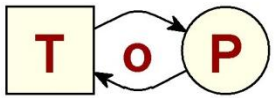
SUMMERSOC

Hersonissos, Wednesday, June 27, 2018. 9.30 – 10.30

Conceptual Fundamentals of Reactive Systems

Same place as Christoph Freytag



Theory of Programming

Prof. Dr. W. Reisig

Wolfgang Reisig

Humboldt-Universität zu Berlin

Conceptual Fundamentals of Reactive Systems

What could this be?

Identify the (??) fundamental concepts

and build a theory on top of this …

We are so wonderfully progressing without

… for a while

There is this deep "Theoretical Informatics" stuff. That's enough.

No. There is something fundamentally new

What to do with such a conceptual fundament?

to make it conceptually simpler,

e.g. PAXOS

better teachable,

better usable, also by non-experts

Road map

1. Components and composition:
   the basic paradigm for communication
2. Fundamental properties of the composition operator
3. Components' contents
4. … even more general

Road map

1. **Components and composition:**
   the basic paradigm for communication
2. Fundamental properties of the composition operator
3. Components' contents
4. … even more general

An analogy: Classical computing

Turing machine:

A sequence of containers

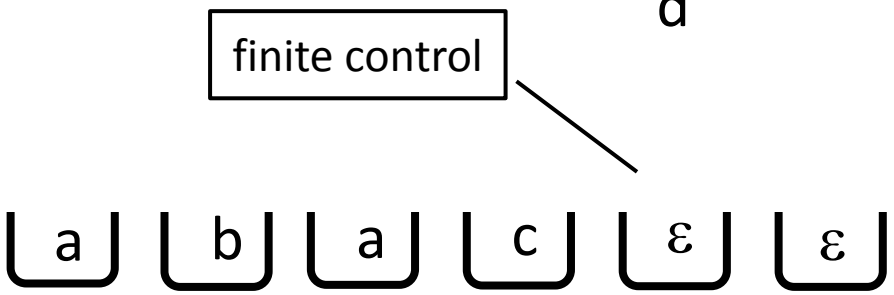
Each container

holds a symbol

d

actual container

can be updated



The starting point for
computable functions

The starting point for react. systems …?

Reactive systems are ***fundamentally*** different.

Don’t compute functions at all

Are not intended to terminate

Can not be abstracted to *one* device

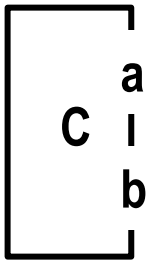
…

… requires a fundamentally new computation model

A *system* consists of *components*.

Each component has an *interface*.

An interface contains *gates*.



Components are *composed*
by gluing *matching gates*

Literature describes many such system models

A system consists of *components*.

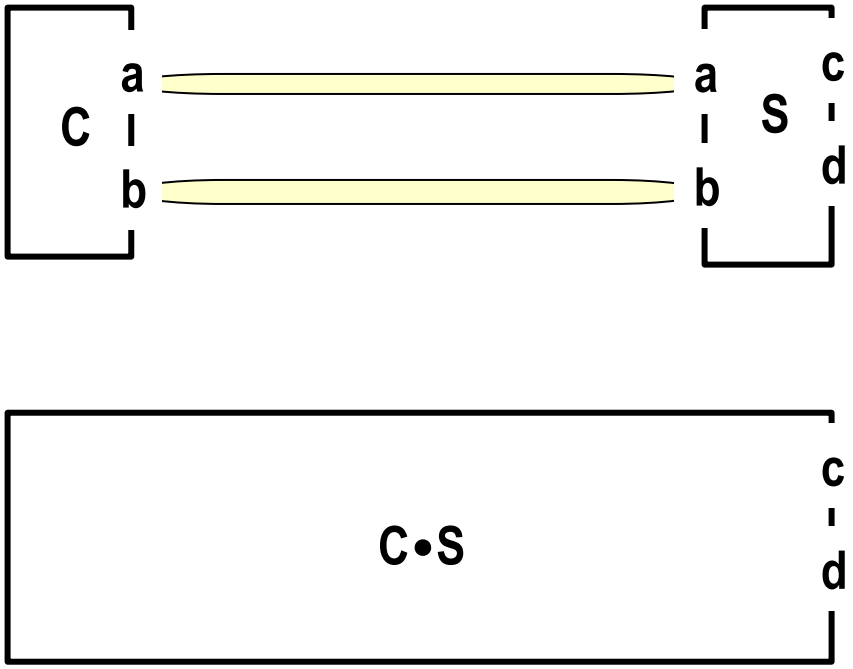
Each component has an *interface*.

An interface contains *gates*.

Components are *composed* by gluing *matching gates*

Each gate has a *label*.

Equally labeled gates *match.*



A small, but decisive variant

A system consists of *components*.

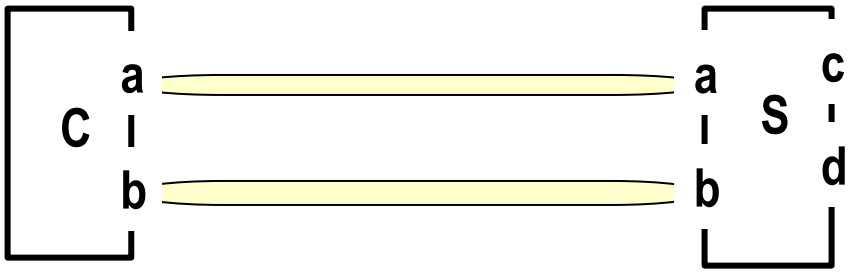
Each component ~~has an *interface*~~.
A has *two* interfaces, ***A** and **A***.

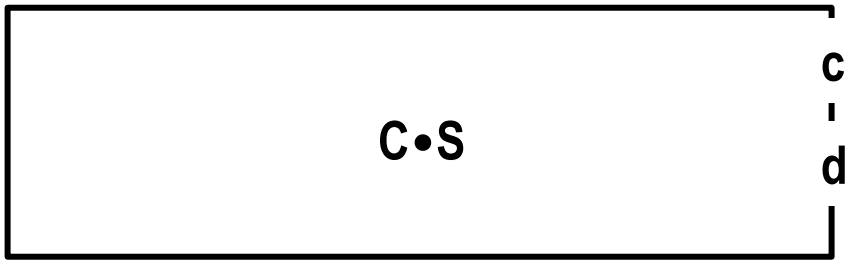
An interface contains *gates*.

Components **A** and **B** are *composed*
by gluing *matching gates*
of **A*** *and* ***B**.

Each gate has a *label*.

Equally labeled gates
of **A*** *and* ***B** *match.*





A small, but decisive variant

A system consists of *components*.

Each component ~~has an *interface*~~.

A has *two* interfaces, ***A** and **A***.

An interface contains *gates*.

Components **A** and **B** are *composed*

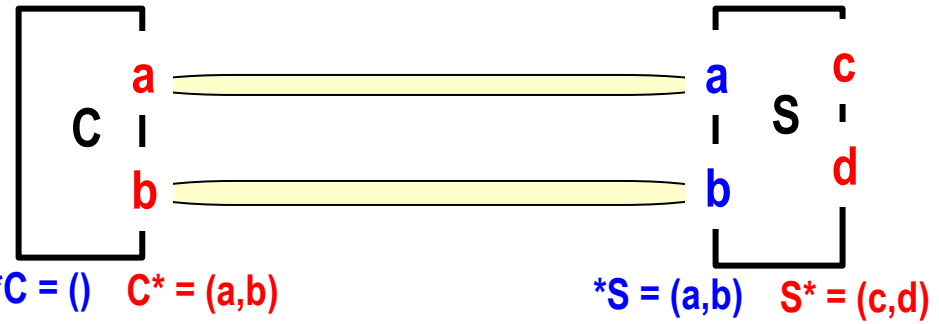
by gluing *matching gates*

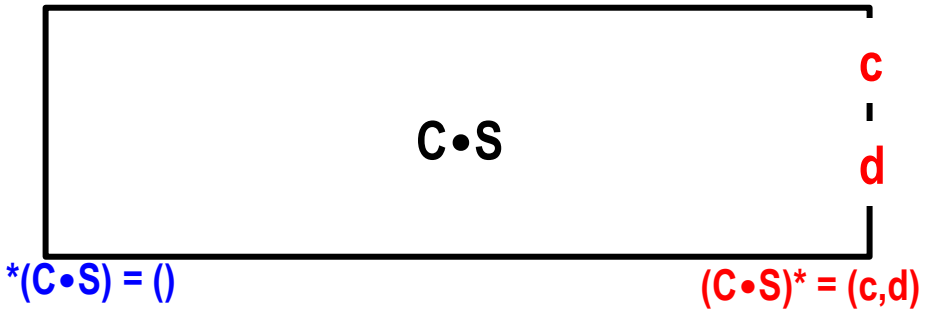
of ***A*** *and* ***B**.*

Each gate has a *label*.

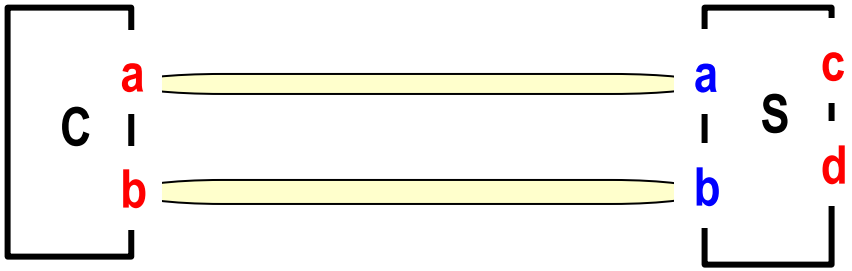
Equally labeled gates

of ***A*** *and* ***B*** *match.*

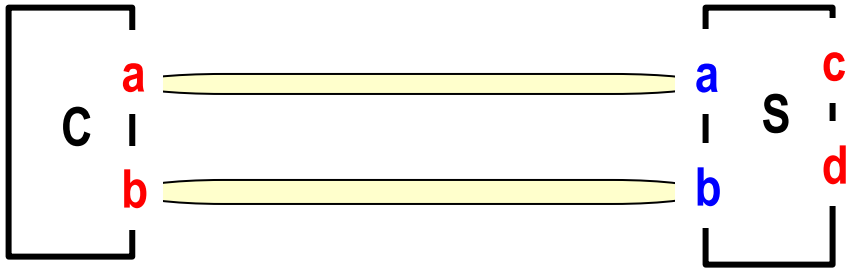


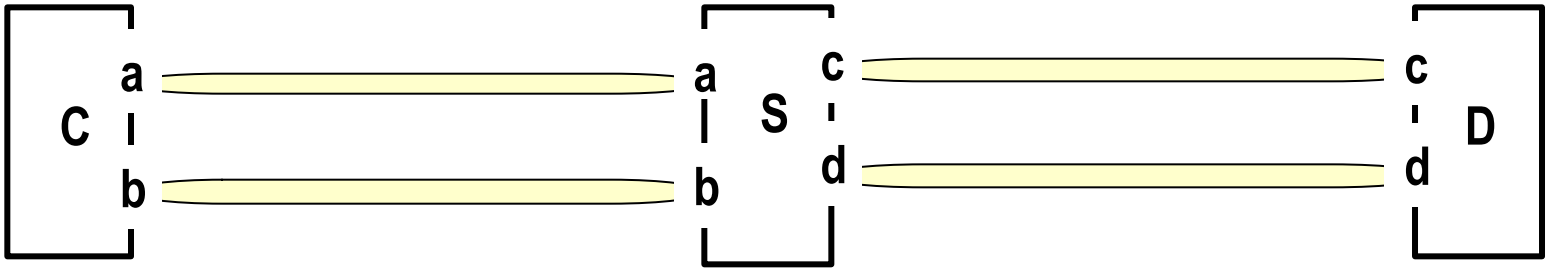


A small exercise: operating system architecture



A small exercise: operating system architecture



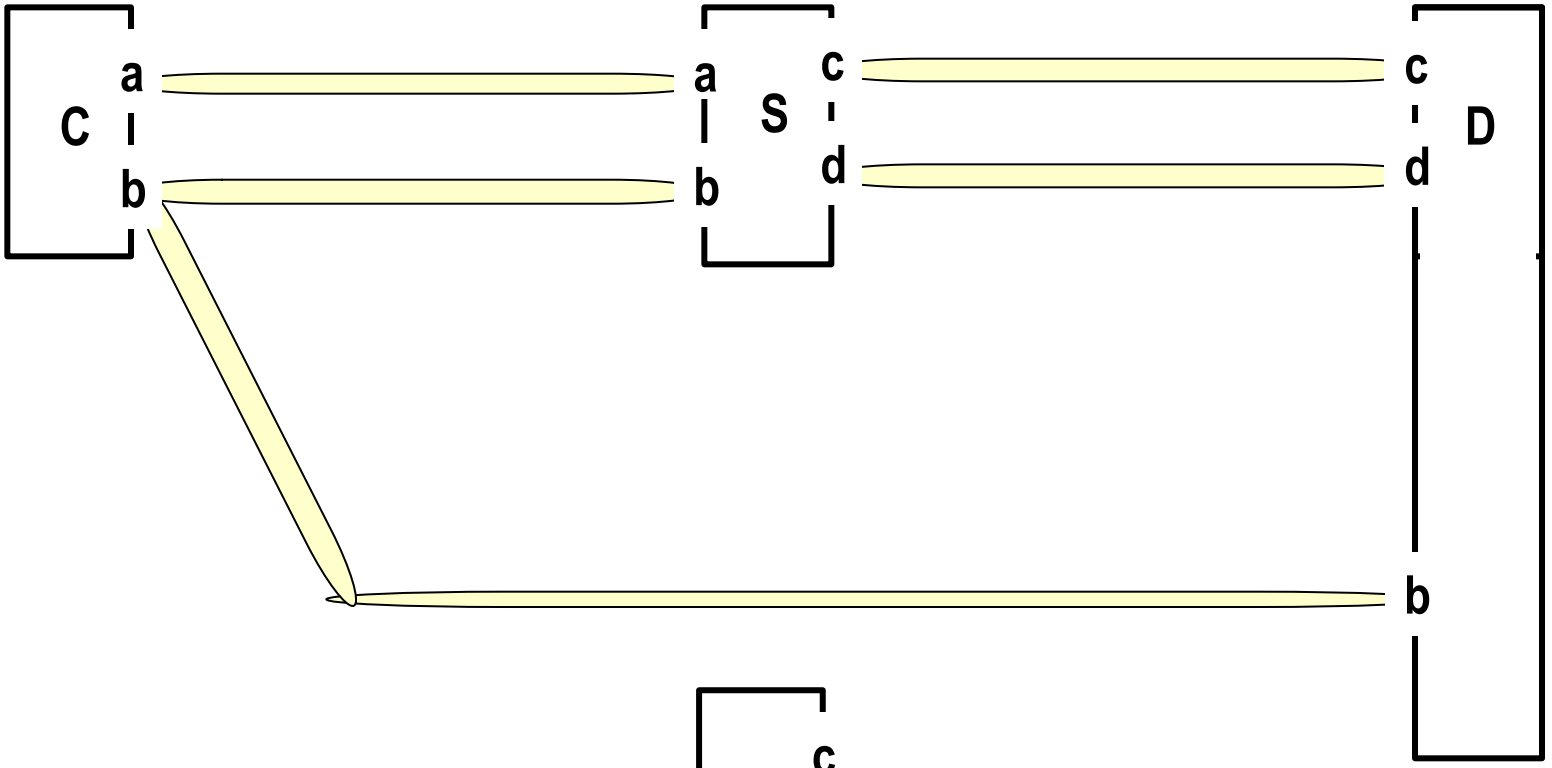


C client
S service
D data base

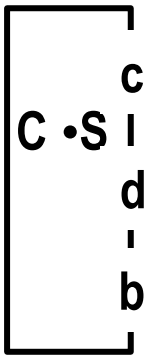
C • S • D

a messages from client
b messages to client

c message from service
d message to service

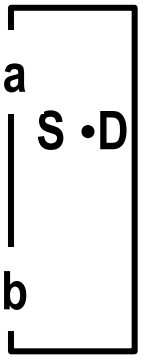


C client
S service
D data base



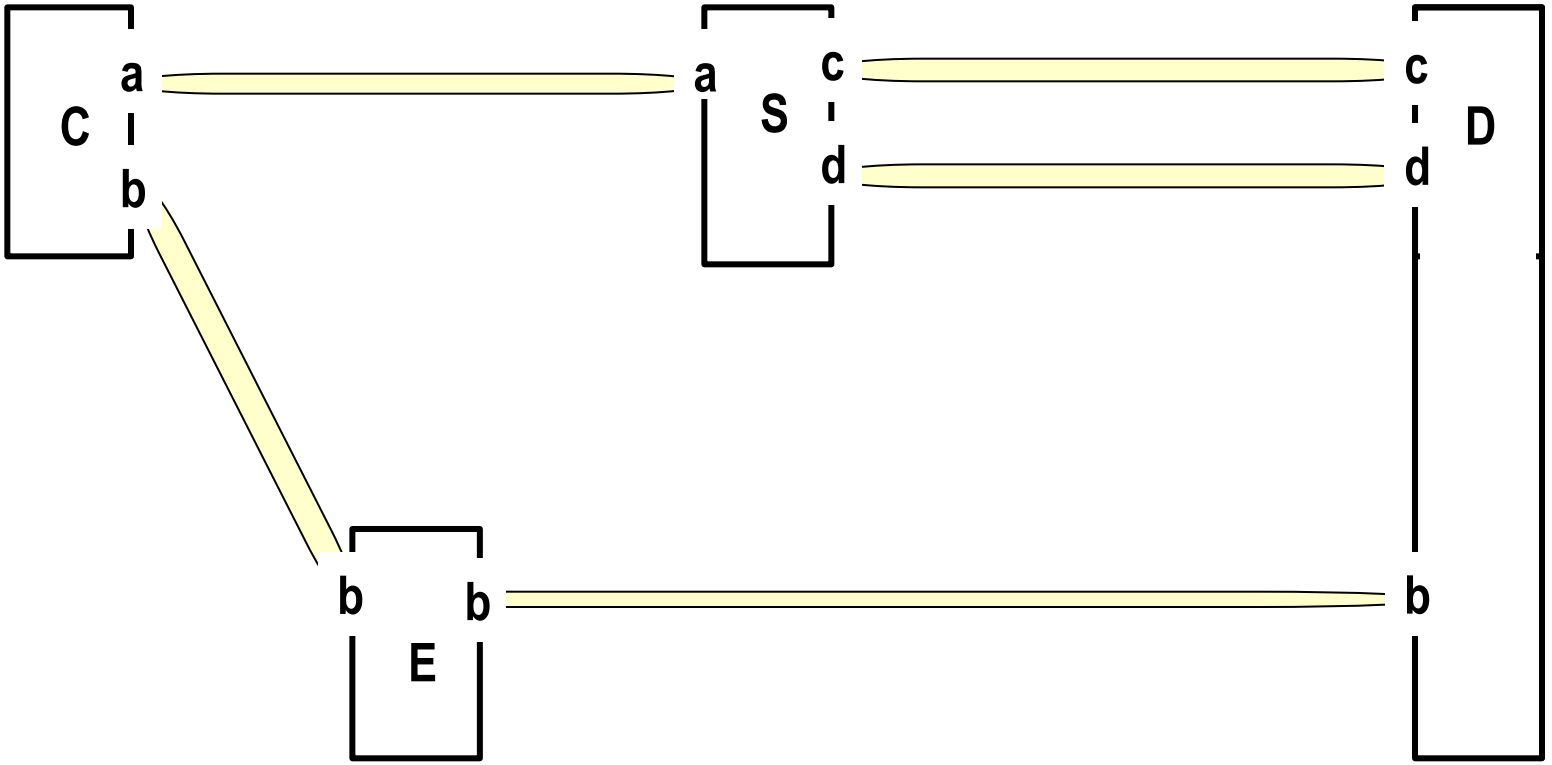
C • S • D

alternative:



a messages from client
b messages to client

c message from service
d message to service



C • E • S • D

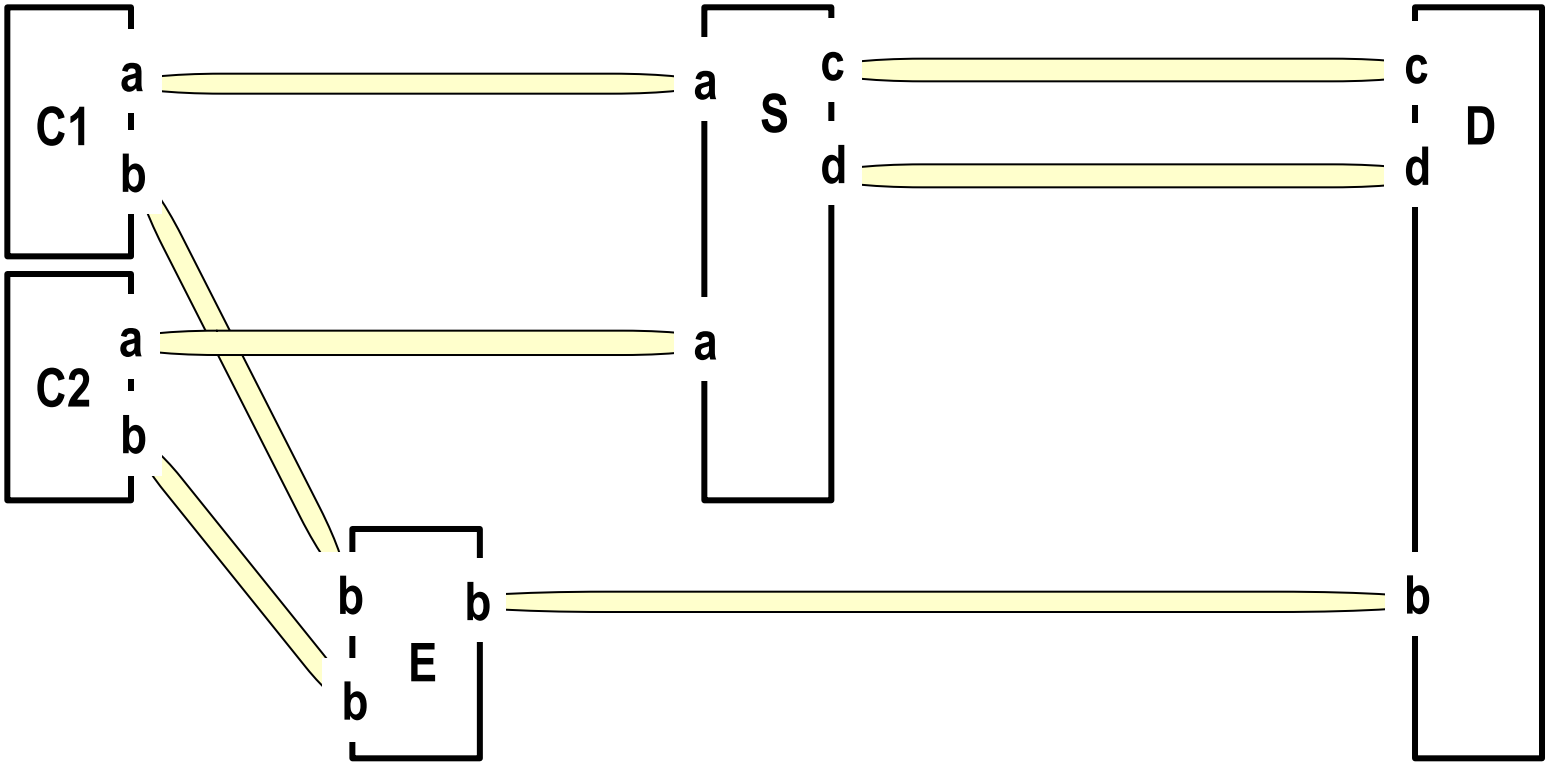
C • S • E • D

a messages from client

b messages to client

c message from service

d message to service



C1 • C2 • E • S • D

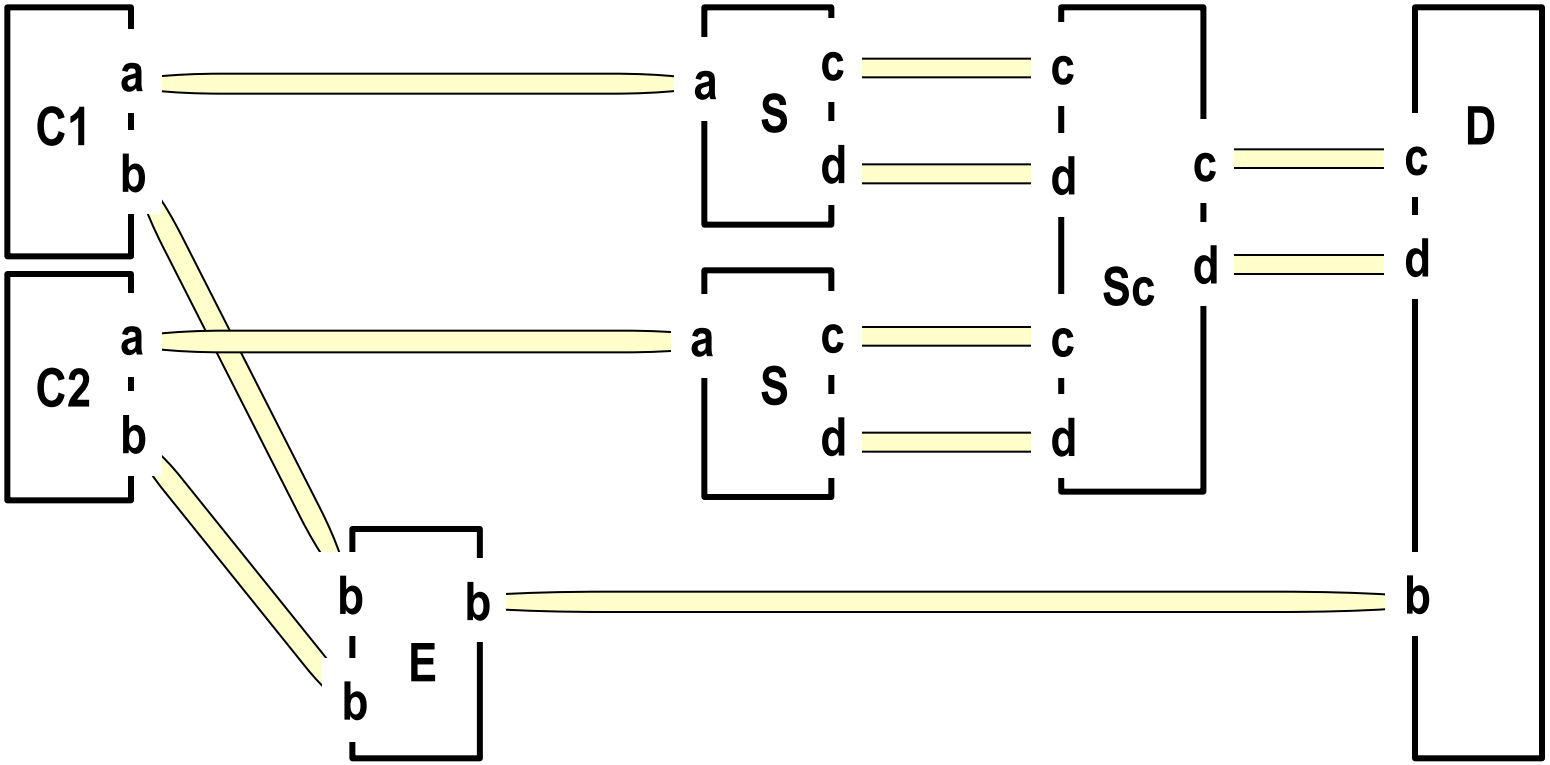
C1 • C2 • S • E • D

a messages from client

b messages to client

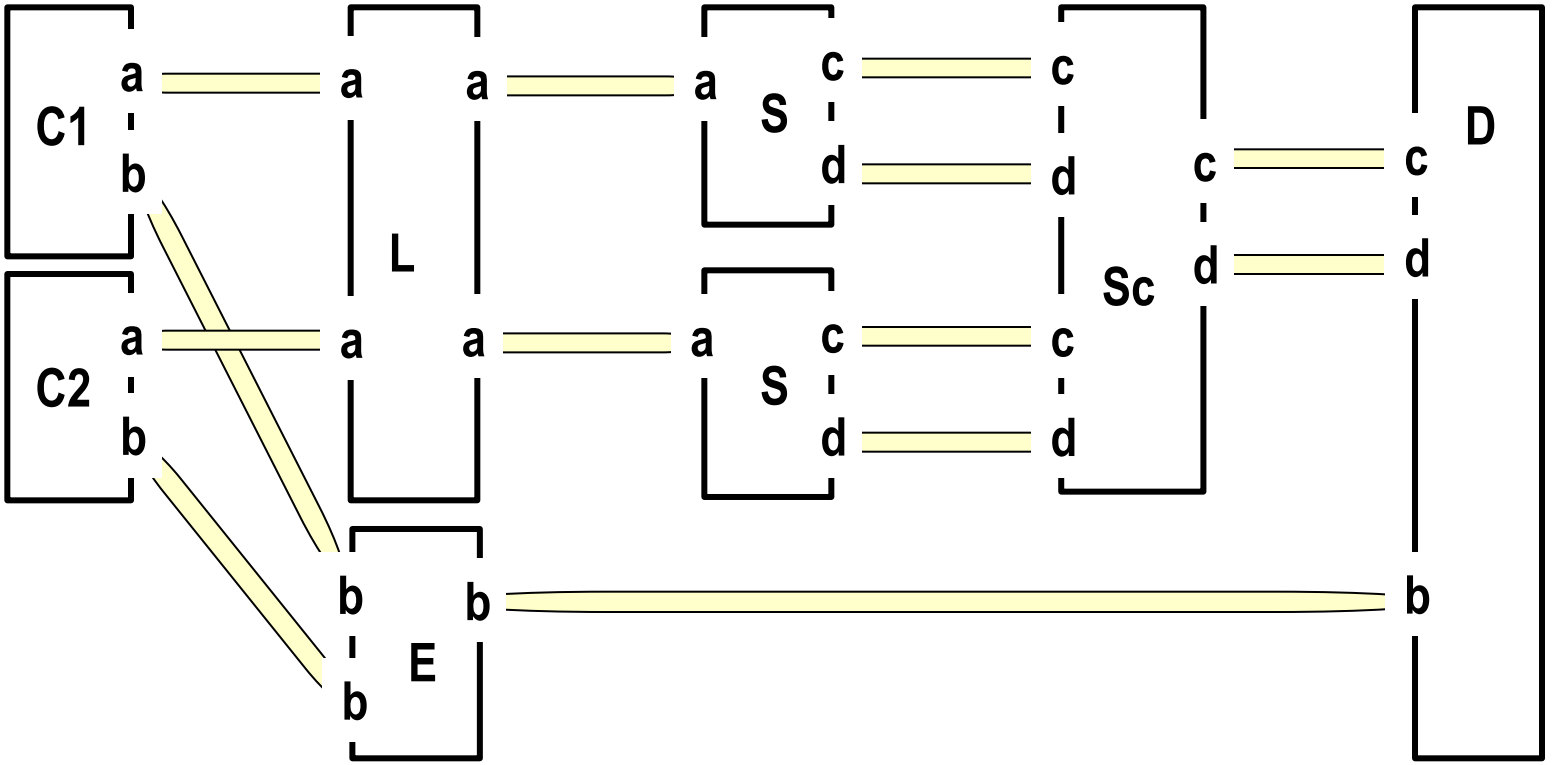
c message from service

d message to service



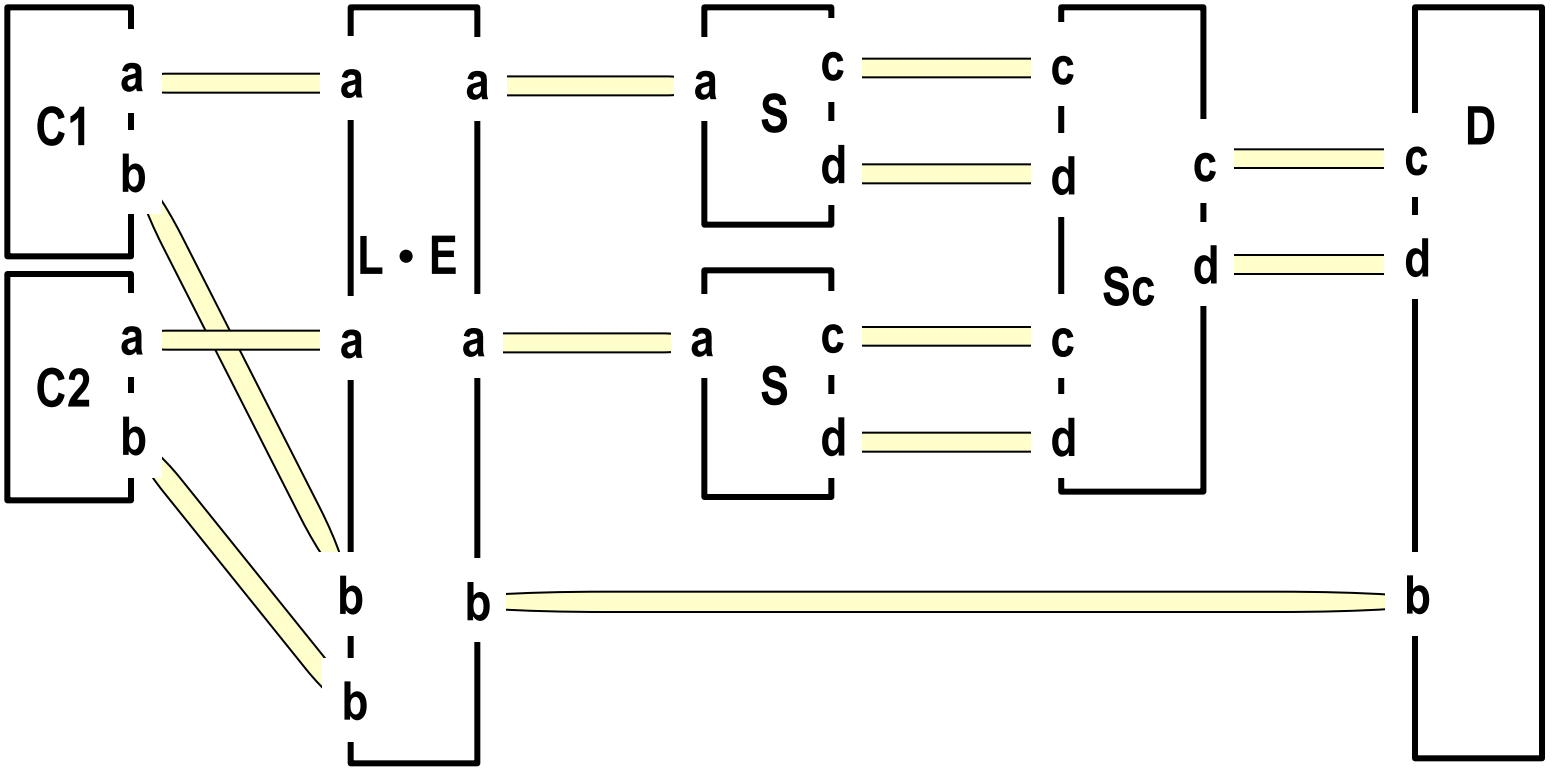
C1 • C2 • E • S • S • Sc • D

a messages from client
b messages to client
c message from service
d message to service



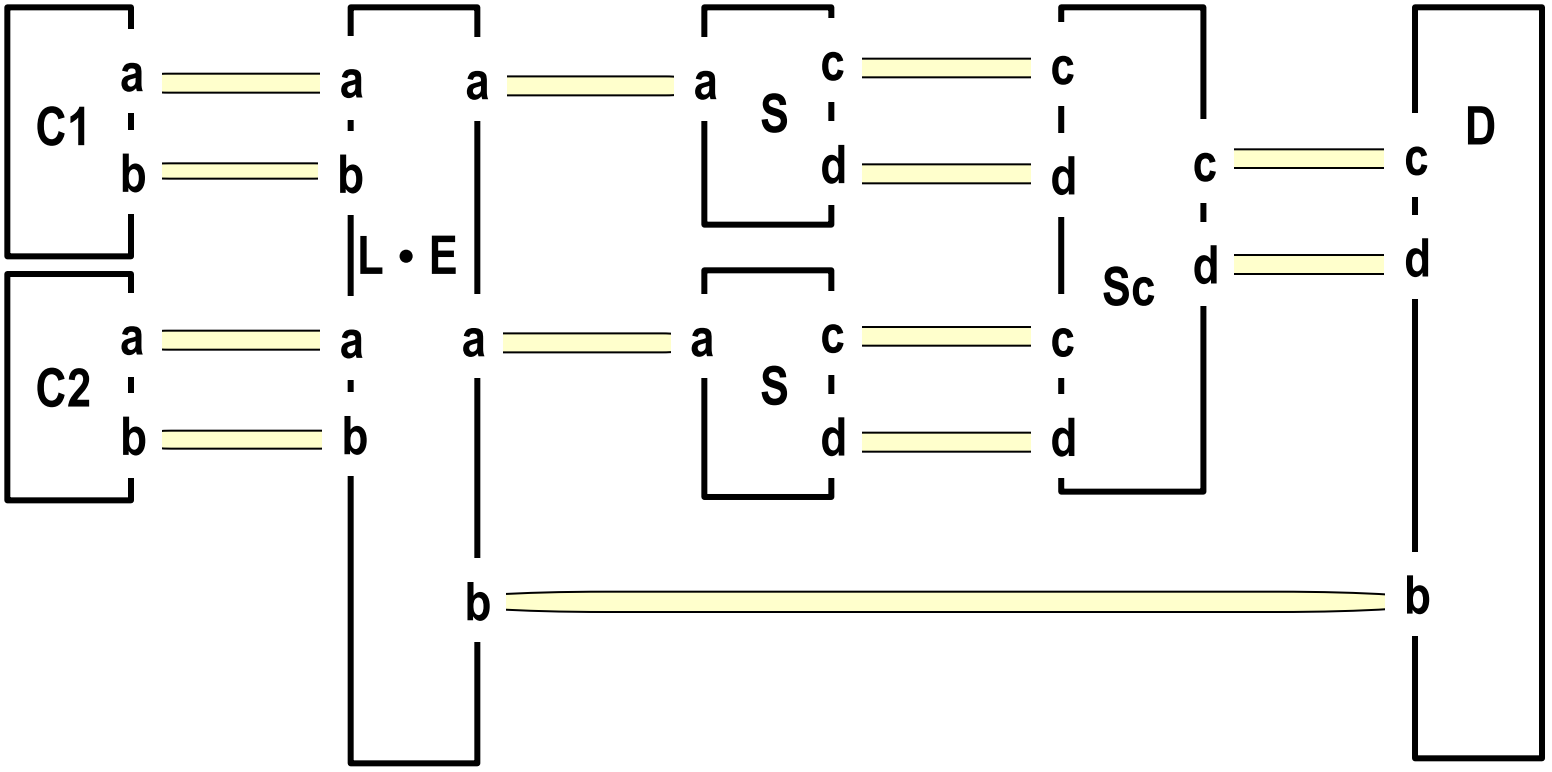
C1 • C2 • L • E • S • S • Sc • D

a messages from client
b messages to client
c message from service
d message to service



C1 • C2 • (L • E) • S • S • Sc • D

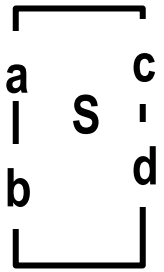
a messages from client
b messages to client
c message from service
d message to service

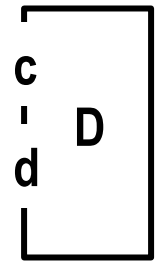


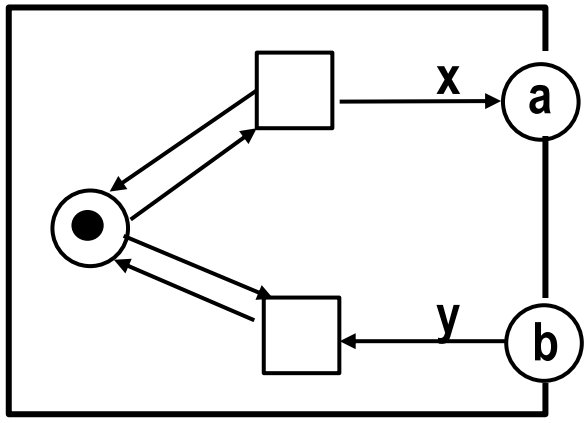
C1 • C2 • (L • E) • S • S • Sc • D

a messages from client
b messages to client
c message from service
d message to service

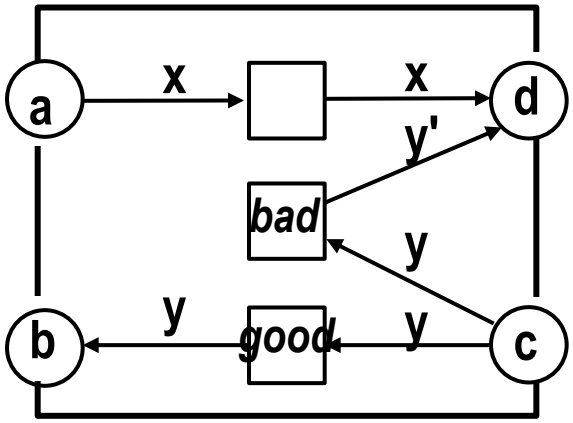
A look inside the components



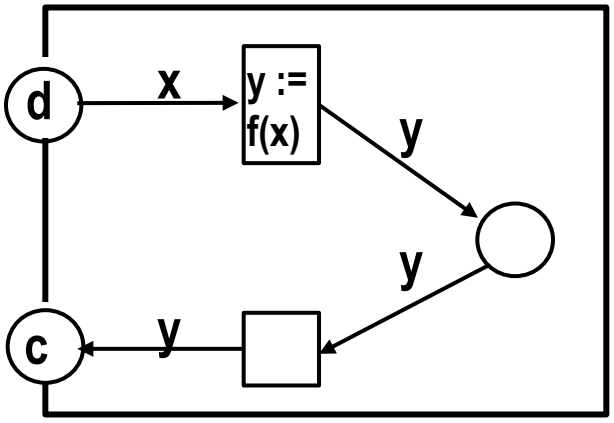




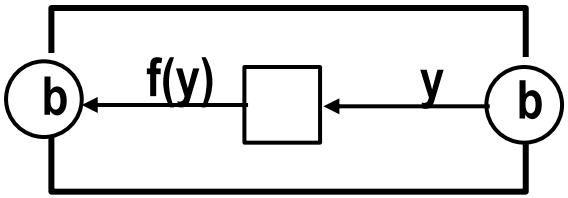
behavior of client **C**



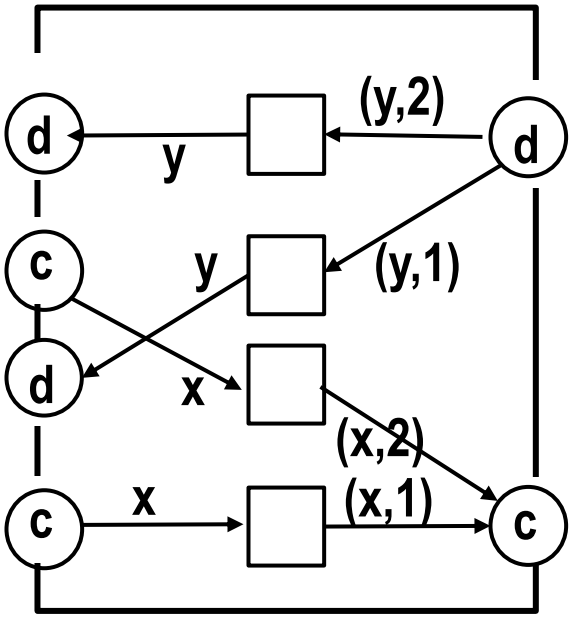
behavior of service **S**



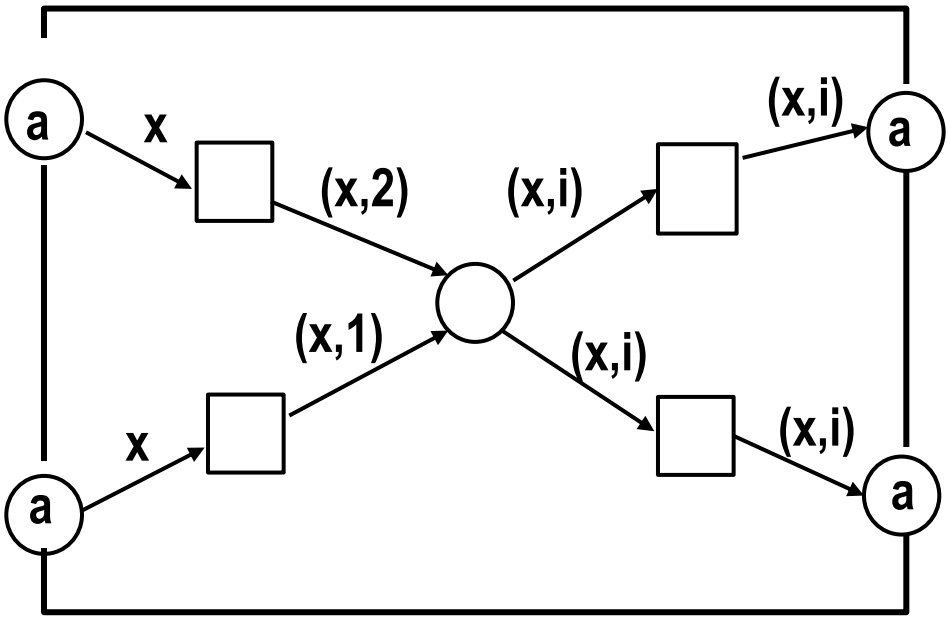
behavior of Data base **D**



behavior editor **E**

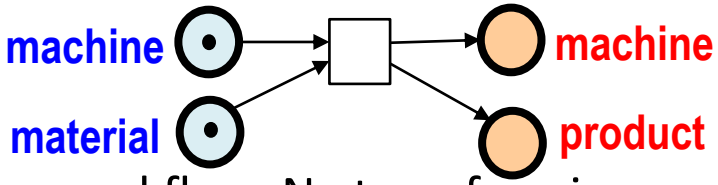


behavior of
Scheduler **Sc**



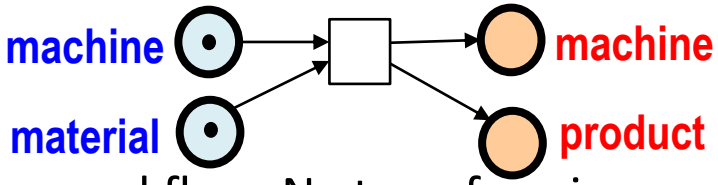
behavior of
Load balancer **L**

Multiple labels

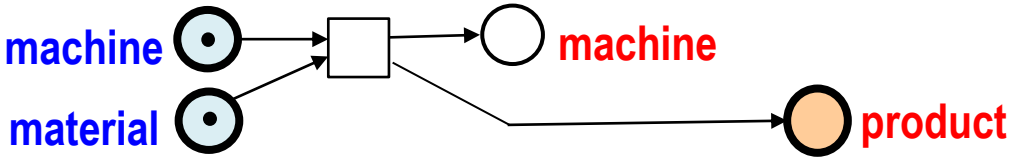


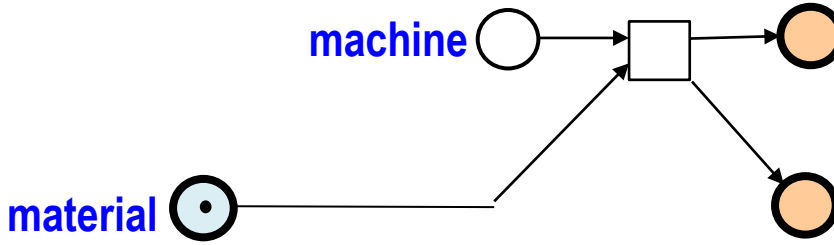
a. workflow N, transforming material into products by help of a machine

Multiple labels

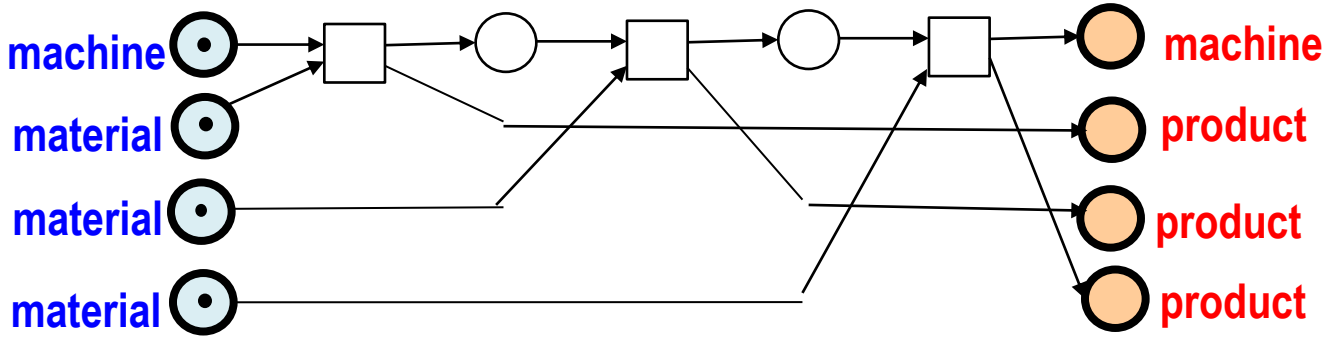


a. workflow N, transforming material into products by help of a machine





b. composed workflow, N•N



c. composed workflow, N•N•N

Summing up

Double sided components are intuitively most natural

customer	and	*supplier*
provider	and	*requester*
producer	and	*consumer*
buy side	and	*sell side*
input	and	*output*
requred	and	*offered*

Road map

1. Components and composition:
   the basic paradigm for communication
2. **Fundamental properties of the composition operator**
3. Components' contents
4. … even more general

Composition is about *architecture*

- strict and formal for the interface
- entirely liberal for the components‘ contents

Assume a fixed interface alphabet Λ .

Let $\mathcal{C}$ denote the set of components with

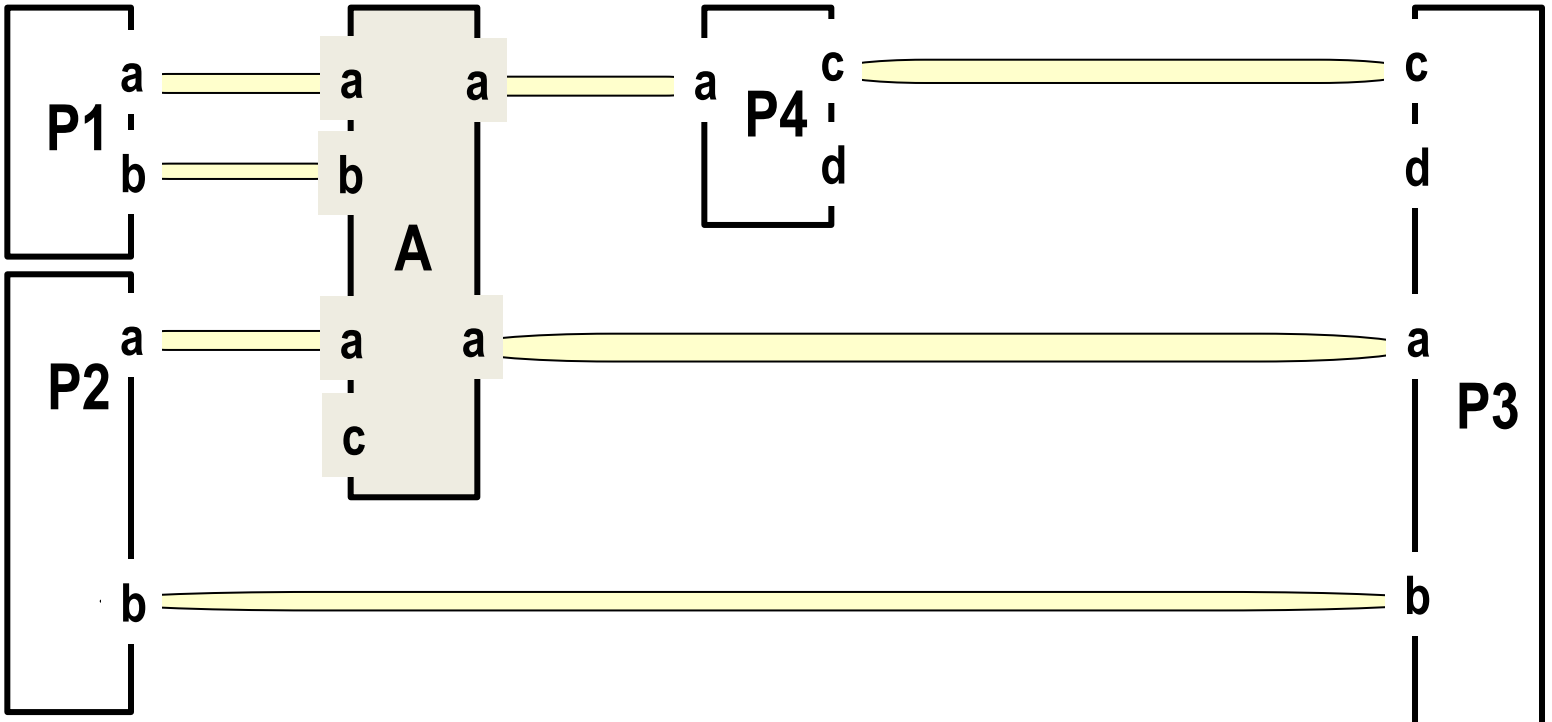
- gate labels from Λ and
- empty contents

Components‘ contents: later

Composition on $\mathcal{C}$ is *universal*:

Given *any* finite component network:
Can it be composed from its components?

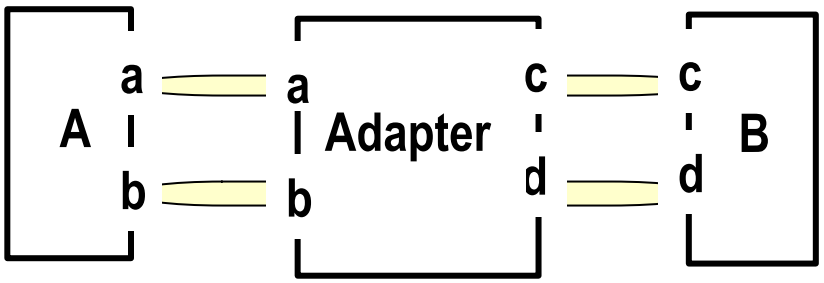
Yes!



P1 • P2 • A • P3 • P4

Composition on **C** is *total*

Any two components in **C** can be composed.



Want to glue **a** with **c** and **b** with **d**.
New kind of composition?
Composability predicate for **A** and **B**?

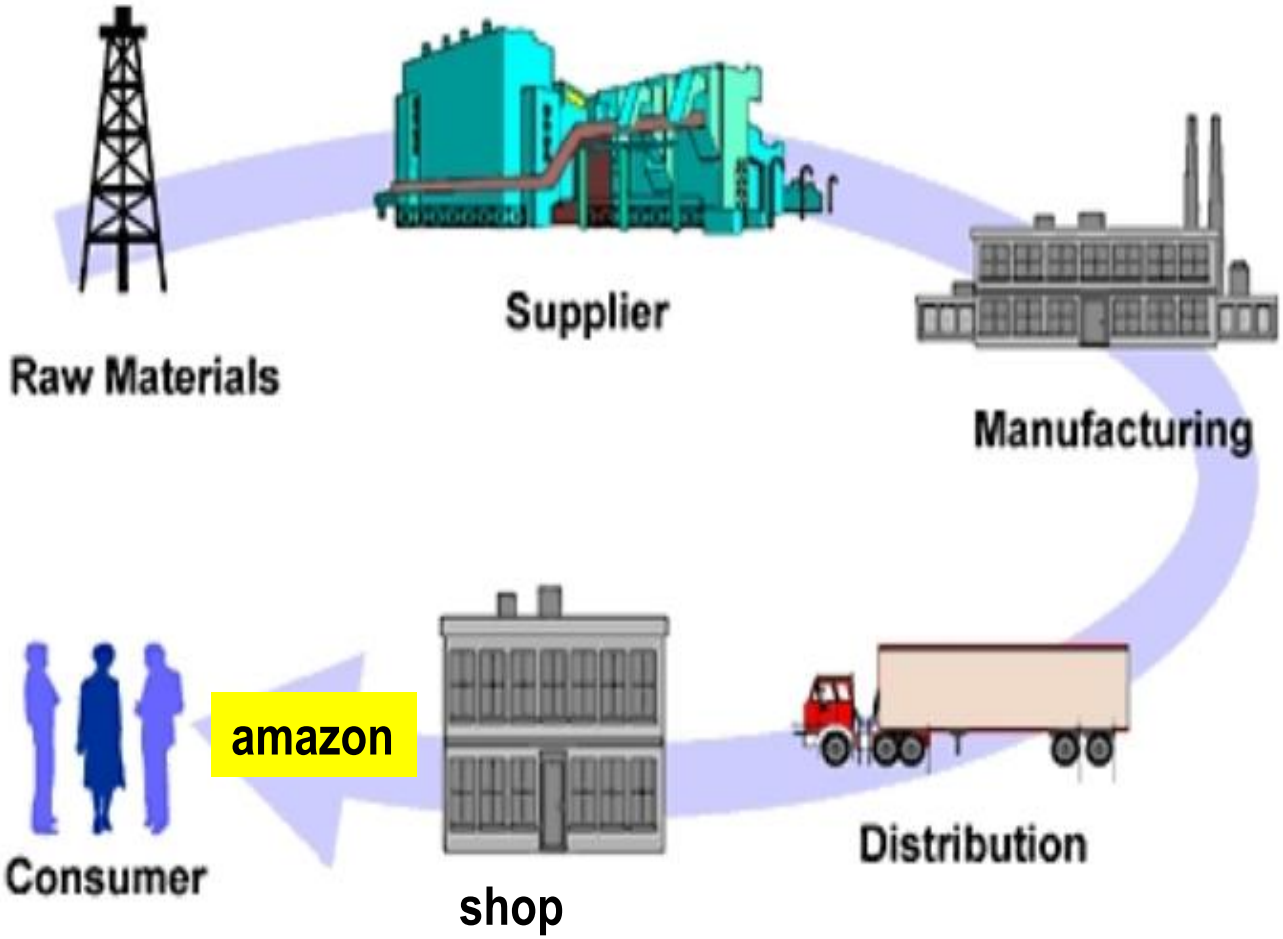
Instead: Construct an adapter, **C**, internally organizing composition, and consider **A • C • B.**

One notion of composition allows a generic, ultimatively simple, infrastructure.

Composition in $\mathcal{C}$ ist *associative*:

$(A \bullet B) \bullet C = A \bullet (B \bullet C)$.

… inevitable for “large” compositions.



RM • Su • Ma • Di • Sh • Am • Co

Summing up: Composition in $\mathcal{C}$ yields a *monoid*

Observation: $\mathcal{C}$ contains $\boxed{e}$ (it holds: $A \bullet e = e \bullet A = A$)

Hence $(\mathcal{C}; \bullet, e)$ is a *monoid*.

… just as the words over an alphabet!

This is the second most beautiful news in recent days

A word on commutativity

Def. An operator **+** on $\mathcal{C}$ is *commutative*

iff for all $A, B \in \mathcal{C}$ holds:

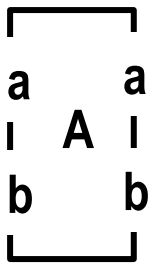
$A + B = B + A$.

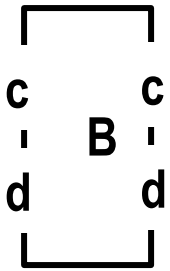
Observation: • is not commutative

Theorem

$A \bullet B = B \bullet A$ holds

if **A** and **B** use disjoint labels.





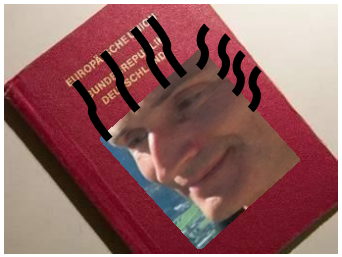
… but Frank loves commutativity!

Frank, please, don't be so stupid!

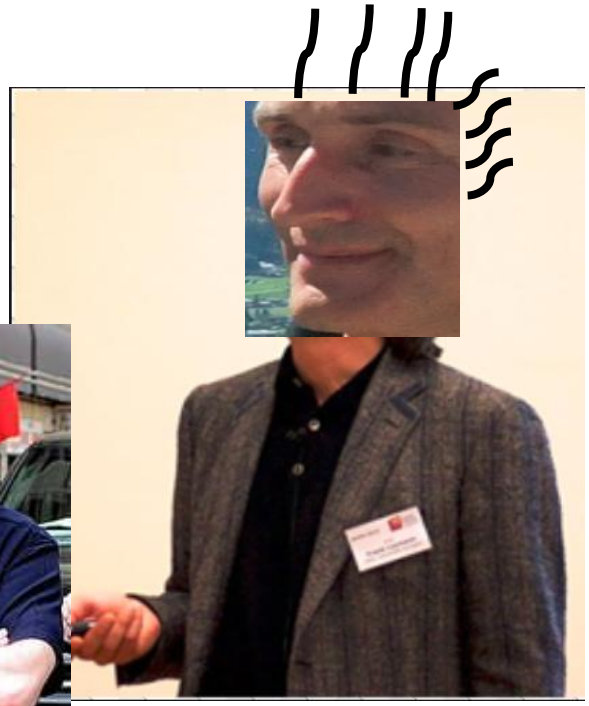
Frank uses three services:

Haircut, new passport, border control.

H



P



B

H • P • B

Good!
Bravo!

… but Frank l… ommutativity!

Frank, pl…

Frank use…

Haircut, ne…

Commutative composition is a bad idea!

H P

P

P • H • B

Bad!

Road map

1. Components and composition:
   the basic paradigm for communication
2. Fundamental properties of the composition operator
3. **Components' contents**
4. … even more general

Remember:

Composition is

- strict and formal for the interface
- entirely liberal for the components‘ contents

Formulate contents (behavior) as you wish!

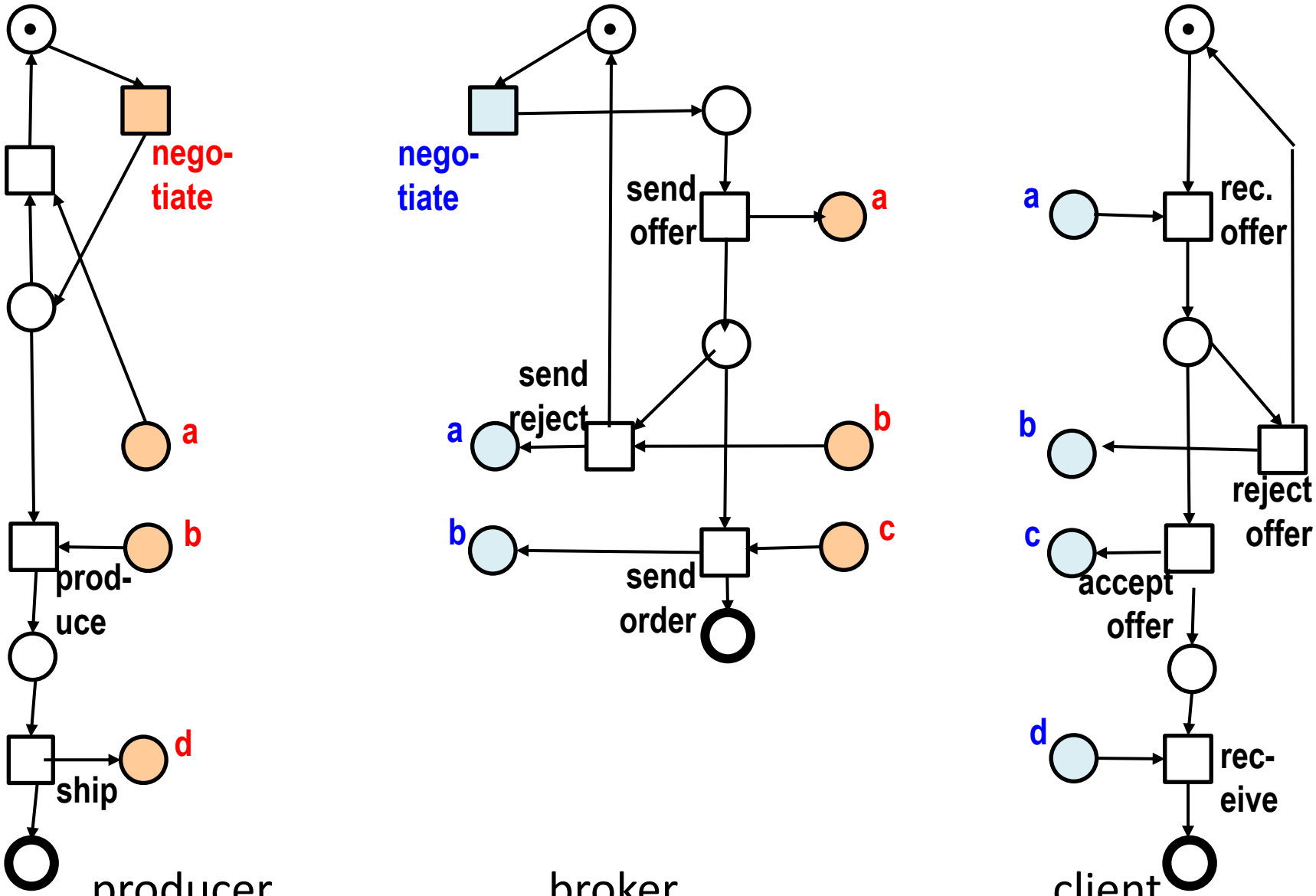
- automata,
- programs,
- π–calculus,
- Petri nets,
- …

Construct classes $\mathbb{D}$
of components such that
$(\mathbb{D}; \bullet, e)$ is a *monoid*.

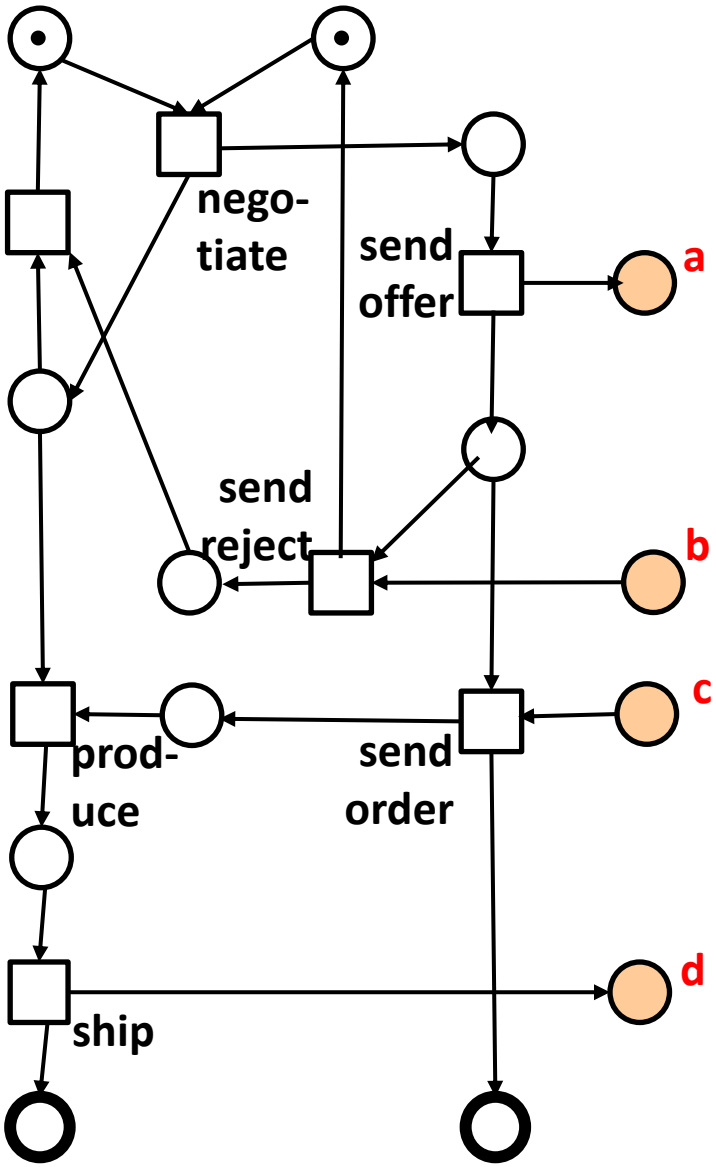
„Instantiations“

The Petri instantiation

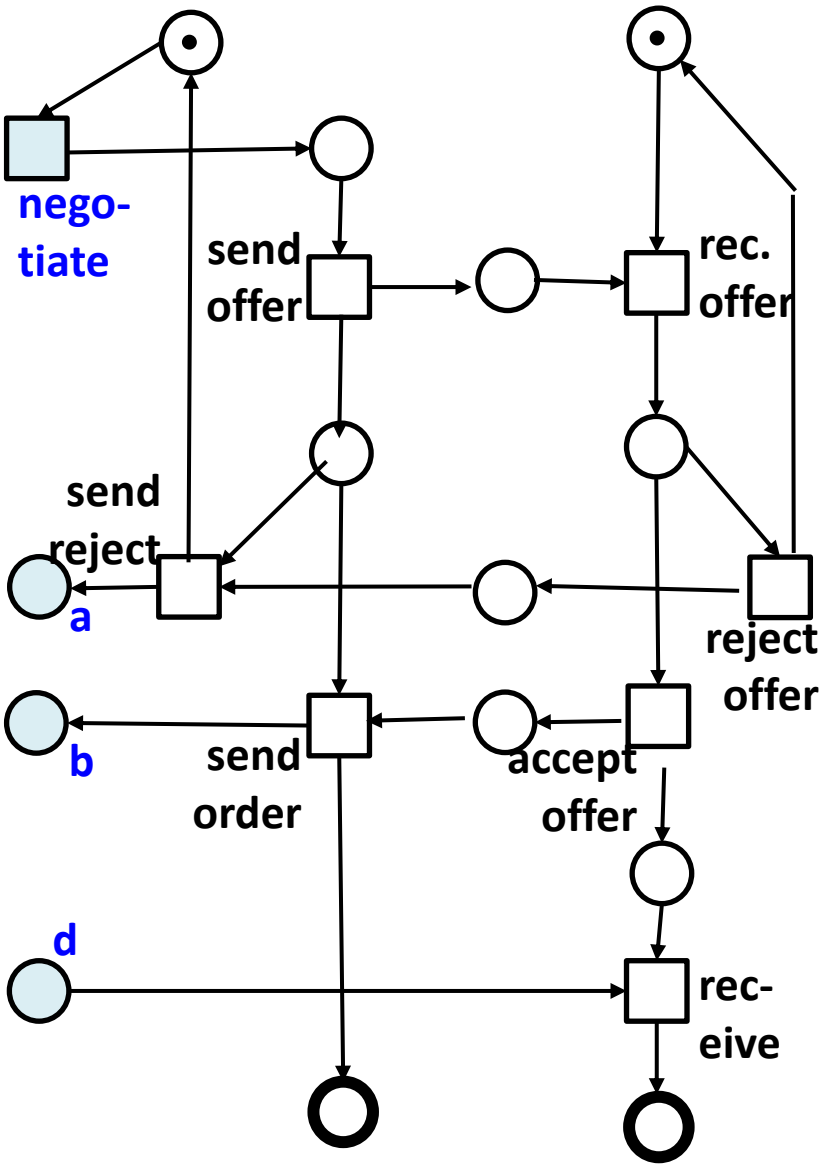
Partition Λ into Λ_P and Λ_T



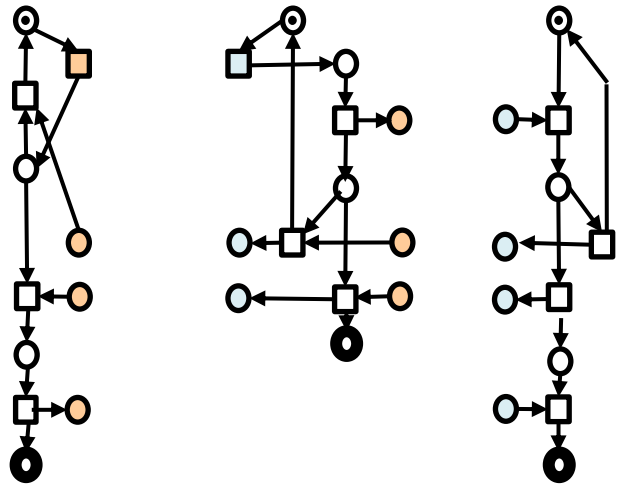
producer • broker

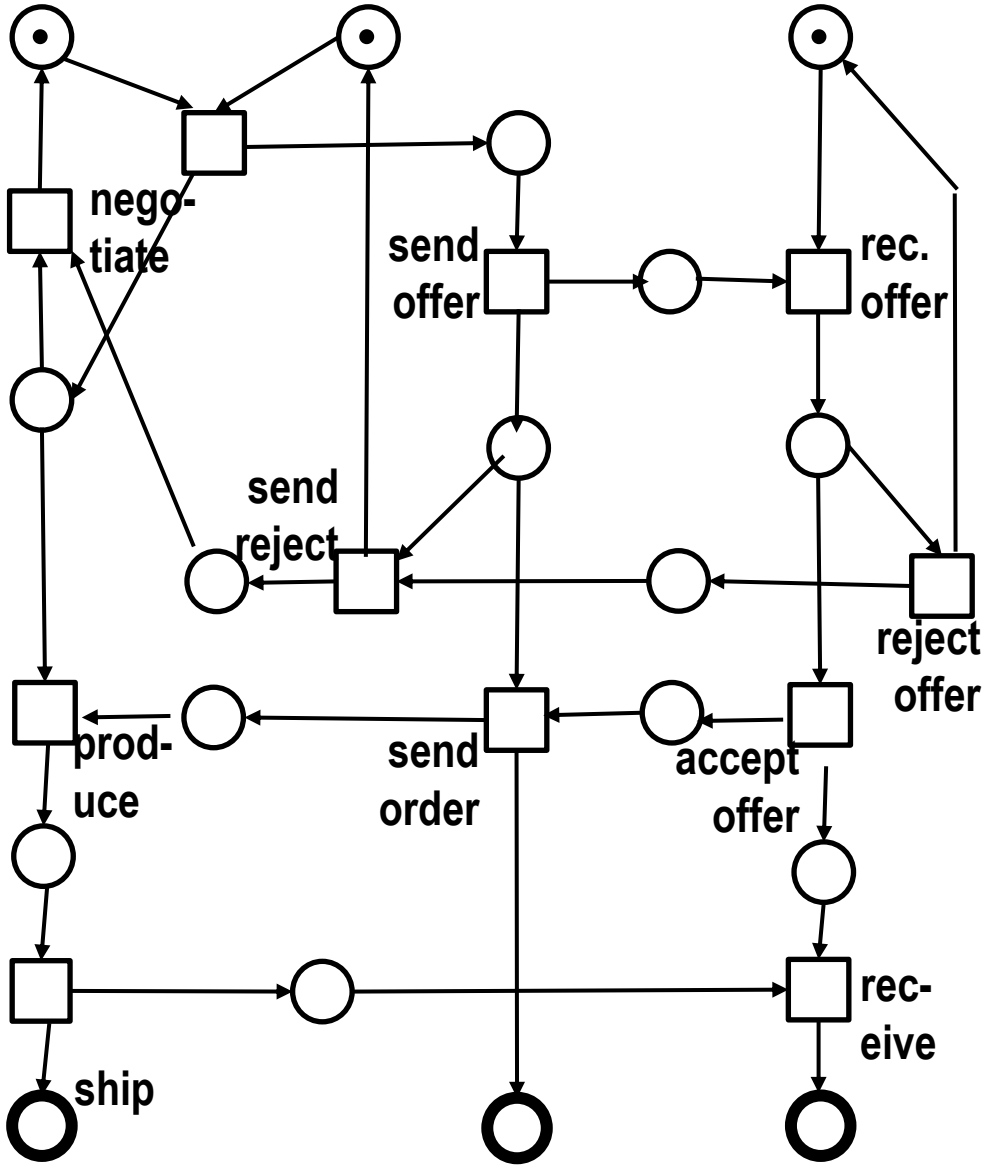


broker • client

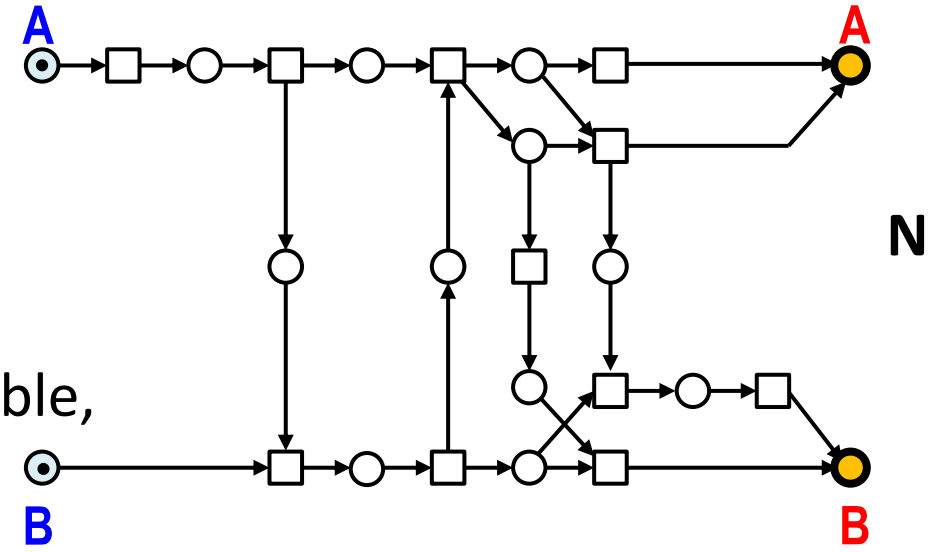


producer • broker • client



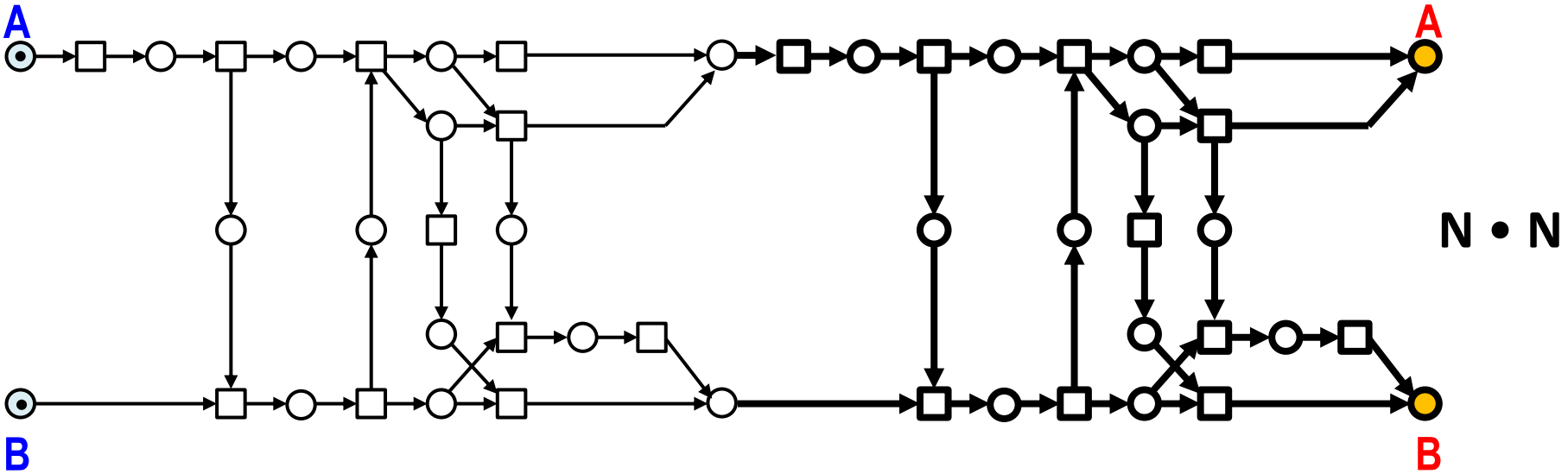


The sound WF instantiation



Def:A workflow is *sound* iff

- all its activities are executable,
- the final state is always reachable,
- upon termination, no further tokens remain.

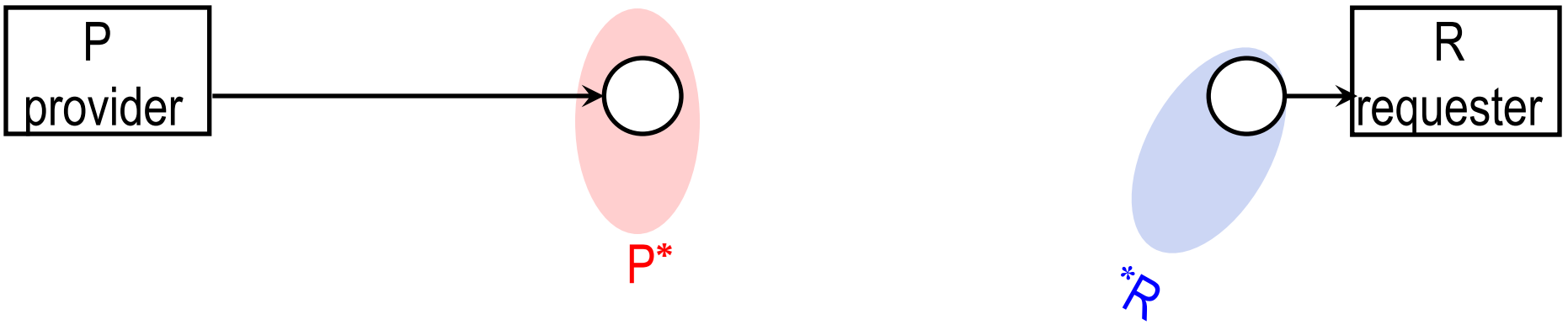


Theorem: Composition of sound workflows is sound.

Road map

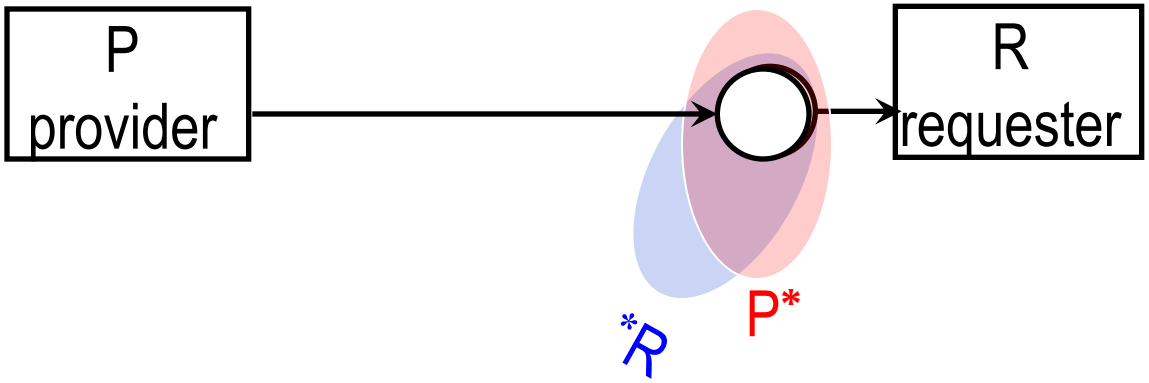
1. Components and composition:
   the basic paradigm for communication
2. Fundamental properties of the composition operator
3. Components' contents
4. **… even more general**

Right and left interface may overlap!

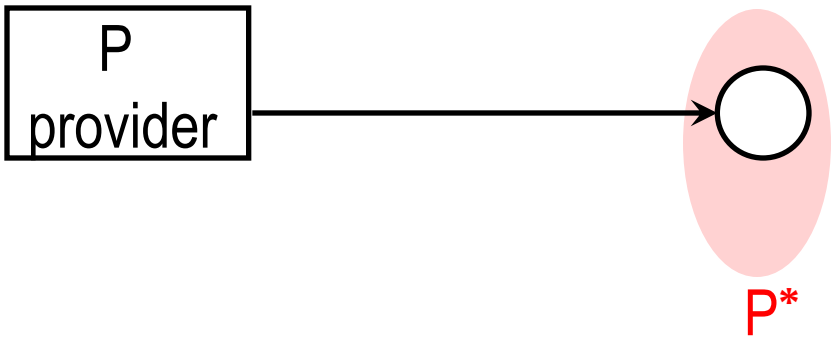


exclusive requester

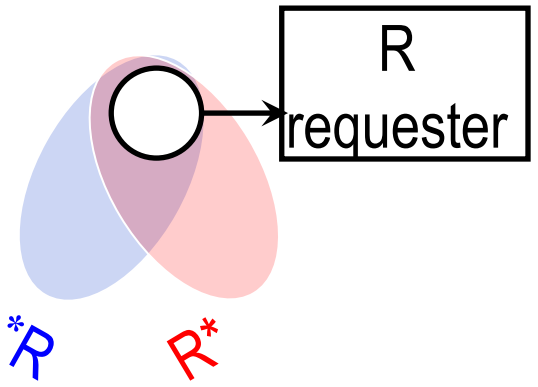
a variant:



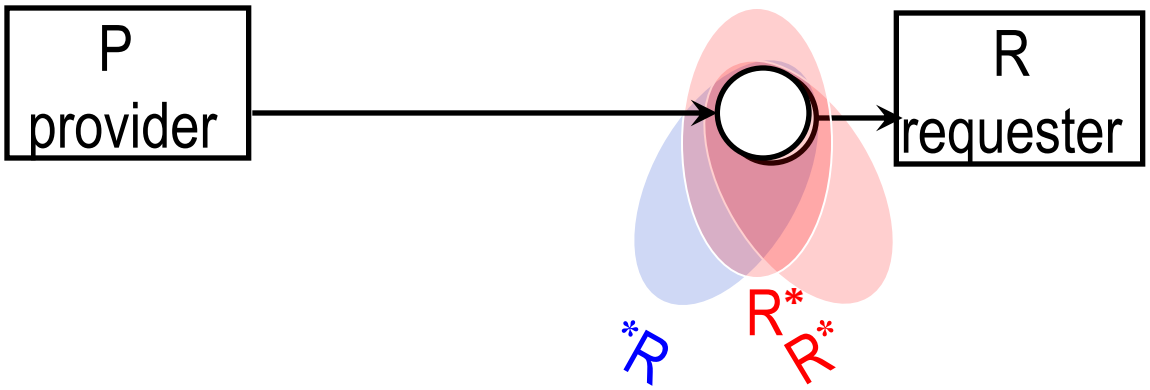
overlapping ports



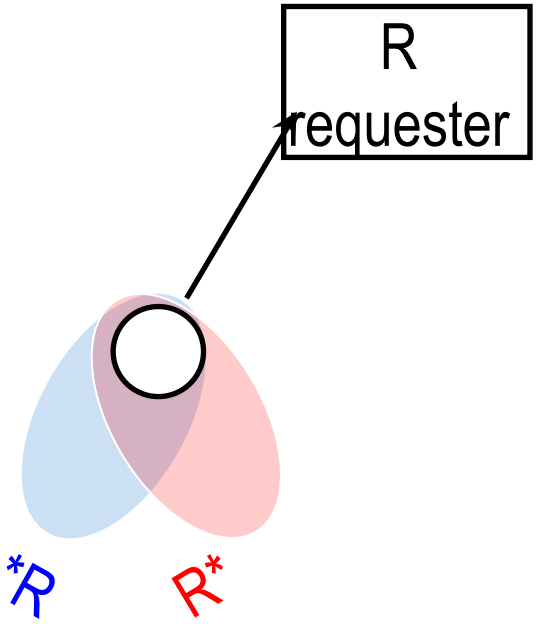
a variant:

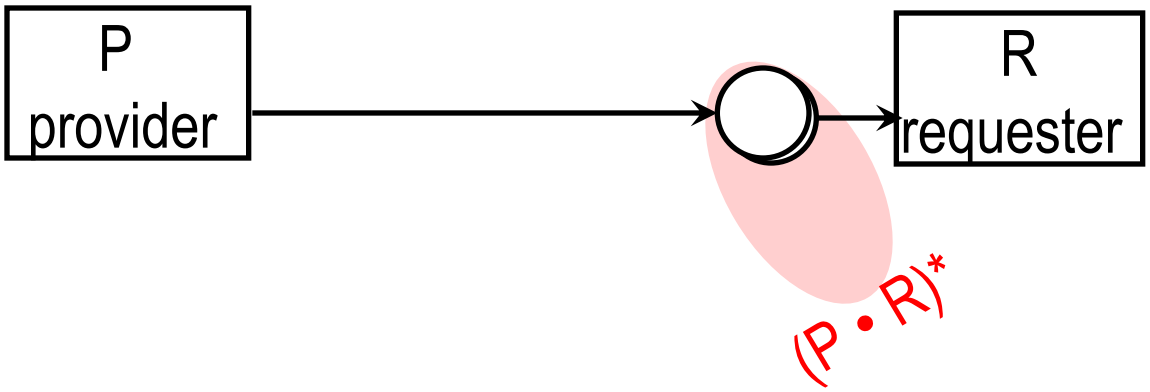


sharing requester

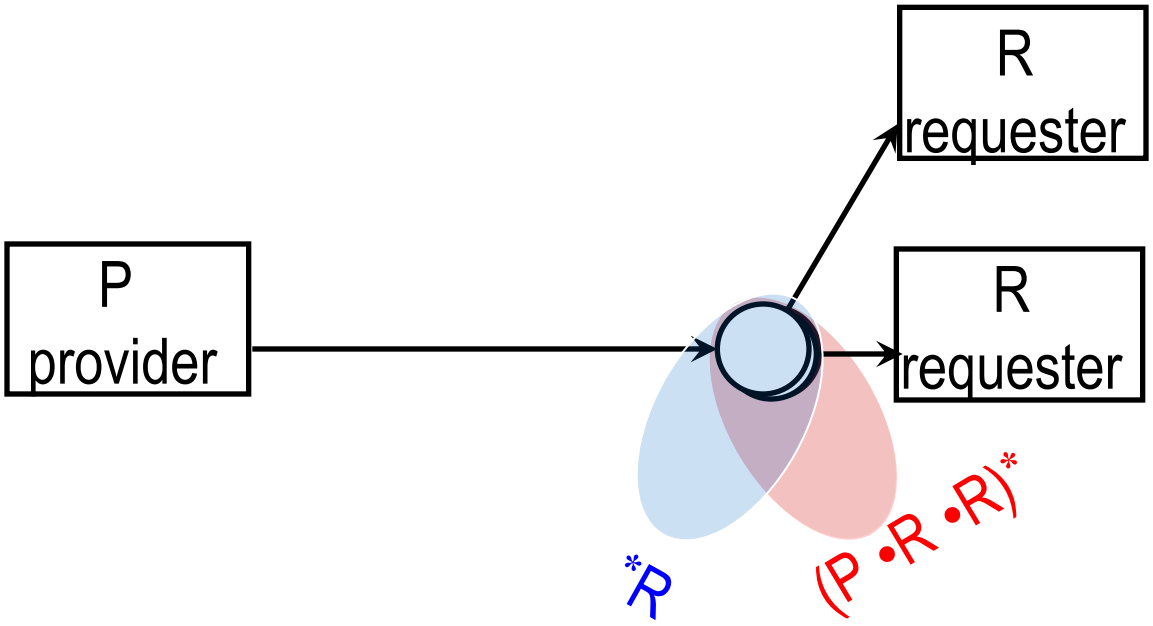


second requester

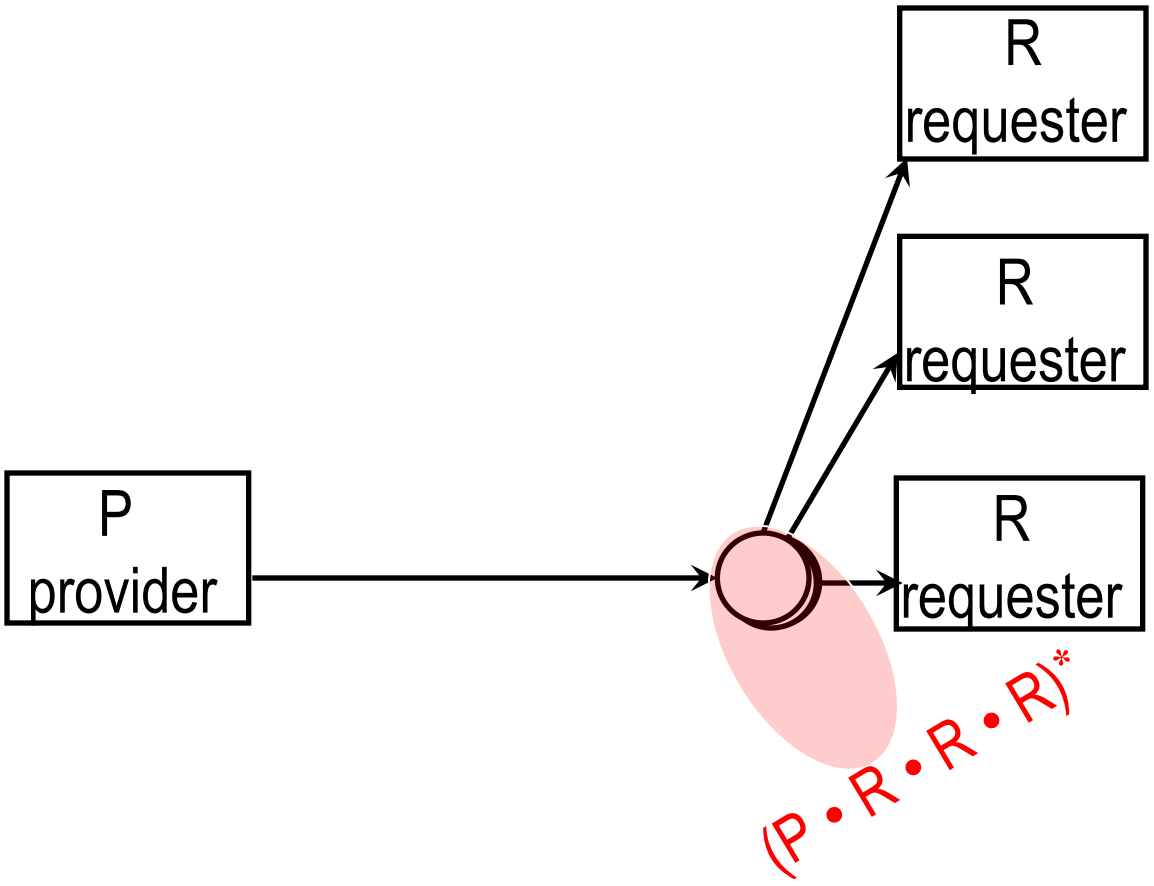




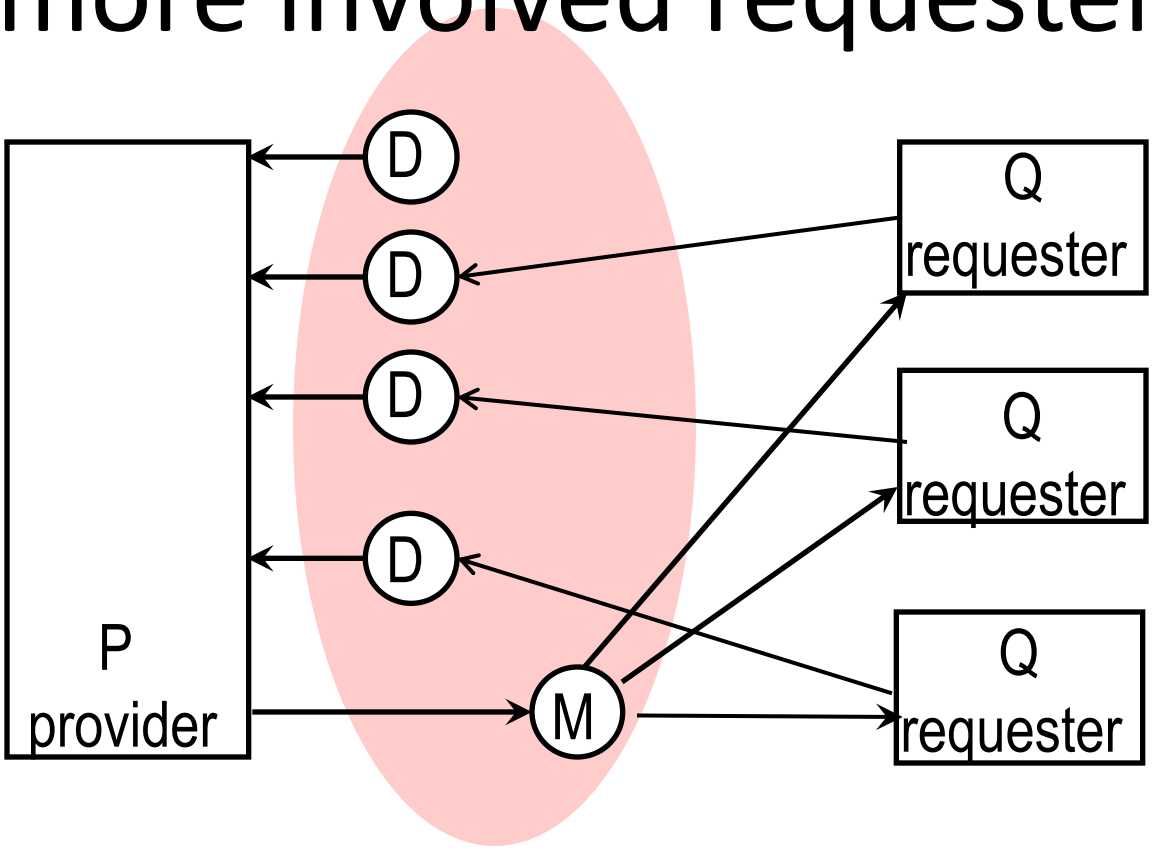
second sharing requester



third requester



more involved requester

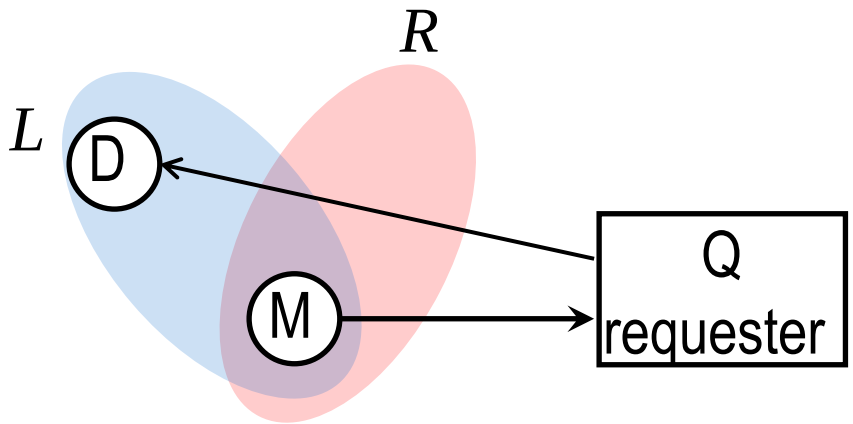


P •Q •Q • Q

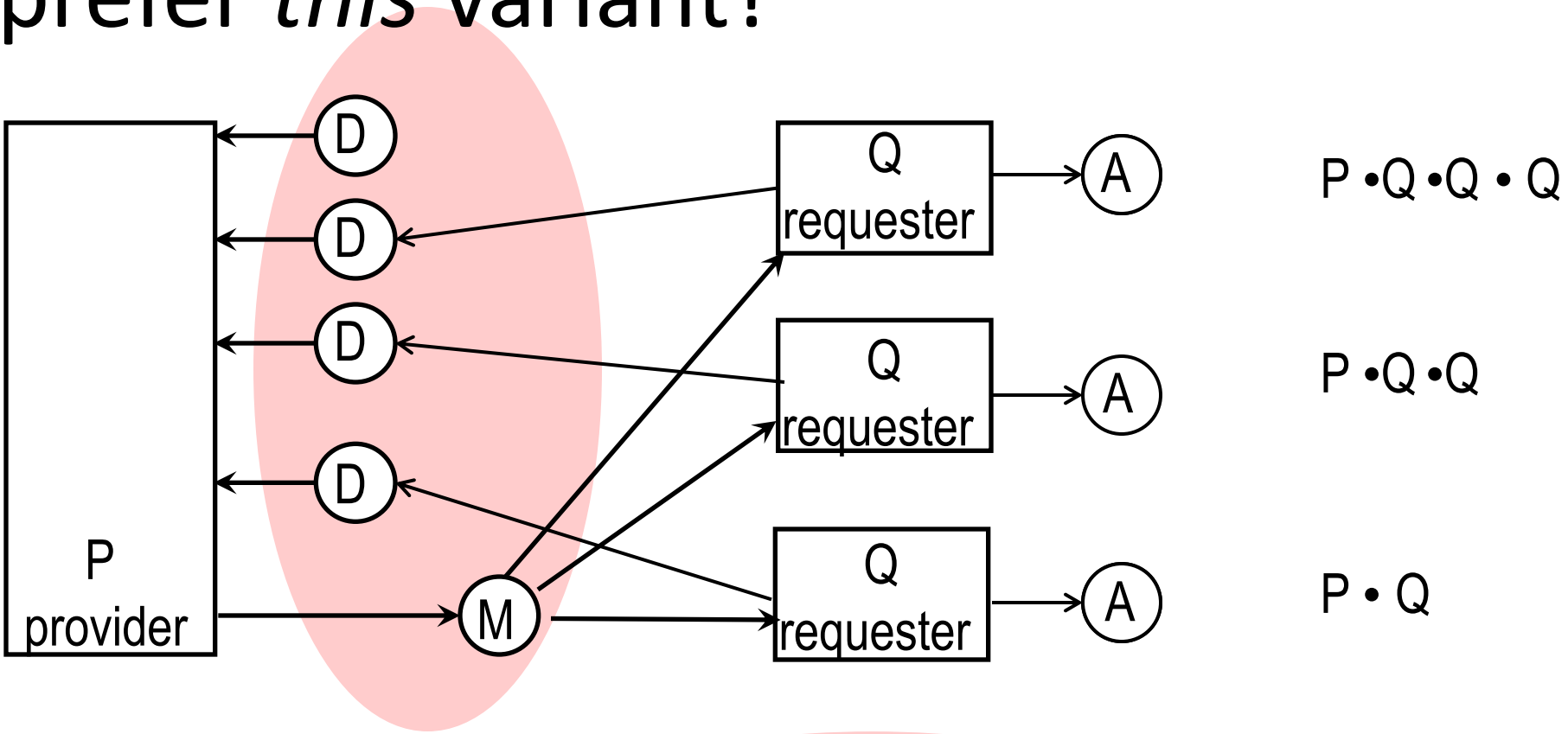
P •Q •Q

P•Q

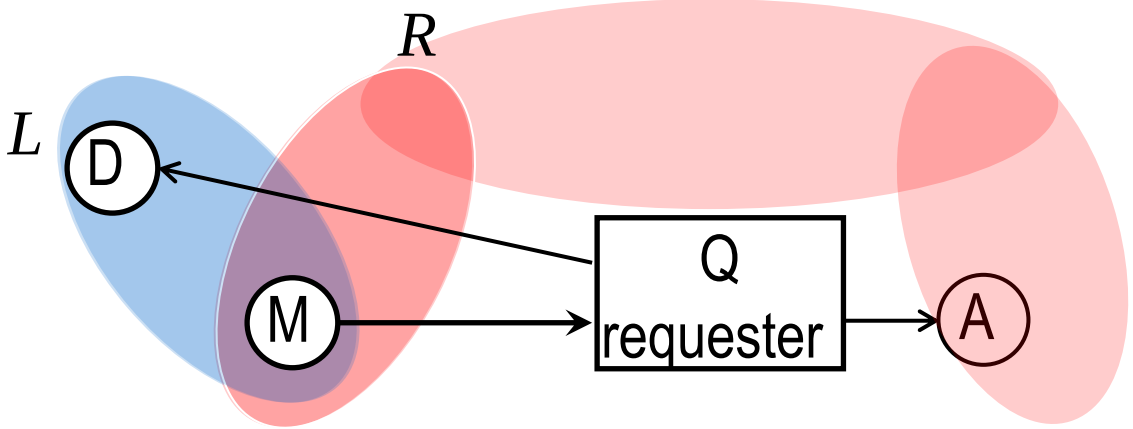
generic reques-ter Q :



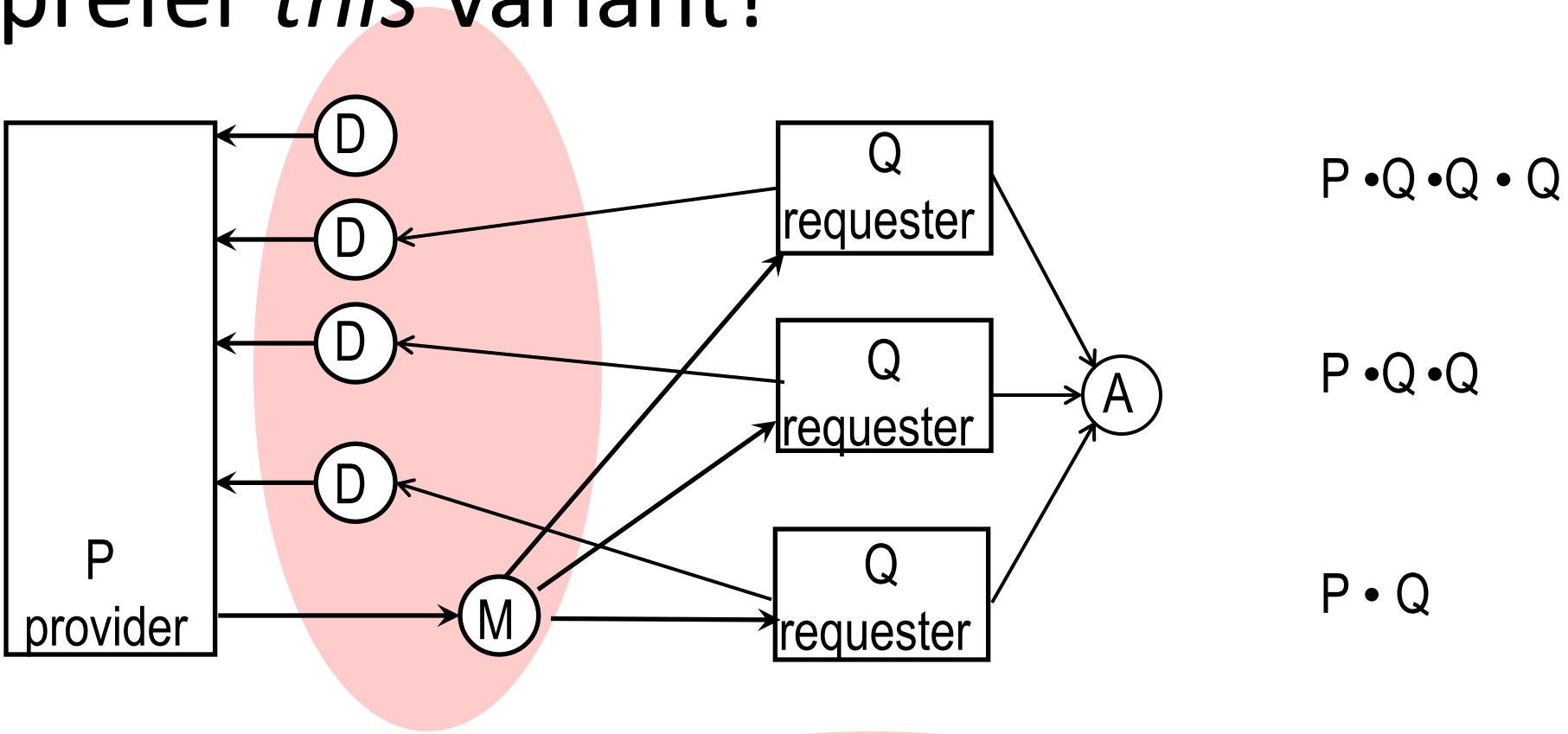
prefer *this* variant?



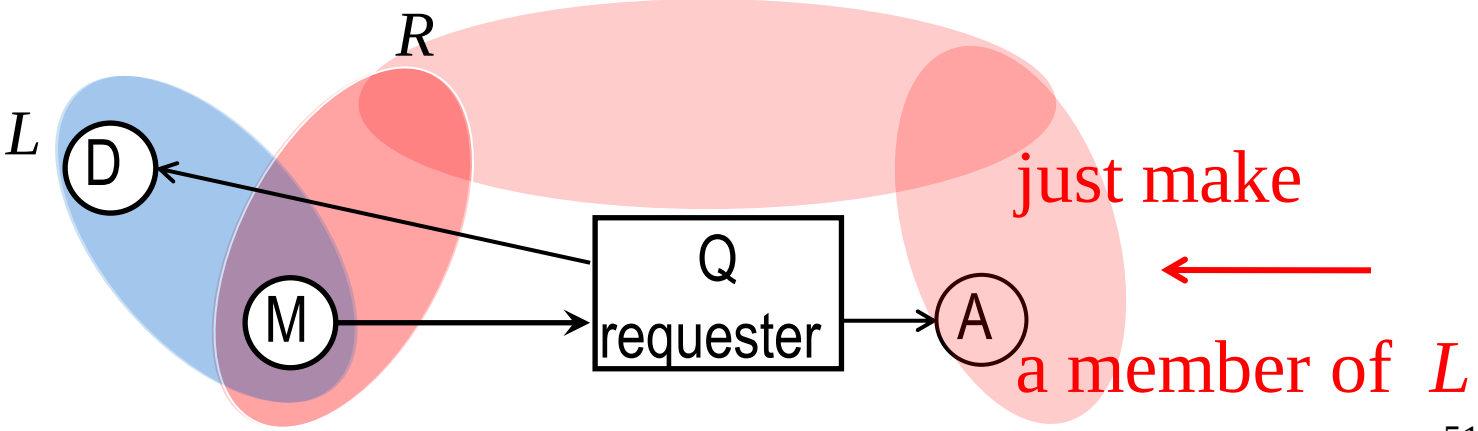
generic requester Q :



prefer *this* variant?



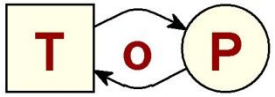
generic requester Q :



SUMMERSOC

Hersonissos, Wednesday, June 27, 2018. 9.30 – 10.30

Conceptual Fundamentals of Reactive Systems



Theory of Programming

Prof. Dr. W. Reisig

Wolfgang Reisig

Humboldt-Universität zu Berlin