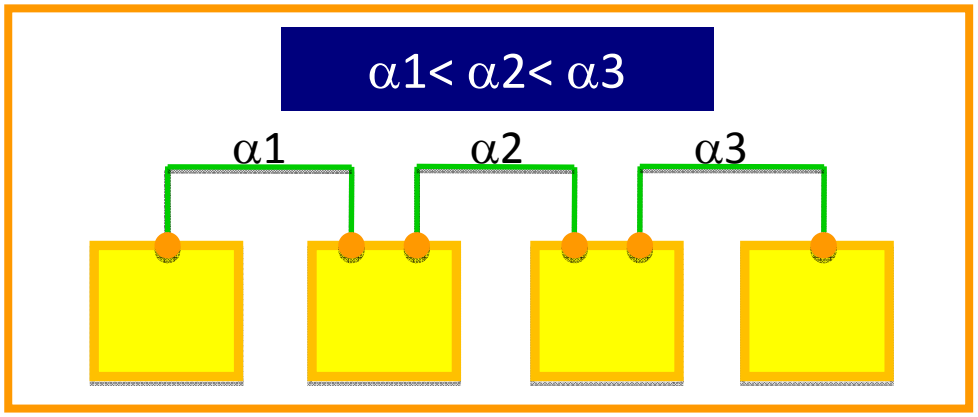
Global state model

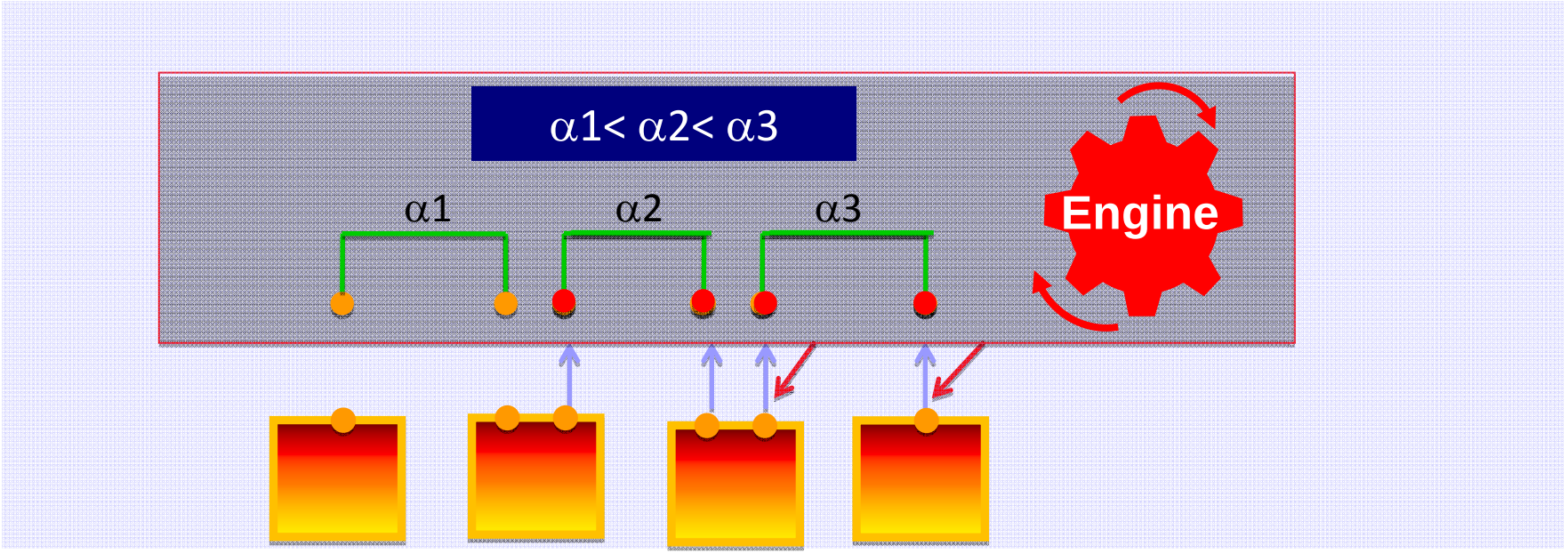
Partial state model

- Before reaching a ready state, the set of the enabled ports is sent to the Engine
- From a ready state, await notification from the Engine indicating the selected port

Single Engine (Forte 2008)

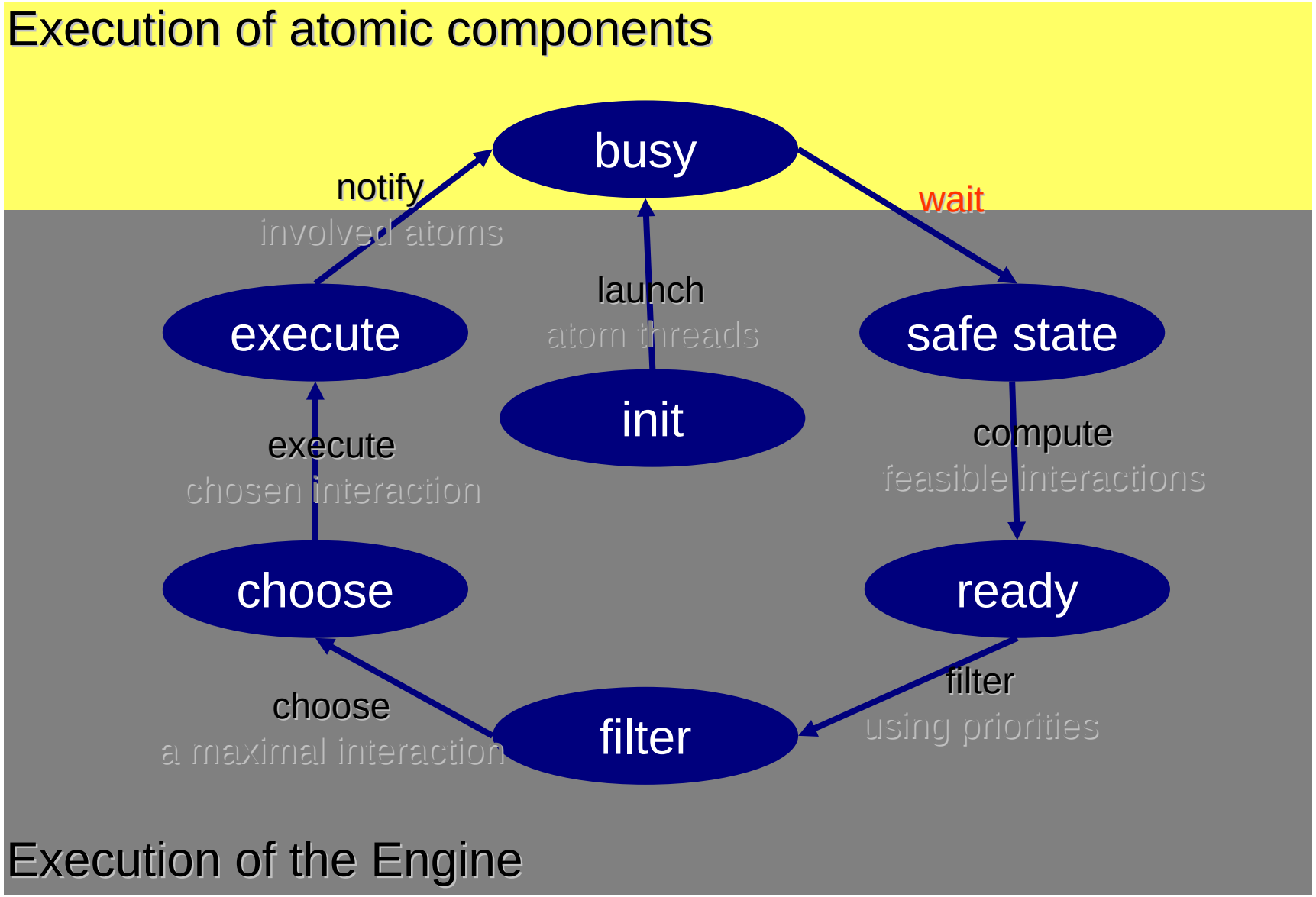
BIP Model





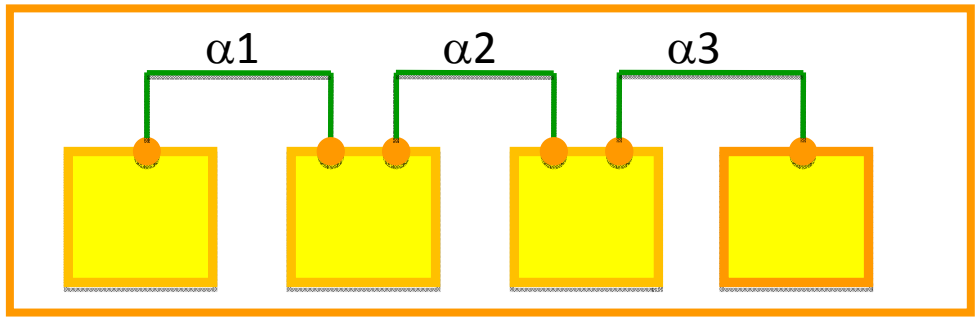
One Engine executes *all* interactions !

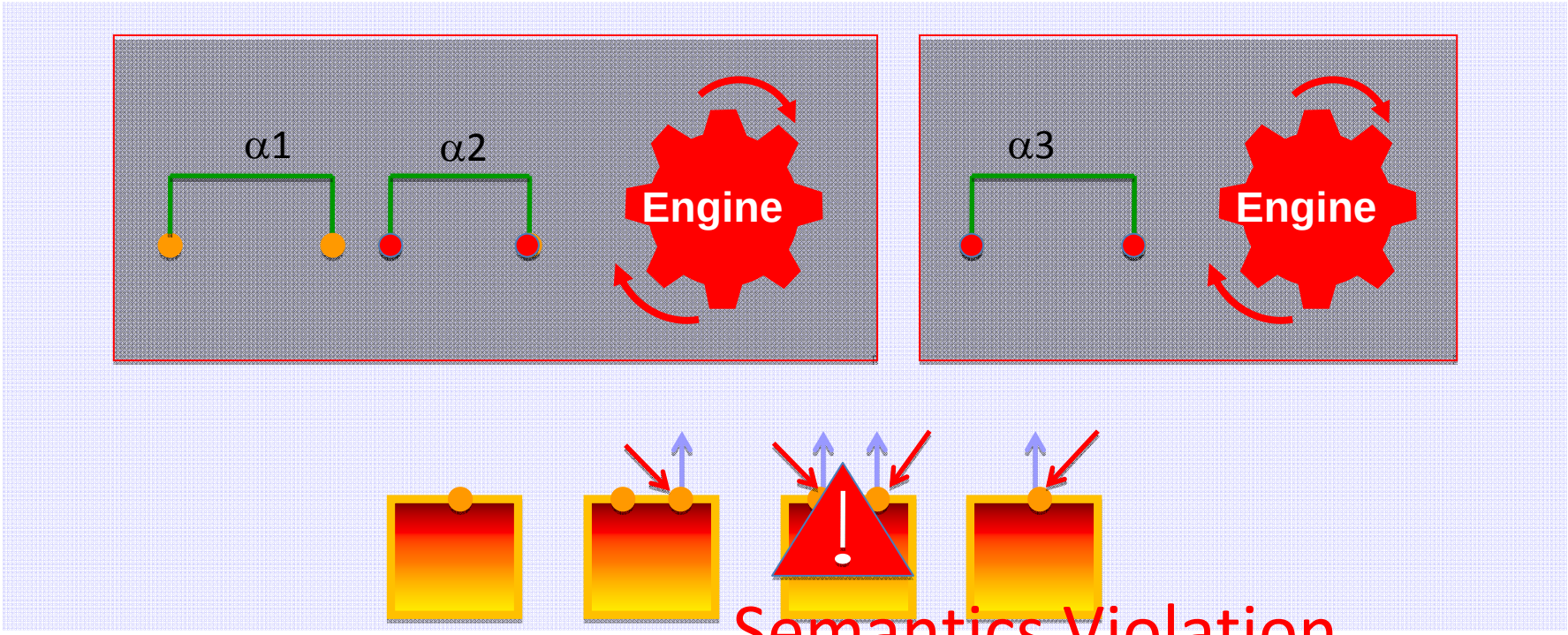
Single Engine Execution



Multiple Engines

BIP Model

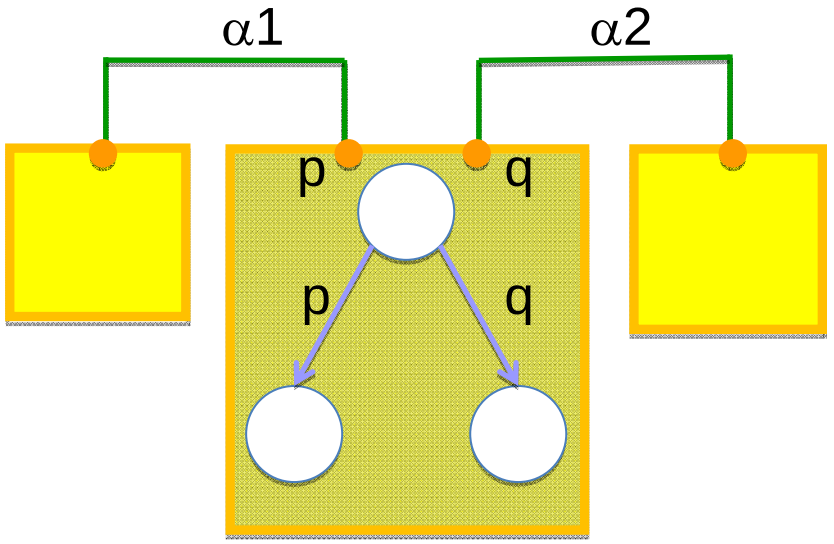




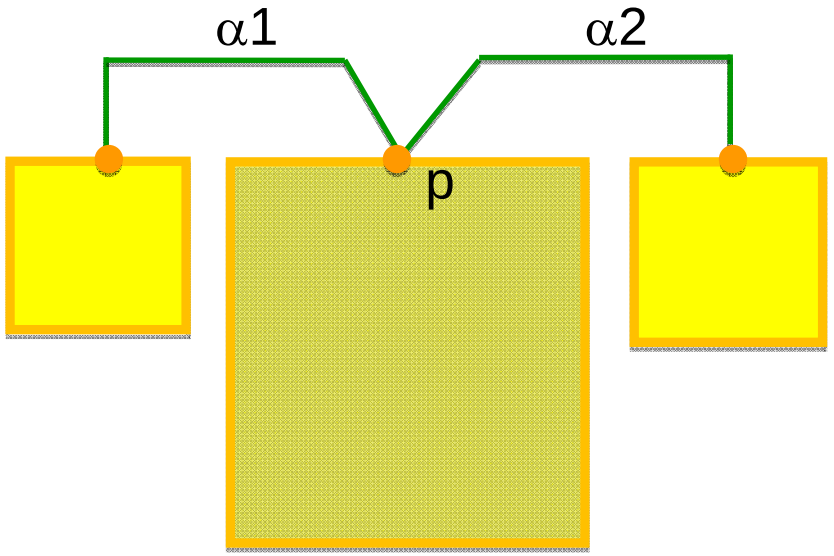
Semantics Violation

Dispatch interactions across *multiple* engines !

Multiple Engines – Conflicting Interactions



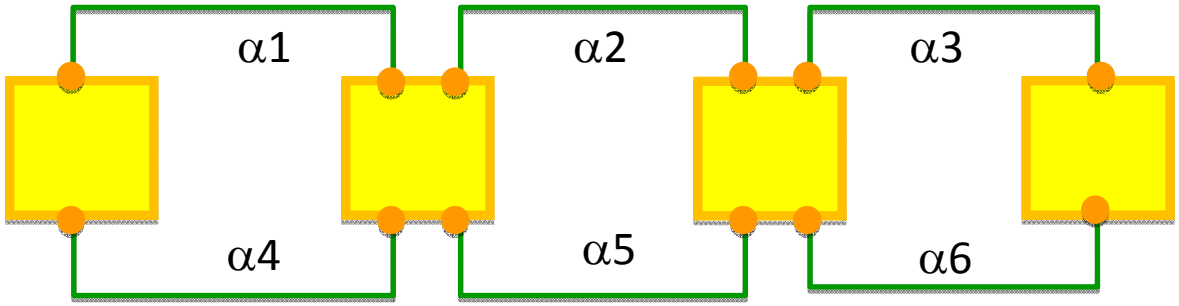
α1 and α2 depend on conflicting transitions



α1 and α2 share a common port p

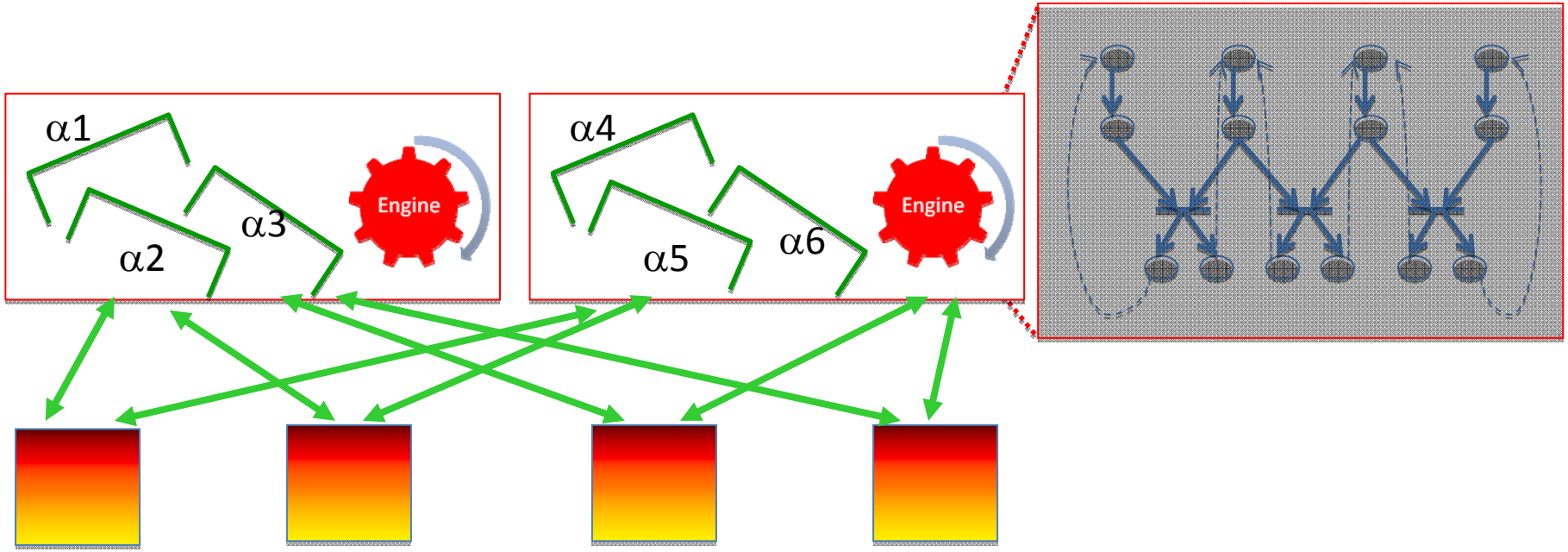
α1 # α2: the interactions α1 and α2 may be in conflict

Conflict-Free Multiple Engines



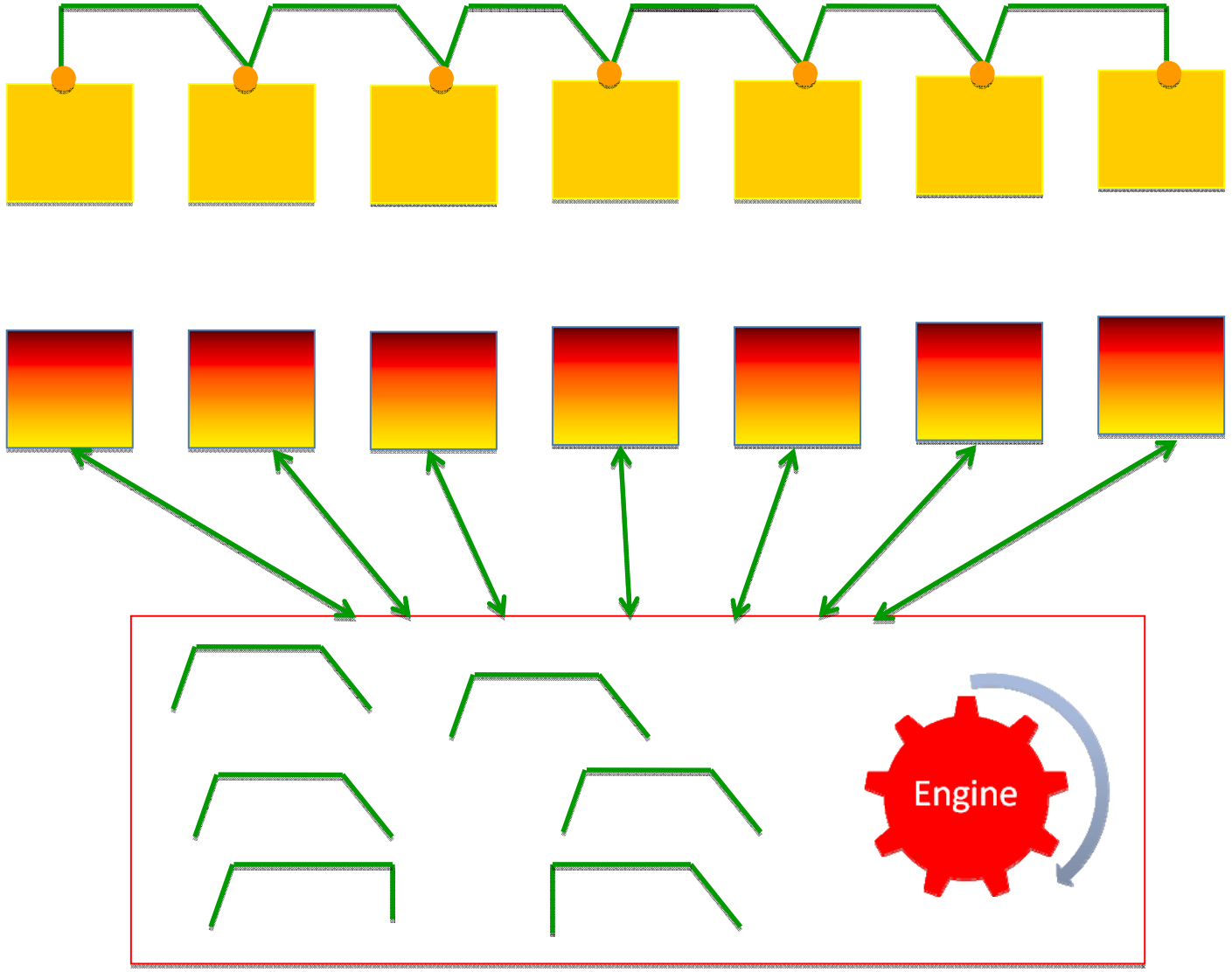
α1 # α2 # α3

α4 # α5 # α6



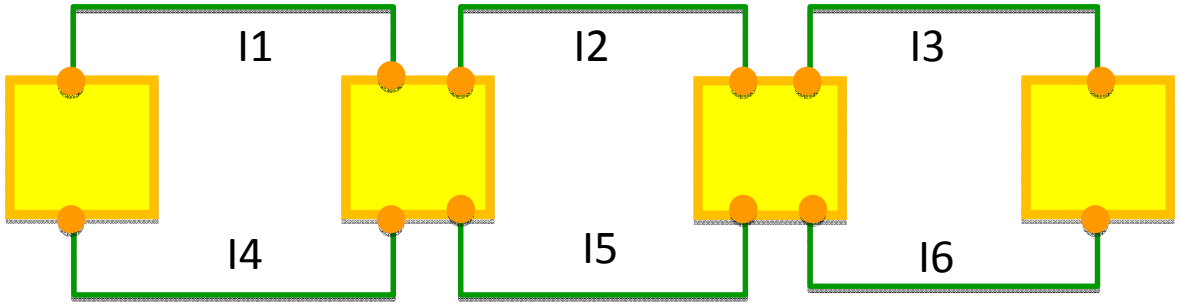
Each engine handles interactions of a class of #*

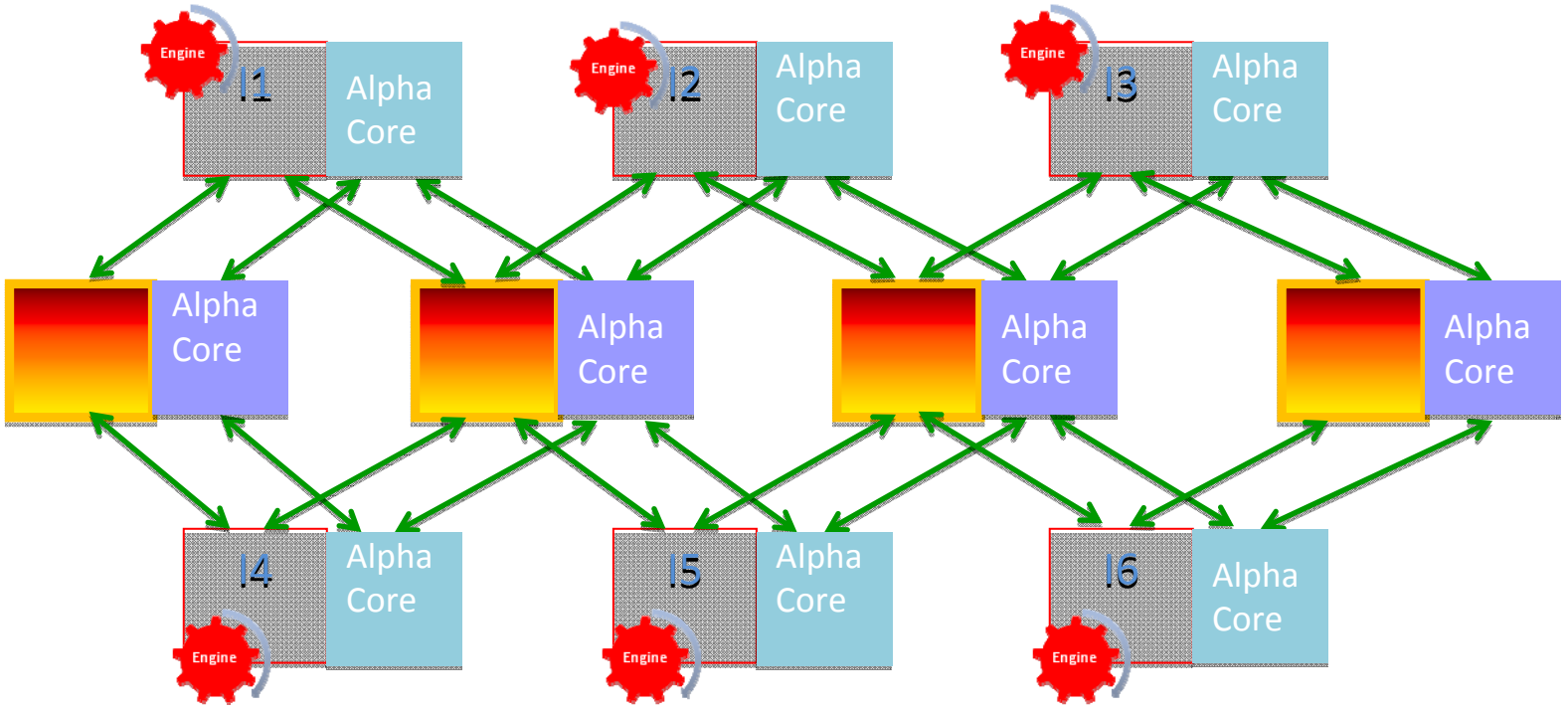
Conflict-Free Multiple Engines – Problem



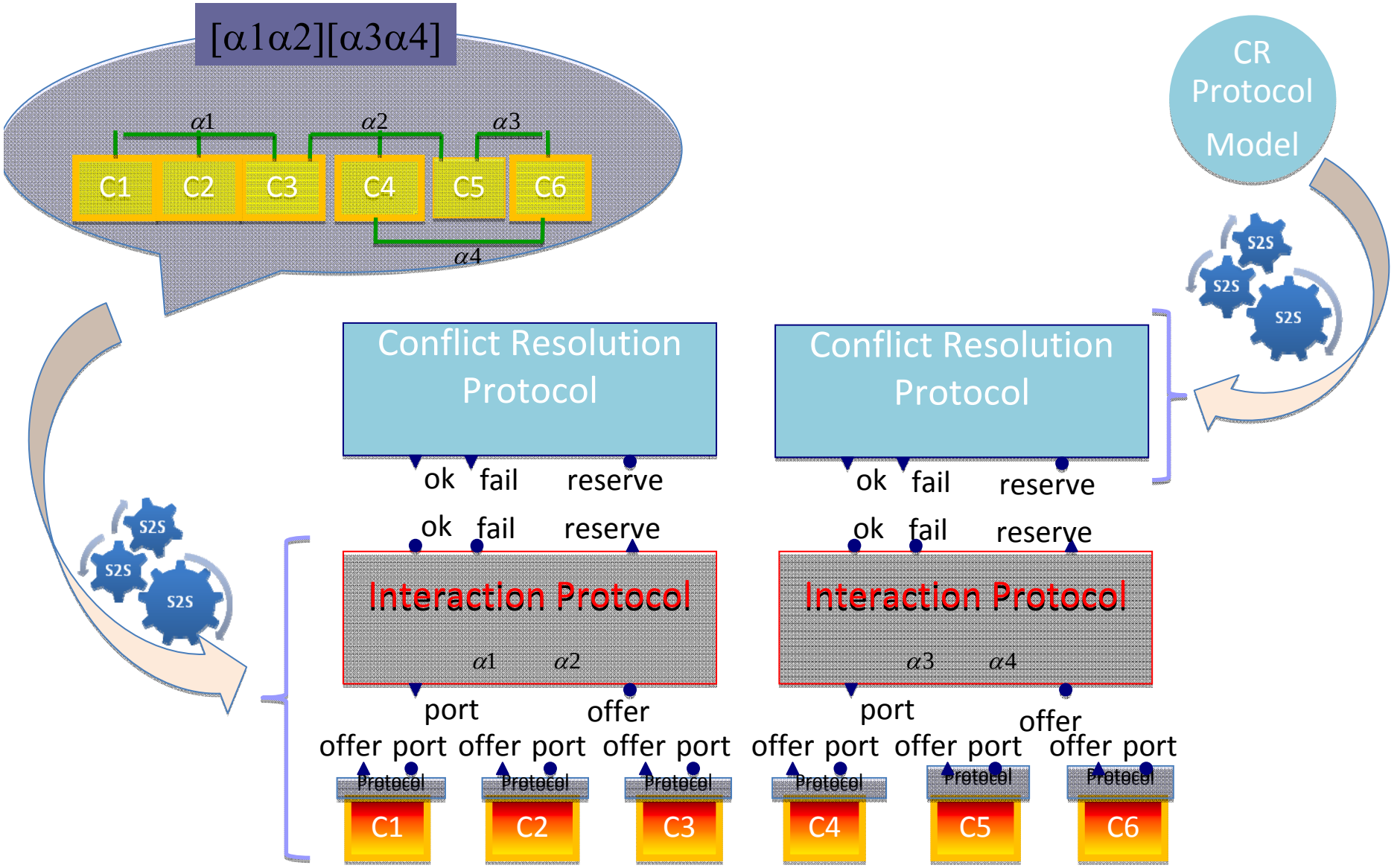
- Taking #* may reduce drastically parallelism between interactions

Multiple Engines - Using the AlphaCore Protocol

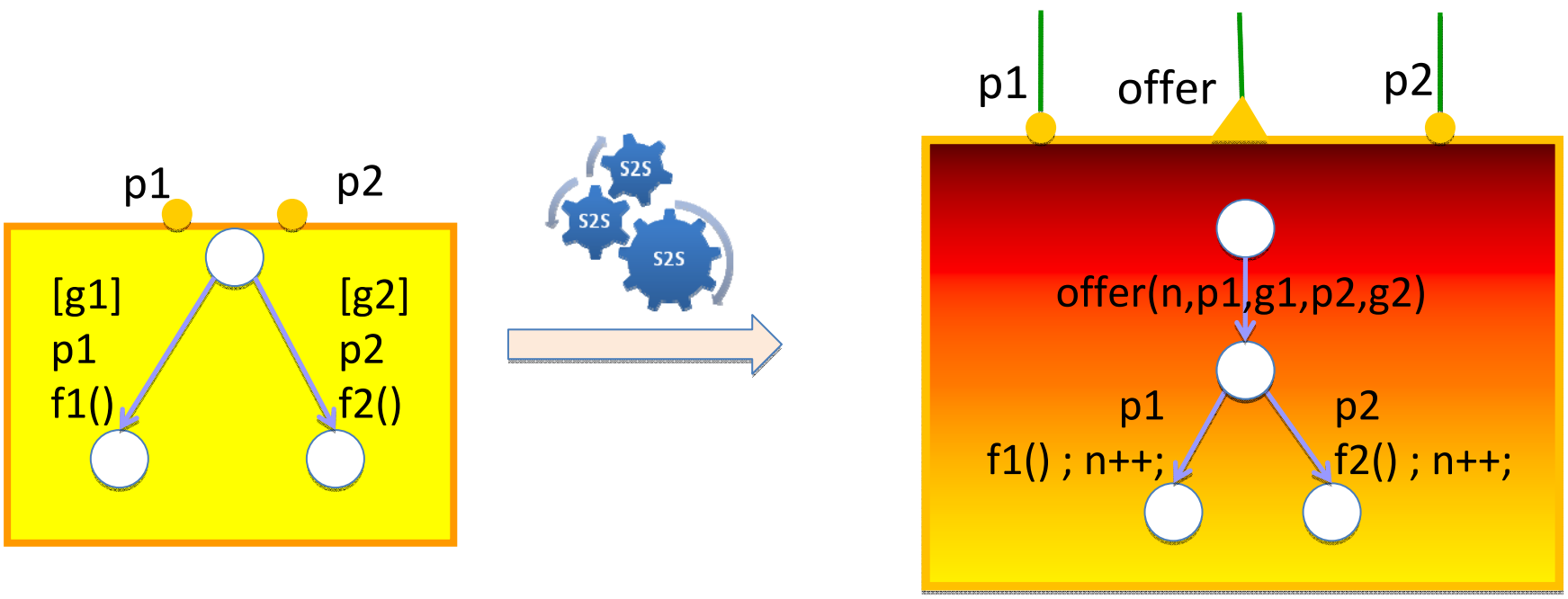




3-Layer Architecture

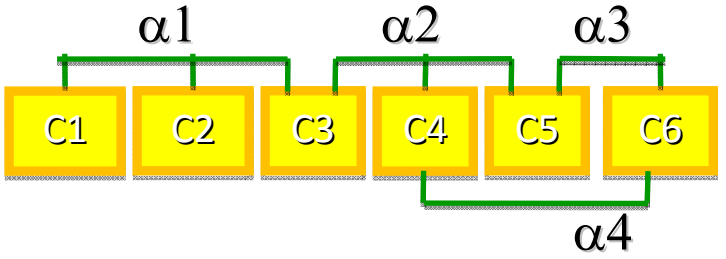


3-Layer Architecture – Atomic Components



n: counts the number
of executed transitions

3 Layer Architecture – Centralized CRP



Centralized Conflict Resolution Protocol

reserve2 ok2 fail2 reserve3 ok3 fail3 reserve4 ok4 fail4

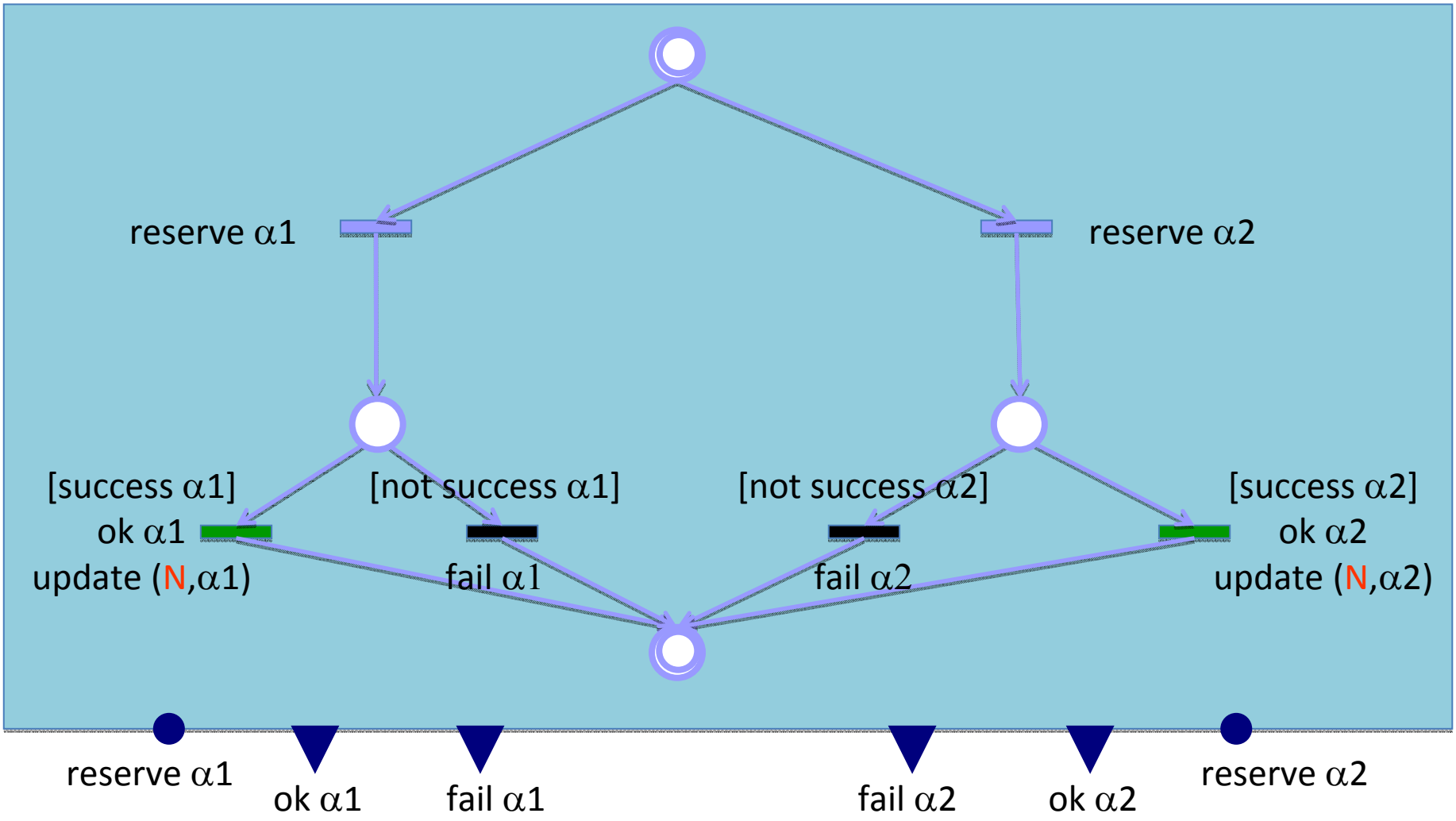
$\alpha 1$ Interaction Protocol $\alpha 2$

$\alpha 3$ Interaction Protocol $\alpha 4$

o1 p1 o2 p2 o3 p3 p4 o41 o42 p5 p6 o51 o52 p7 p8 o6 p5 p6

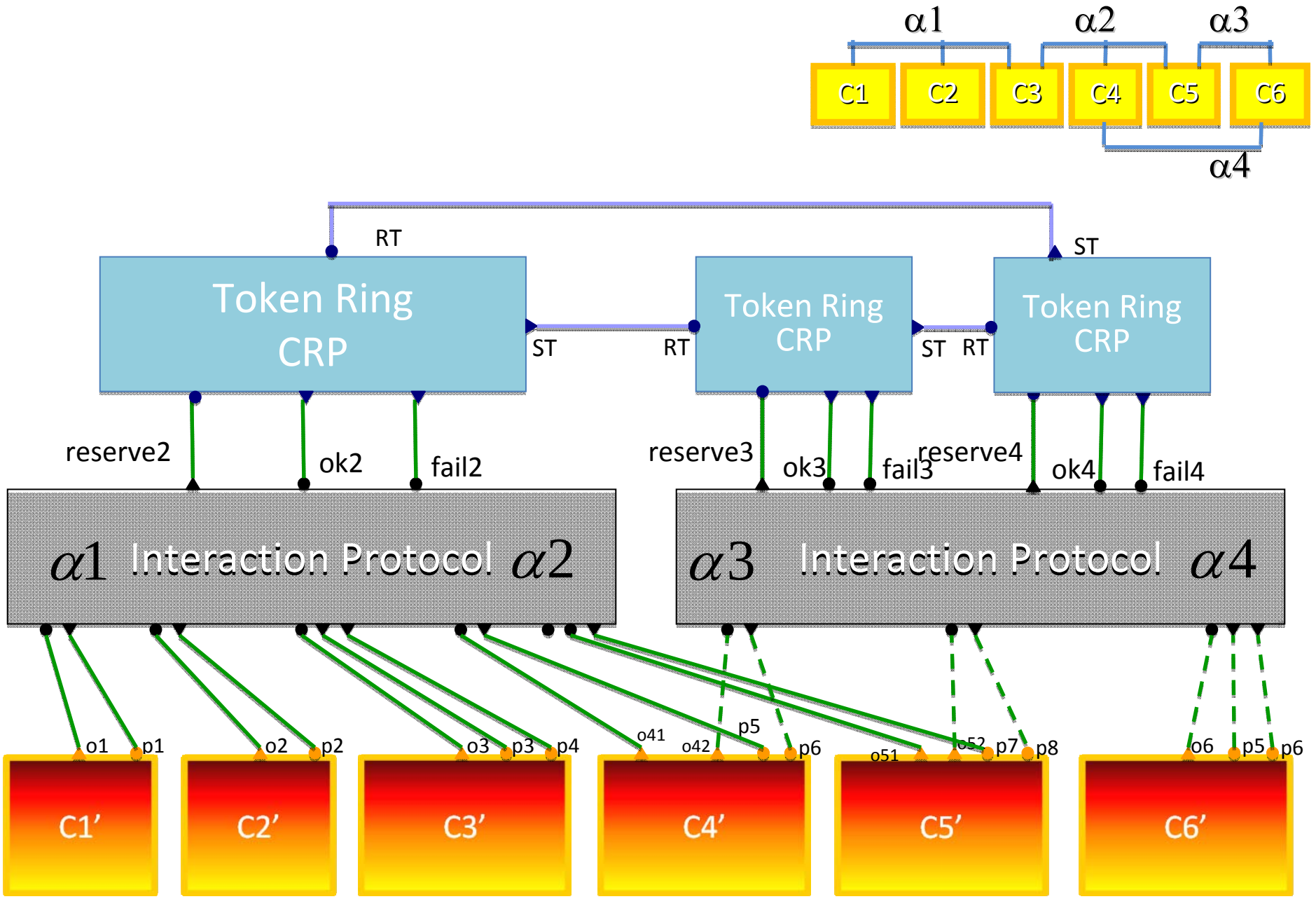
C1' C2' C3' C4' C5' C6'

3-Layer Architecture – Centralized CRP

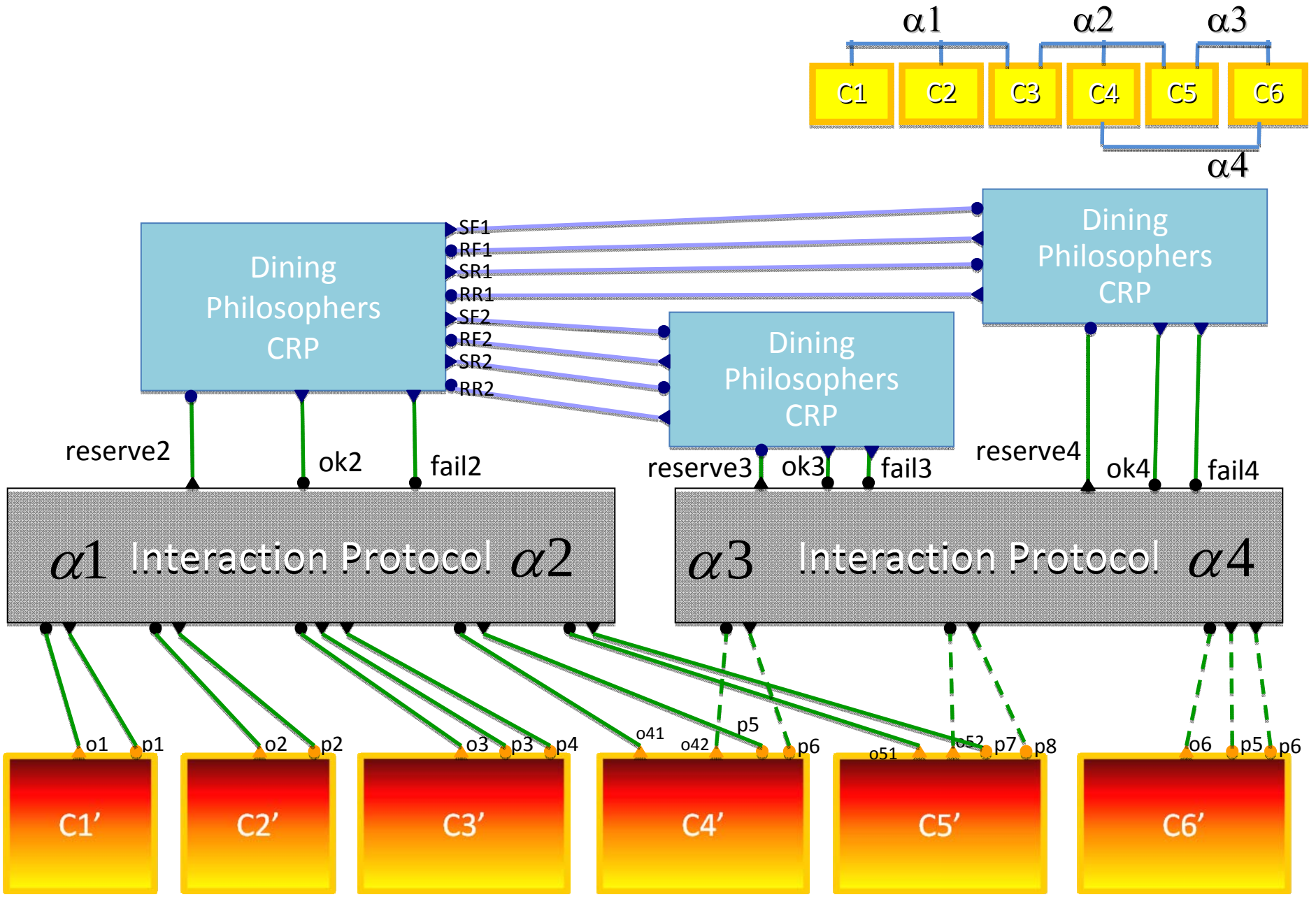


N=[N1, …, Nk]: keeps track of the state of the counters n of the components

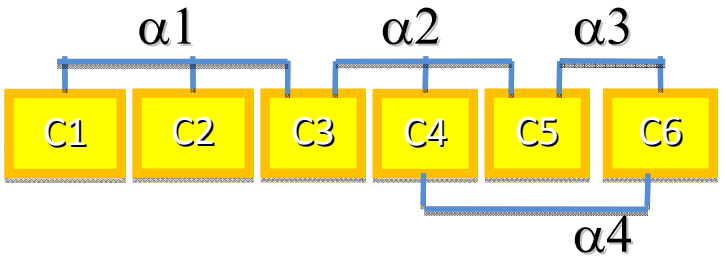
3-Layer Architecture – Token Ring CRP

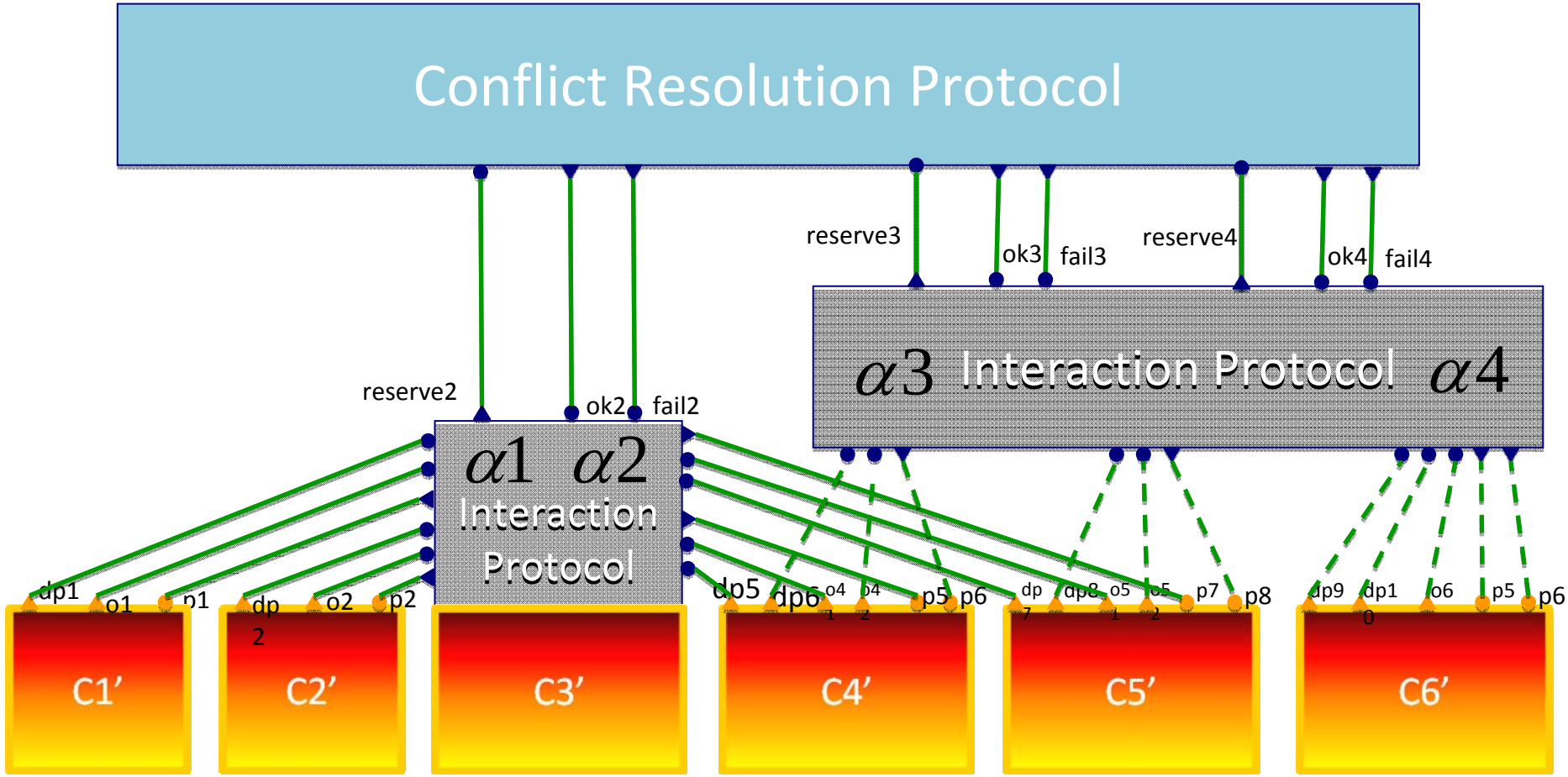


3-Layer Distributed BIP – Dining Philosophers CRP



Monolithic Code Generation – Component Partitioning

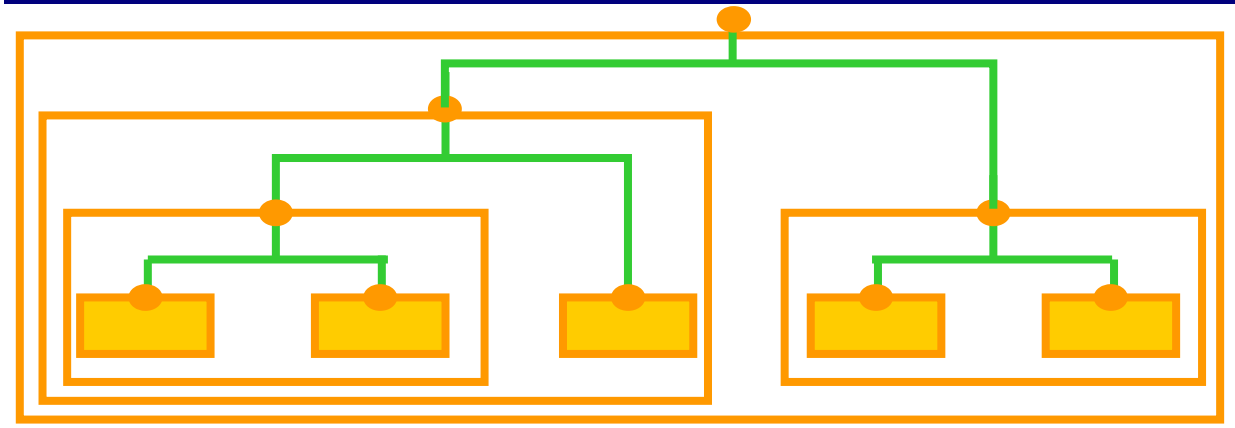


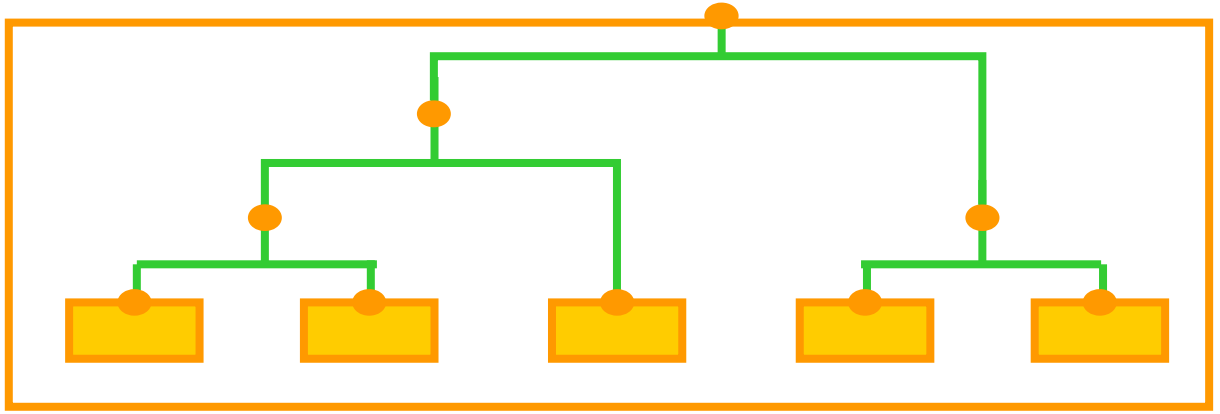


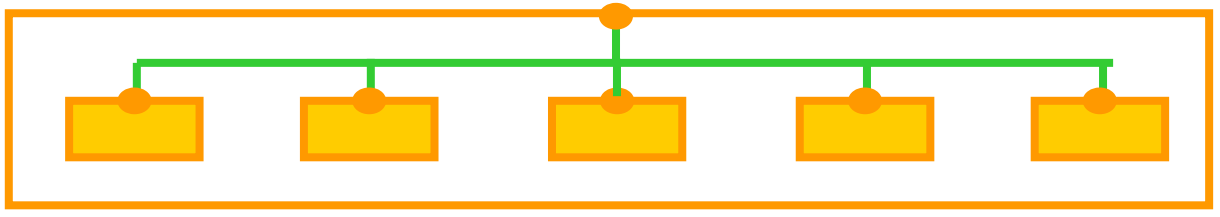
OVERVIEW

- Distributed Model Generation
- Monolithic code generation
- Examples
- Discussion

From Composite Components to C code



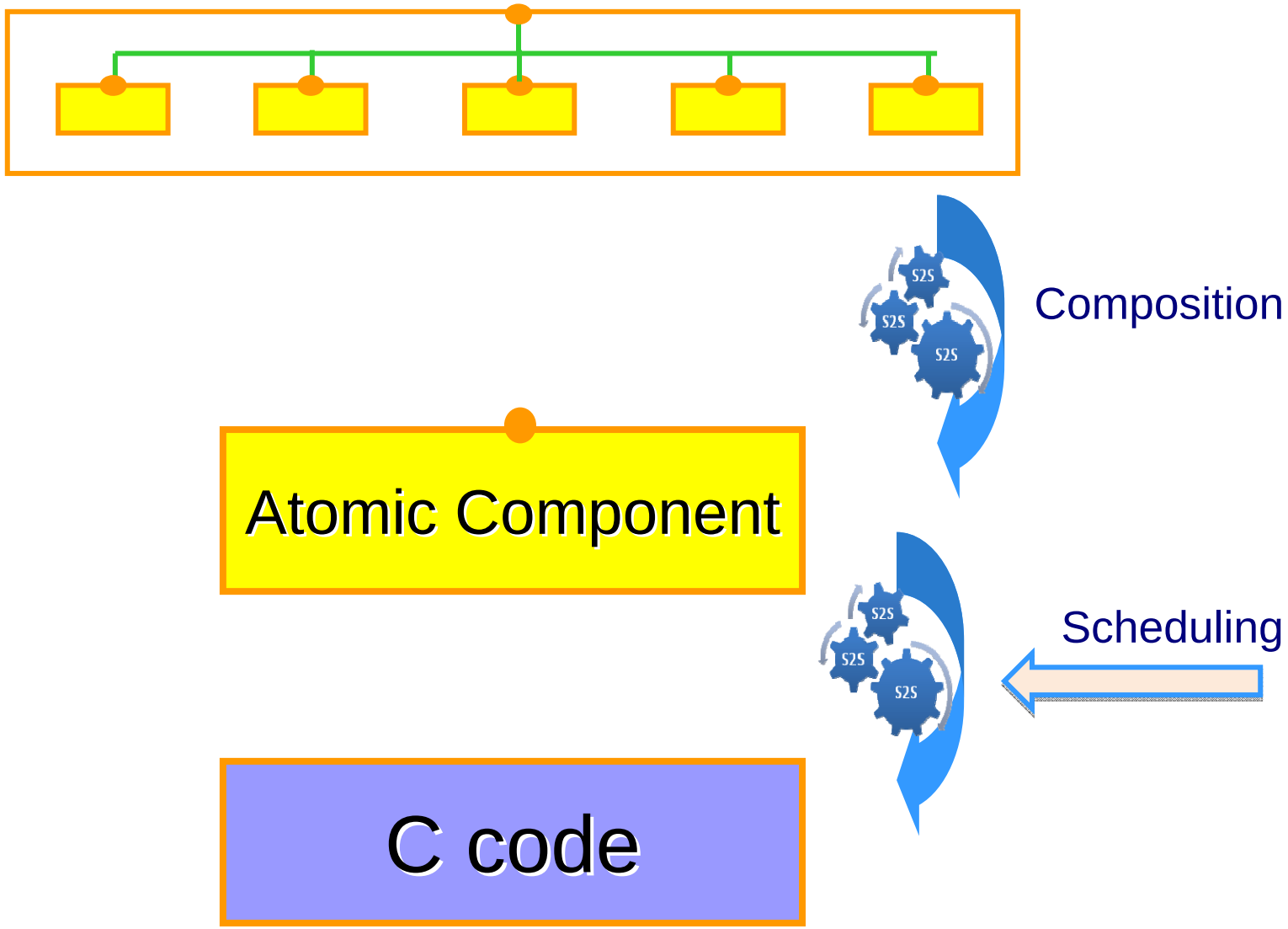




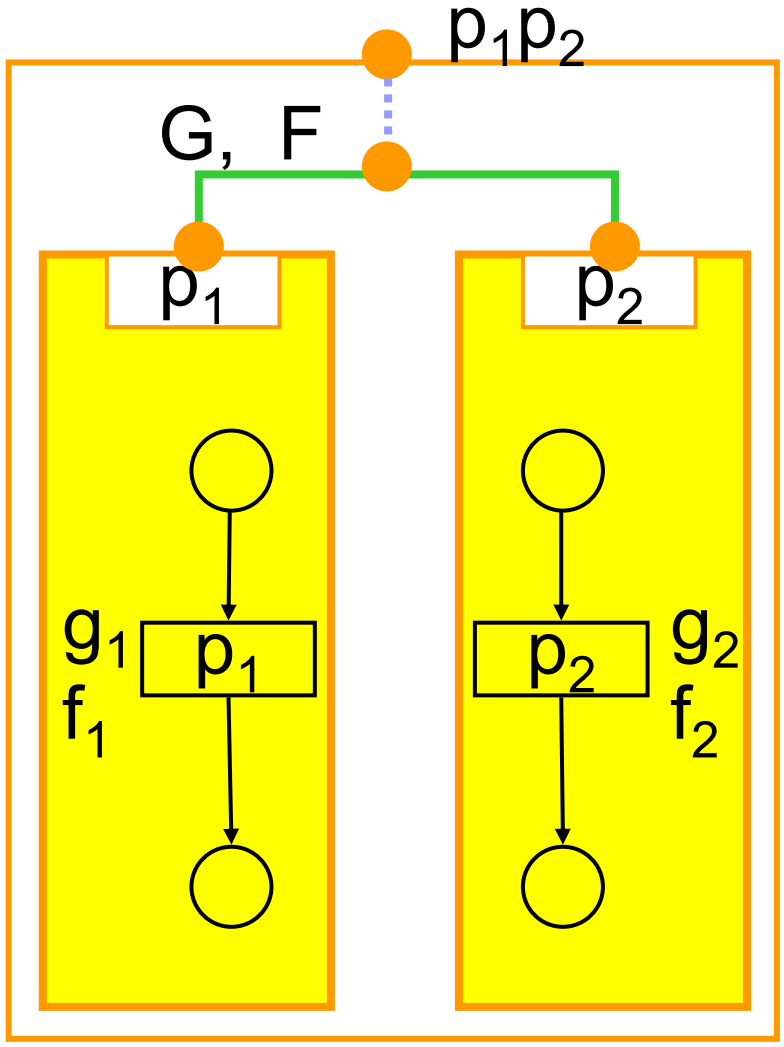
Component Flattening

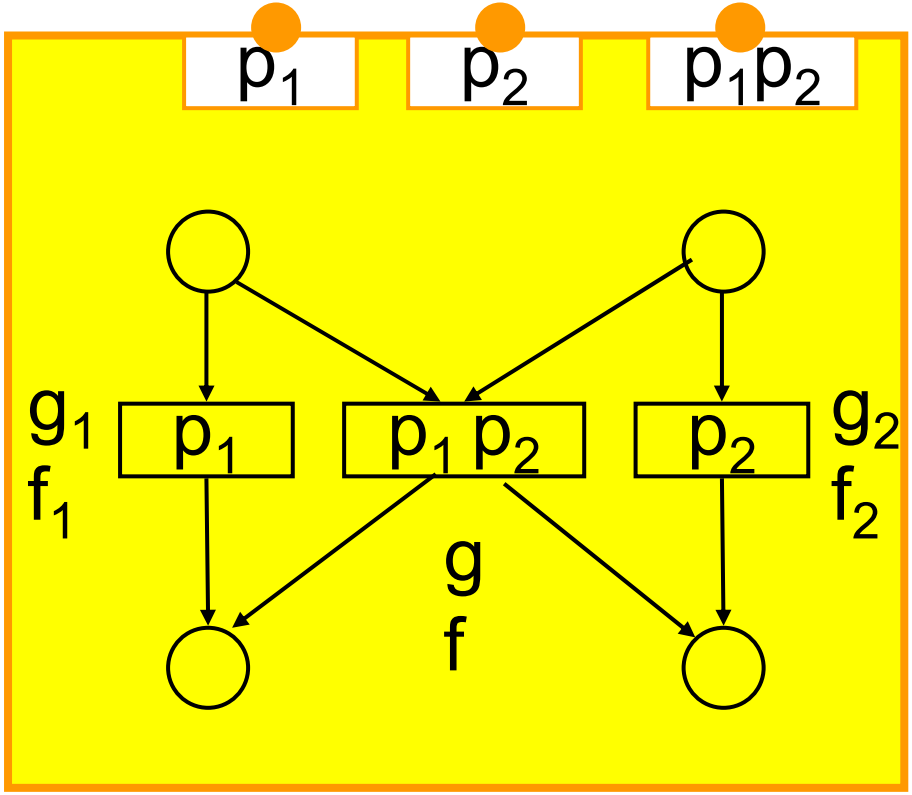
Connector Flattening

From Composite to Monolithic C code



From Composite to Monolithic Components– Composition





Composition

$g = g_1 \wedge g_2 \wedge G$

$f = F;(f_1 || f_2)$

Example – MPEG4 Video Encoder

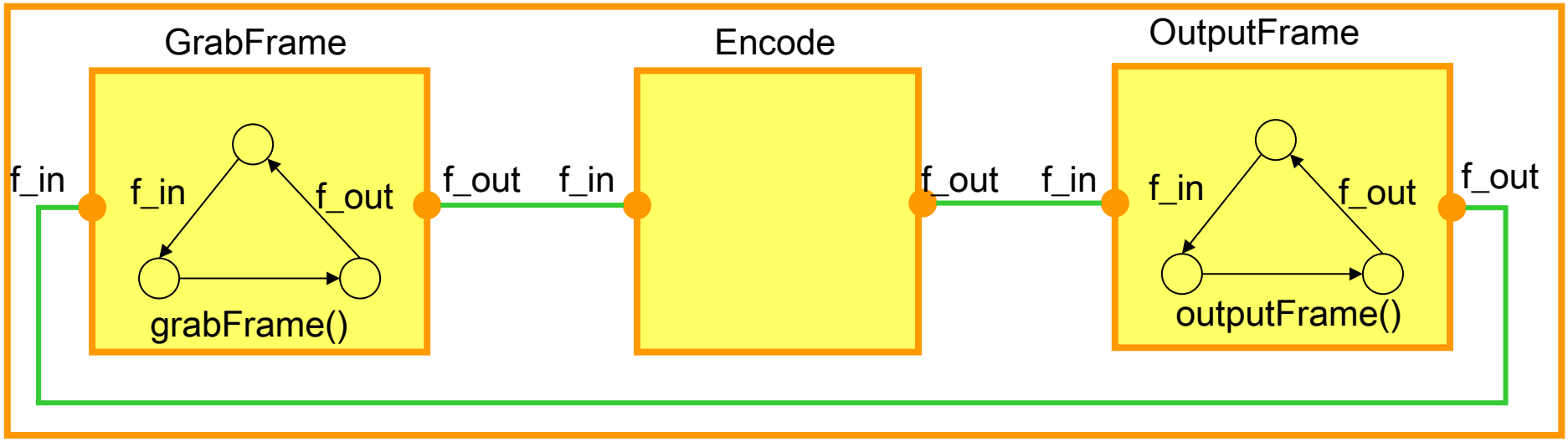
Transform the monolithic sequential program (12000 lines of C code) into a componentized one:

++ reusability, schedulability analysis, reconfigurability

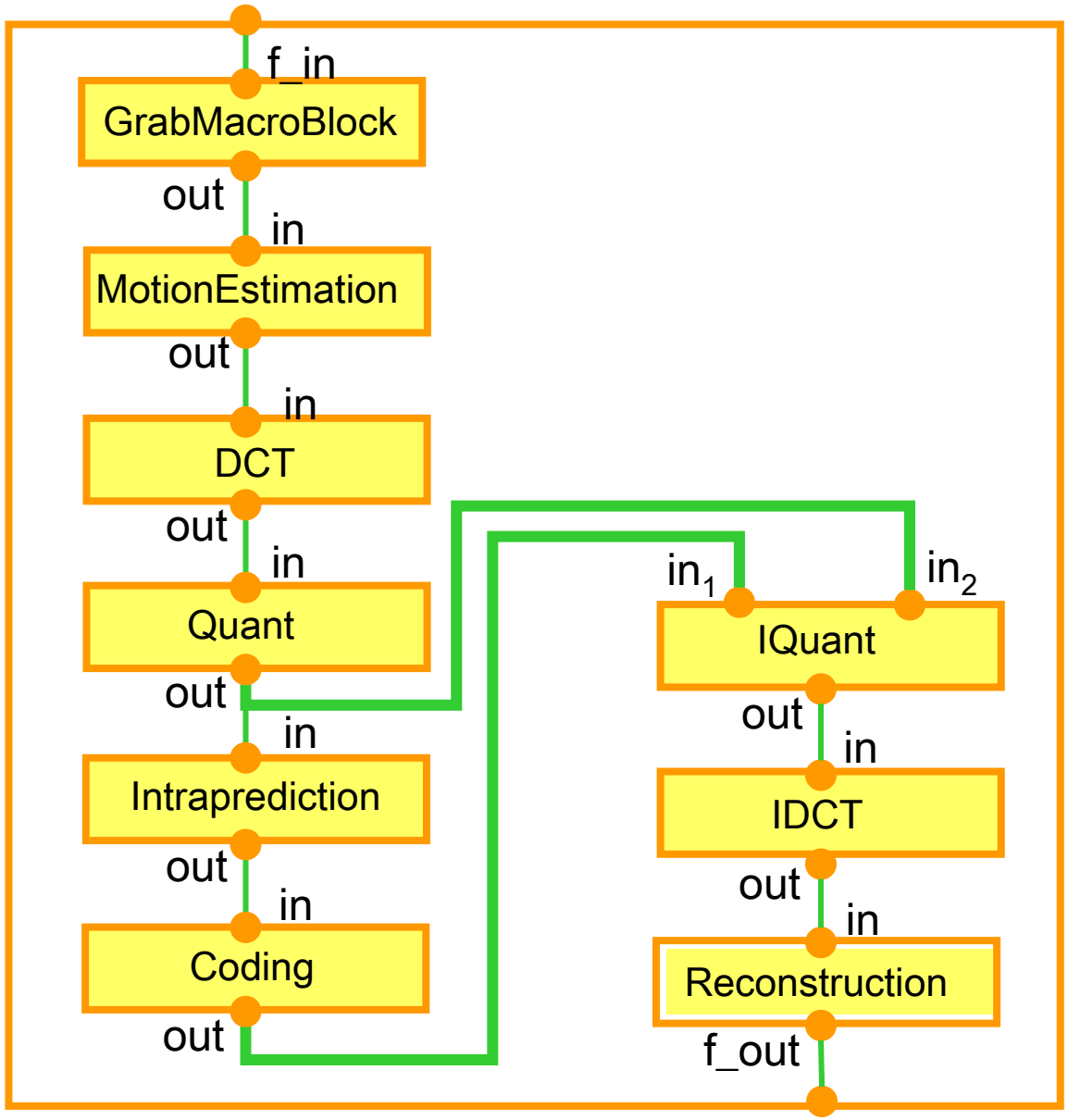
– – overhead in memory and execution time

Decomposition:

- GrabFrame: gets a frame and produces macroblocks
- OutputFrame: produces an encoded frame
- Encode: encodes macroblocks



Example – MPEG4 Video Encoder

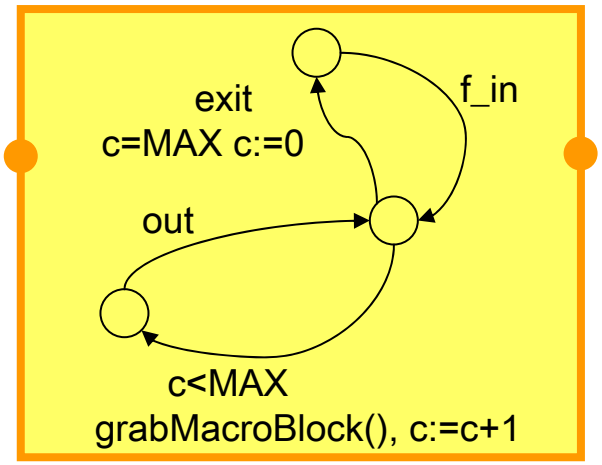


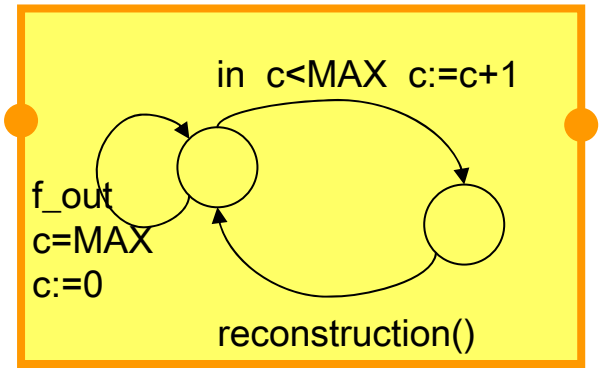
GrabMacroBlock: splits a frame in (W*H)/256 macro blocks, outputs one at a time

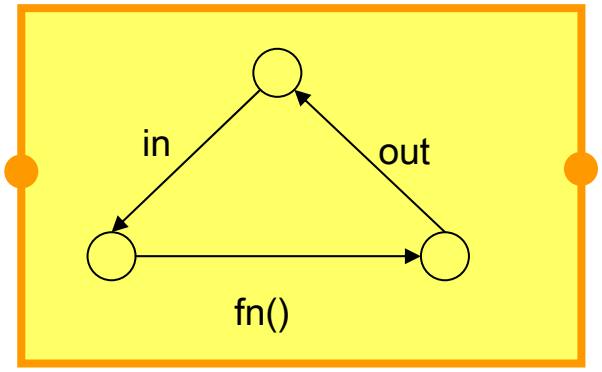
Reconstruction: regenerates the encoded frame from the encoded macro blocks.

: buffered connections

Example – MPEG4 Video Encoder







MAX=(W*H)/256
W=width of frame
H=height of frame

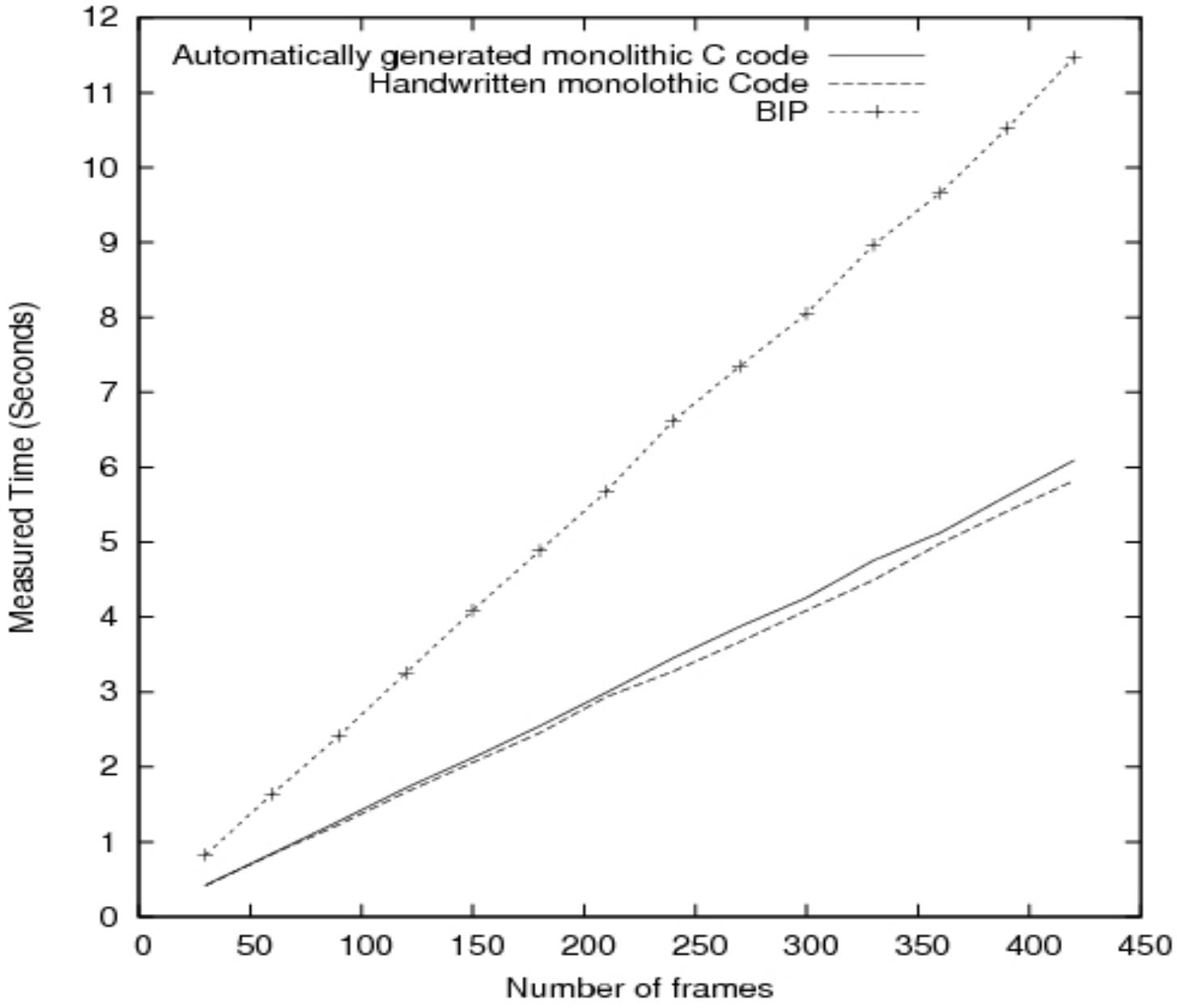
Example – MPEG4 Video Encoder: Results

- ~ 500 lines of BIP code
  - Consists of 20 atomic components and 34 connectors
  - Components call routines from the encoder library
- The generated C++ code from BIP is ~ 2,000 lines
- BIP binary is 288 KB compared to 172 KB of monolithic binary

100% overhead in execution time wrt monolithic code

- ~66% due to computation of interactions (can be reduced by composing components)
- ~34% due to evaluation of priorities (can be reduced by applying priorities to atomic components)

Source-to-Source – MPEG4 Video Encoder: Results

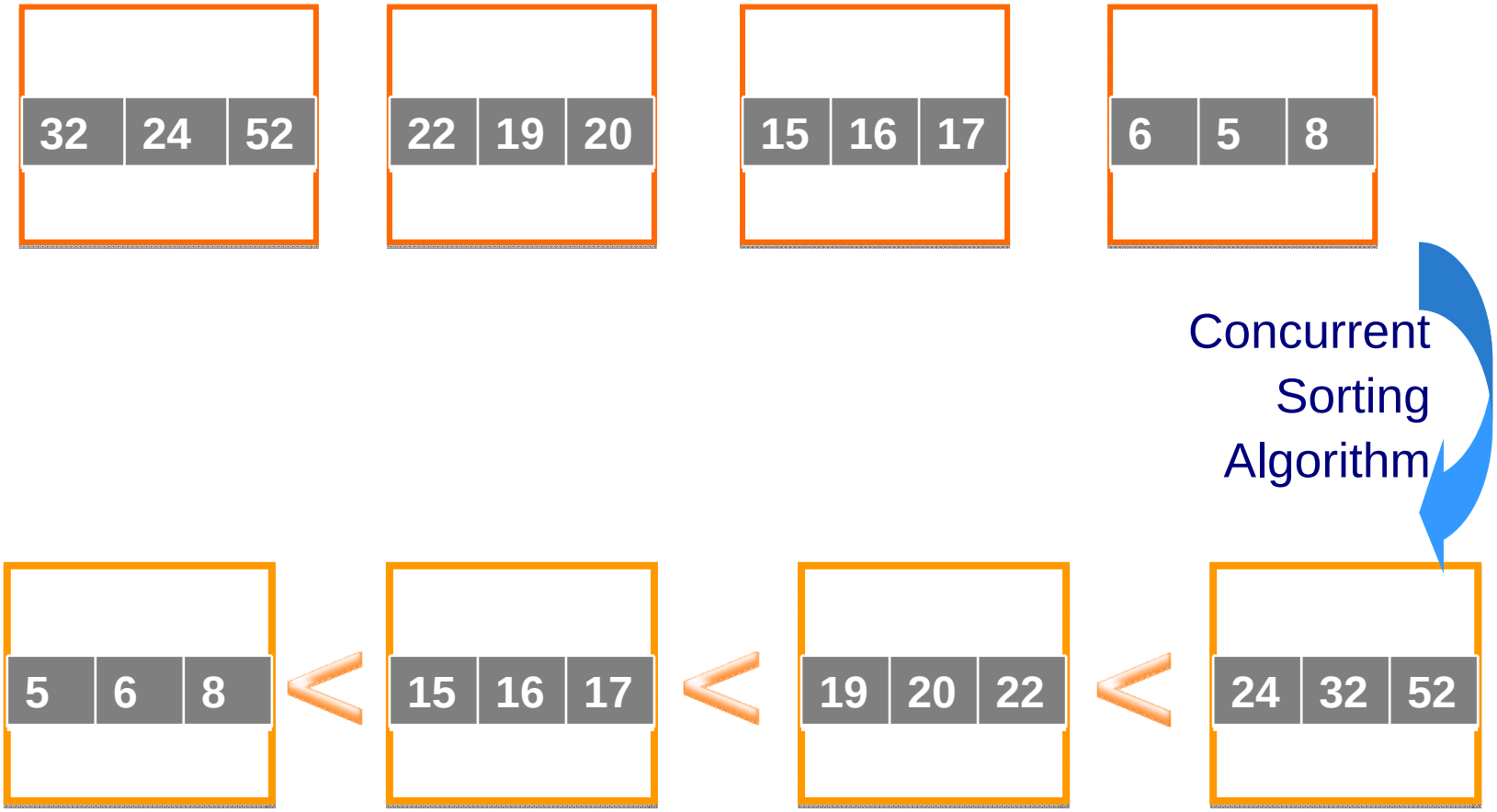


OVERVIEW

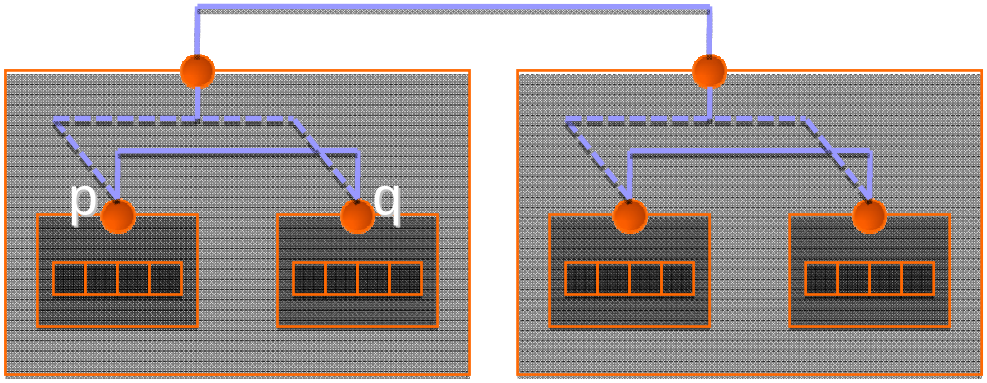
- Distributed Model Generation
- Monolithic code generation
- Examples
- Discussion

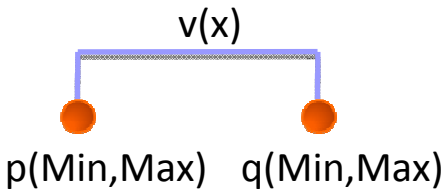
Concurrent Sorting

Inspired from a network sorting algorithm

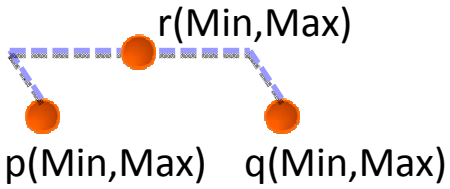


Concurrent Sorting

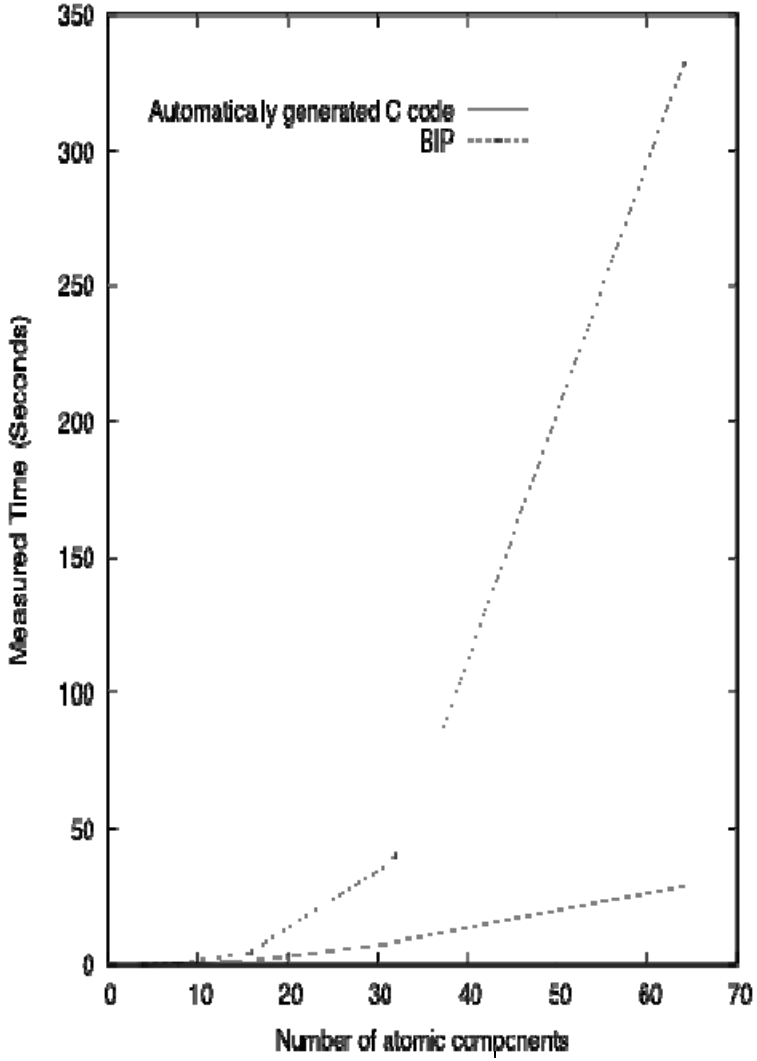




G: p.Max > q.Min
U: none
D: x:= p.Max ; p.Max:=q.Min; q.Min:=x

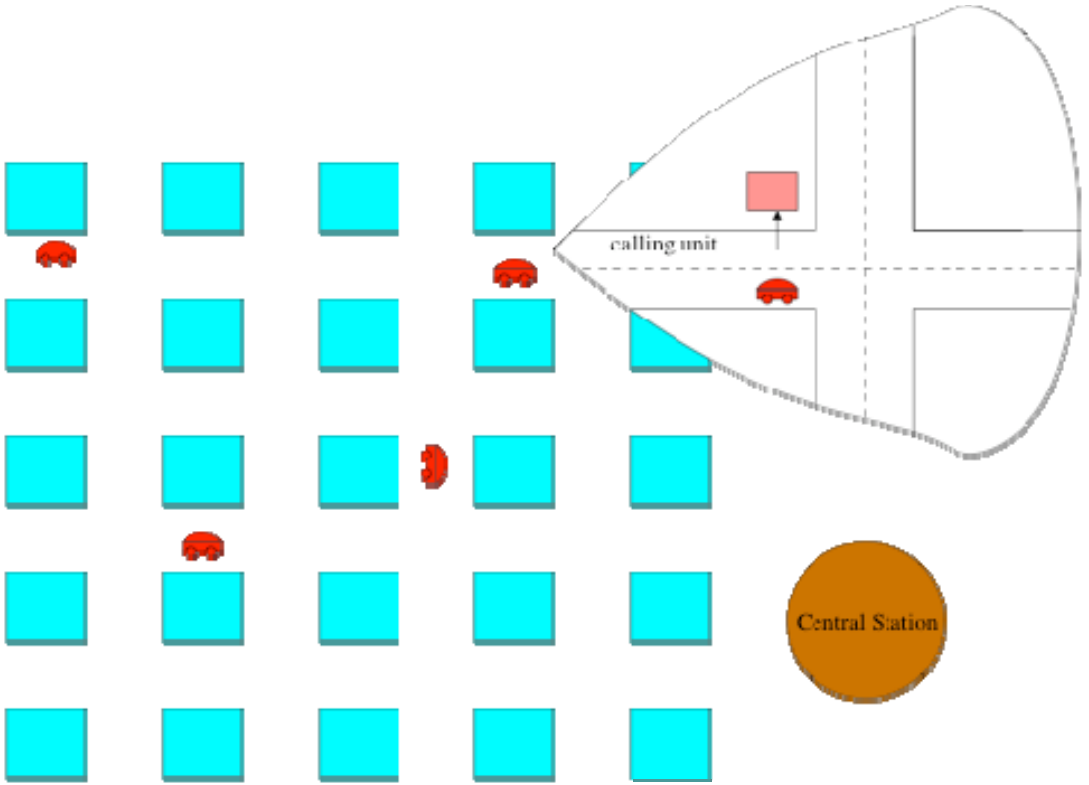


G : p.Max <= q.Min
U : Min := p.Min;Max := q.Max
D : p.Min := Min; q.Max := Max

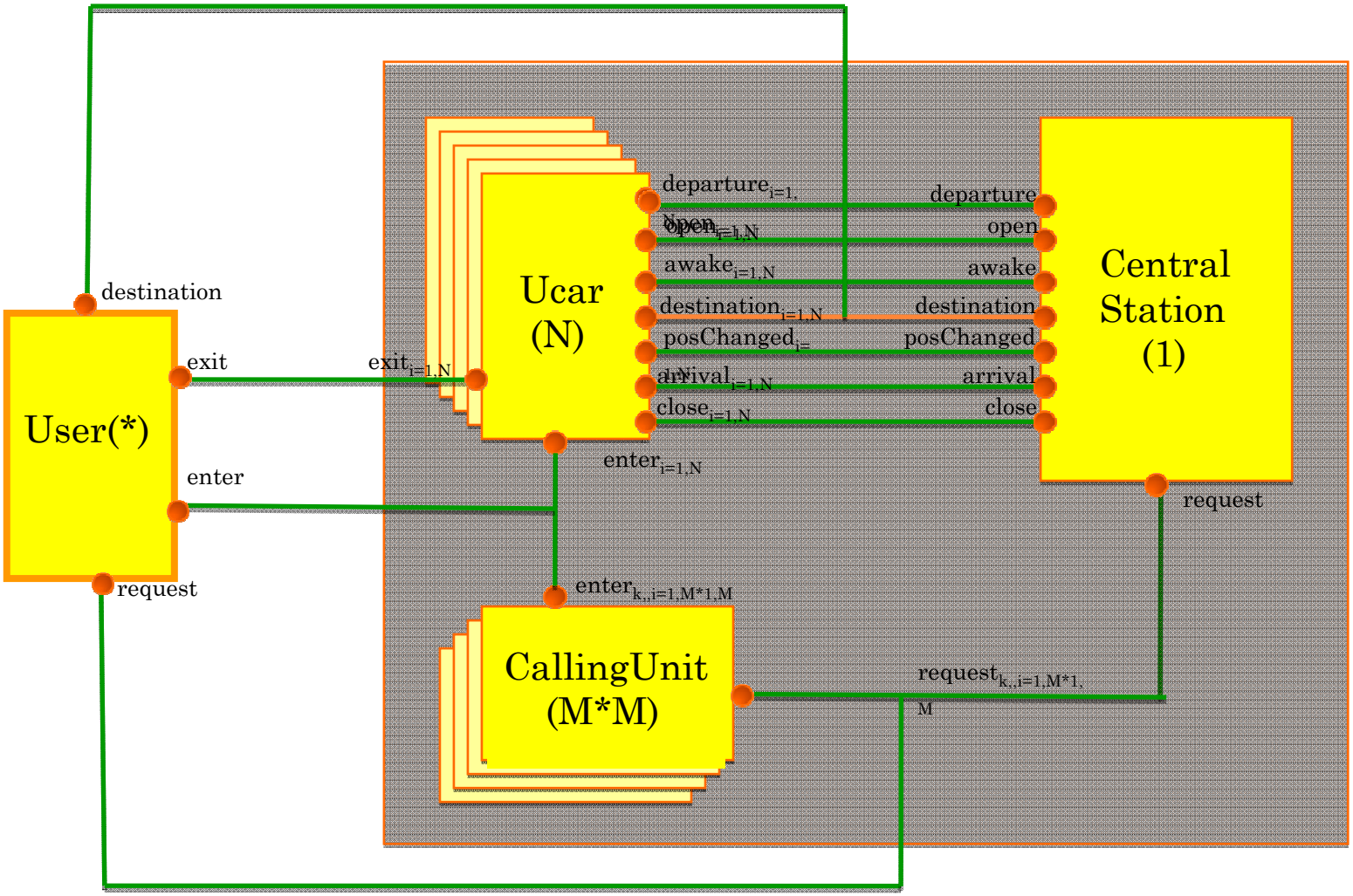


UTOPAR

- UTOPAR is an automated transportation system managing various requests for transportation
- Industrial case study of the Combest Project



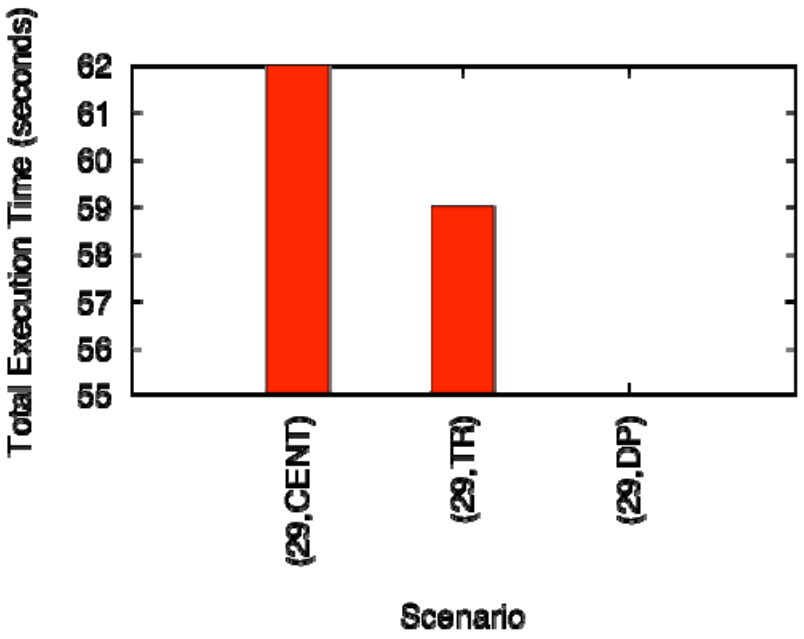
UTOPAR in BIP



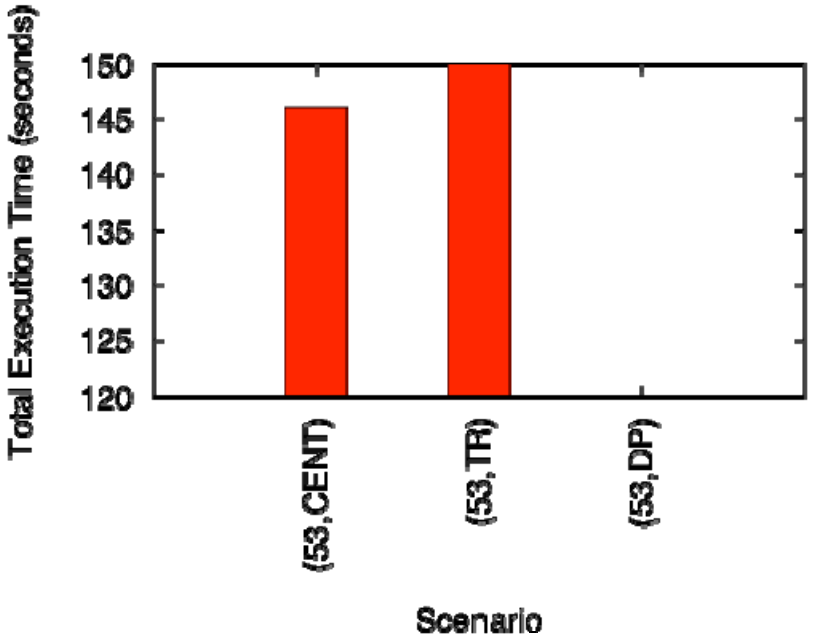
UTOPAR in BIP

- Fully automated generation of distributed C++ code, using sockets

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



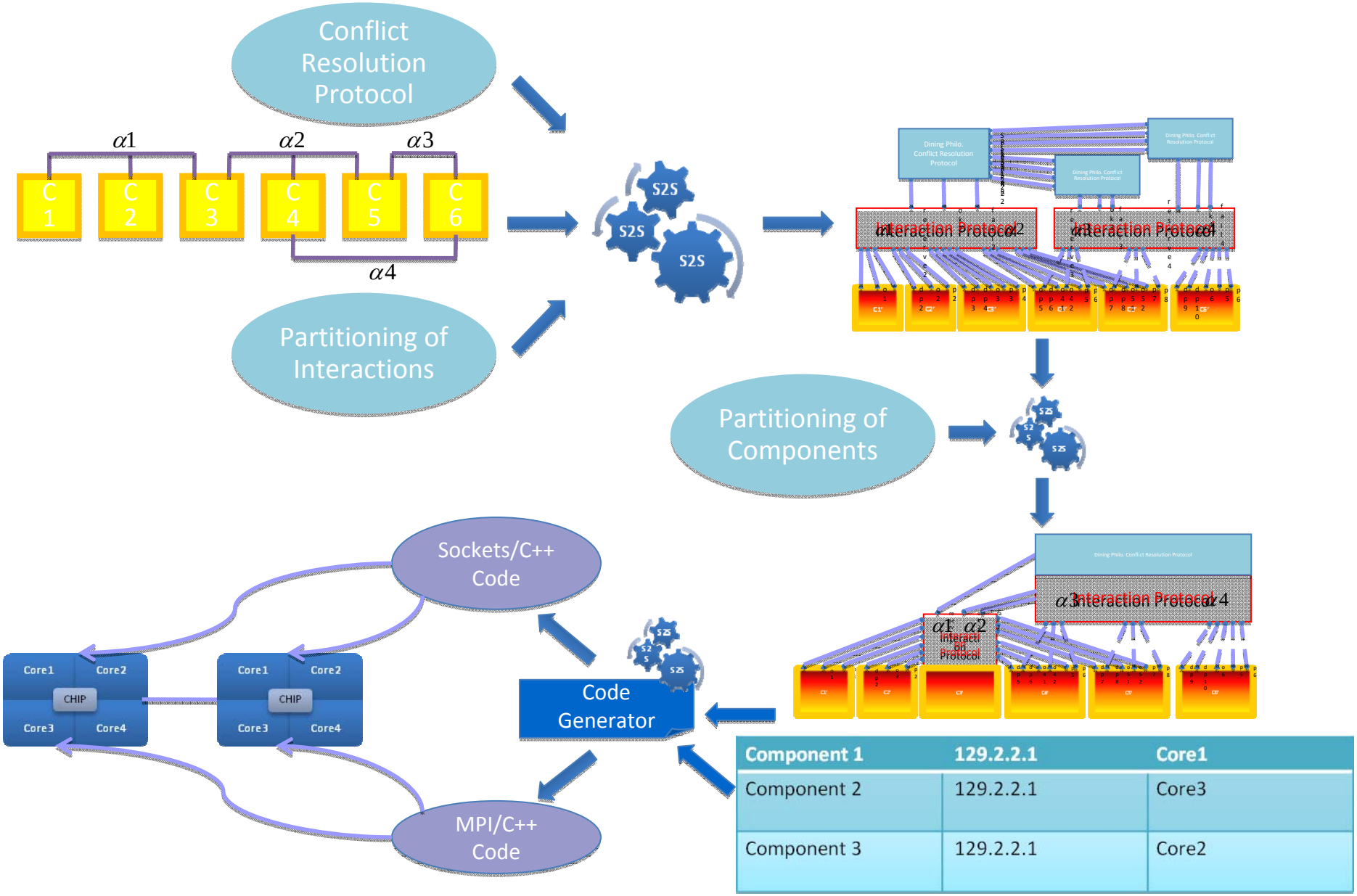
49 = 7 * 7 calling units and 4 cars –
53 Interaction Protocols



OVERVIEW

- Distributed Model Generation
- Monolithic code generation
- Examples
- Discussion

Discussion - Detailed Design Flow

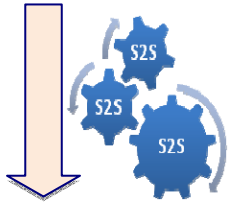


Component 1	129.2.2.1	Core1
Component 2	129.2.2.1	Core3
Component 3	129.2.2.1	Core2

Discussion

Single Engine: Execution of Interactions + Conflict Resolution

Interaction Partitioning

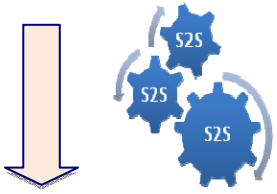


Multiple Engines: Interaction Protocol + Conflict Resolution Protocol

Tradeoff: Degree of parallelism vs. Overhead for coordination depending on

- Interaction partitioning
- Conflict resolution protocol (Centralized, Token Ring, Dining Philosophers etc.)

Component Partitioning



Implementation: maximal parallelism, load balancing, resource management

Discussion – Ongoing Work

- Distributed model with priorities – an additional layer is needed
- Application for automated implementation on many-core platforms
- Global analysis techniques jointly taking into account
  - interaction partitioning
  - component partitioning
  - HW resources
- Performance evaluation and design space exploration tools
- Case studies

Thank You