SUMMERSOC 2015

Wednesday, July 1st, 2015

Service Orientation as a Paradigm of Programming

Wolfgang Reisig

Humboldt-Universität zu Berlin

Humboldt Universität Informatik

T o P

Theory of Programming

Prof. Dr. W. Reisig

SOAMED

This talk:

1. Prelude: The grand challenge
2. In praise of models
3. Tentative basic notions
4. A notion of composition

5. Marvin Triebel will expand on tools

This talk:

1. Prelude: The grand challenge

2. In praise of models

3. Tentative basic notions

4. A notion of composition

The three paradigms of programming

1. Conventional (procedural) programs

memory cells (*“variables”*) and assignment statements
theoretical foundation / expressive power:
the computable functions

2. Object orientation

attributes and methods
theoretical foundation:
abstract data types / algebraic specifications /
signatures and structures (as for 1st order logic)

3. Service orientation

self contained components (reactive systems)
loosely coupled
theoretical foundation: missing

… generally: reactive systems

Multi-user operating systems and data bases,

computer networks,

embedded systems,

interacting, non-terminating software components,

technical devices or organizational units with local computing power.

portable devices,

control systems,

today’s internet

future internet of things

…

modelling techniques

a heterogeneous world

with loosely related notions, concepts, properties and results:

ALLOY,

B,

BPMN,

event structures,

Message Sequence Charts

live Sequence Charts,

Petri Nets,

Process Algebras,

Statecharts,

UML patterns

… and languages

BPEL *YAWL* *WSDL* *domain specific languages*

no common background

no theoretical foundation

structure of textbooks on SOA

First part:
in plain English:
"… SOA is an implementation independent concept, …"
using many notions, poorly related.

Second part:
examples of implementations
confusing essential aspects and language dependent aspects

What’s the problem ?

… with a formal foundation
of reactive systems (and, hence, SOC)?

THE taboo of Theoretical Informatics:

THE COMPUTABLE FUNCTIONS
ARE *THE* BASIS OF COMPUTING !!!

In principle, everything can be reduced to classical computability

reactive systems (SOC):
- infinite computation are standard (“always on”)
- complexity is not in computation but in *communication*
- computation is not about sequences of symbols

A *canonical fundamental level of abstraction* is missing

A grand challenge:

… a formal foundation for reactive systems (and, hence, SOC)

… in analogy to the computable functions
for sequential, symbol transforming algorithms.

Informatics is more than symbol crunching automata!

In analogy to physics,
informatics is not only pre-Einstein.
It is pre-Newton.

Towards a formal foundation

aim of e.g.

BPMN:

expressive power (L)

all modeling
languages L
for business
processes

results in 204 symbols …

aim of a

generic Mod. language

expressive power (L)

all modeling
languages L
or business
processes

results in ????

A formal foundation is a base to ...

- describe semantics of implementations
- characterize expressivity of formalisms
- relate representations (equivalence, simulation)
- clarify the elementary notions of the area
- derive properties
  from structural and behavioral descriptions
- teach the area systematically

This talk:

1. Prelude: The grand challenge
2. In praise of models
3. Tentative basic notions
4. A notion of composition

Why does Science develop Theories?

THE paradigm : physics, astronomy.

Recently: "theoretical biology“

What about informatics?

1970ies: Intended as a general theory for handling information …

Instead: Informatics became business and technology.

Thesis:

Eventually we need a deep, comprehensive theory of Informatics!

We should learn from physics!

Models

Theory building means to create *models*.

Successful models

- are often intuitively not trival
  and not immediately self-evident
- but provide *structurally simple*
  (and quantifiable) “laws of nature”.

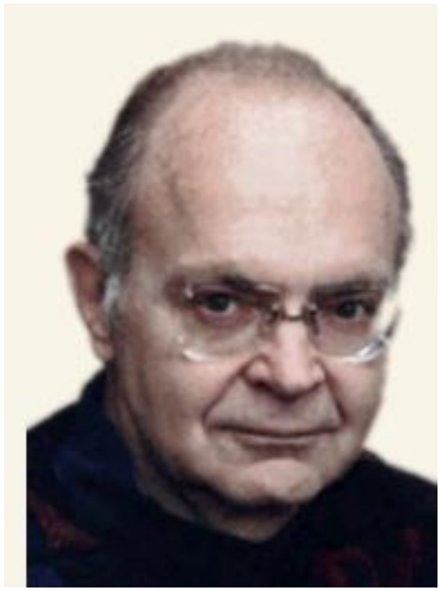
Mature models fit amazingly well with mathematics.

Occam’s razor governs the choice of the “right” model.

A Theory of informatics …

“the next state function f
[of an algorithm]
might involve operations
that mortal man can not always perform.”

Don Knuth, 1968

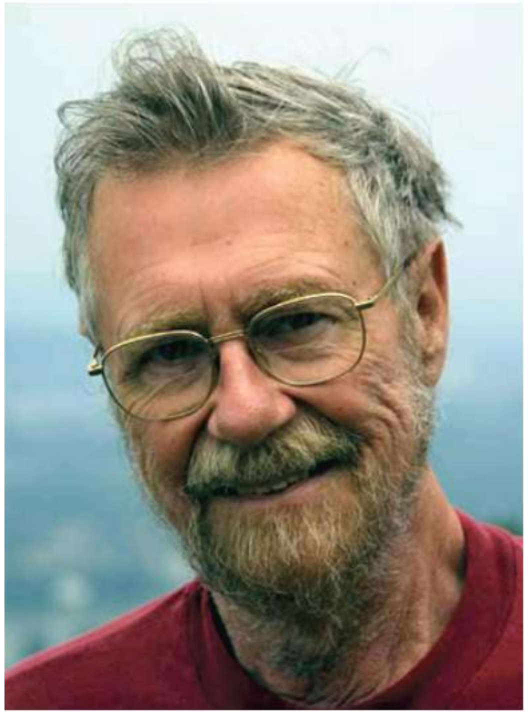


A Theory of informatics …

“Progress is possible only if
we train ourselves to think about programs
without thinking of them
as pieces of executable code. “

“Computer Science is no more about computers
than astronomy is about telescopes.”

E. W. Dijkstra



Models in informatics

„Computer science is a science of abstraction,
creating the right model for a problem
and devising the appropriate mechanizable techniques
to solve it.“

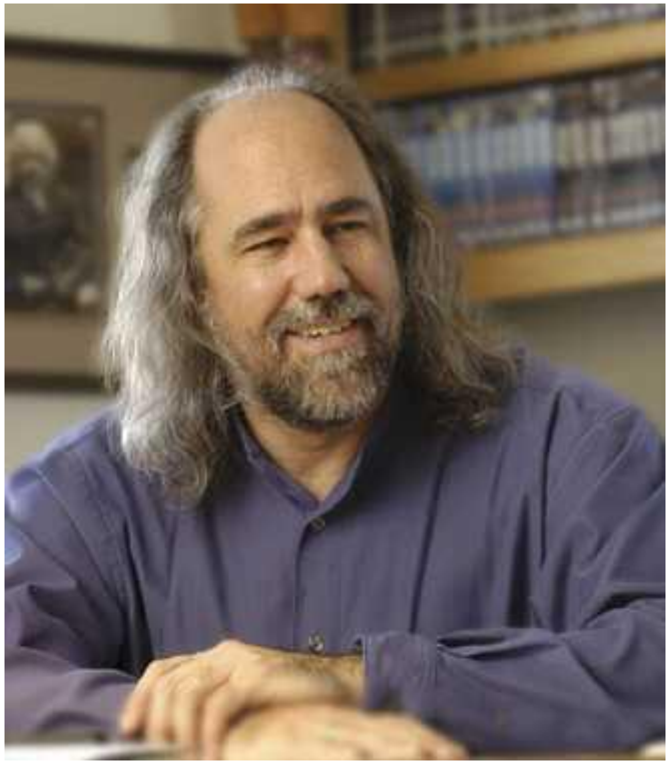
Alfred V. Aho,
Jeffery D. Ullman
1995

Models in informatics

We must “*elevate models*
as to a first class citizenship ...
a peer of traditional text languages
(and potentially its master)”.

“*models as products*”.

Grady Booch, (2004)



Models in informatics

“. . . we should have achieved
a mathematical model of computation,
perhaps highly abstract …
but such that programming languages
are merely executable fragments
of the theory . . .”

Robin Milner, 2005

Adequate modeling techniques for computer embedded systems

… describe structures and algorithms

with components that may never be implemented

user of a cash terminal

software controlled elevator

The modeler freely chooses the level of abstraction

What is a model?

sorry, not in this talk

Model *Jaguar E*

model of the
model Jaguar E

model^2

… too complicated for us

Models in science

… used to describe the laws of nature.

Typical example:

The term “energy”
+ all laws about energy.

There is nothing like *energy* in nature.
The notion of “energy” is an *abstract model*

used to describe an *invariant*.

energy

first hidden in gasoline,
then in acceleration,
then in speed,
then in deformed metal sheet.

What physicists *really* did:
Searched a notion, general enough
to describe what remains invariant
… and called it *energy.*

Scientific models

Physicist do accept intuitively hard models (“*theories*”)

if they offer convincing explanations,

in particular *invariants*.

Invariant in Chemistry

$CH_4 + 2\ O_2 \rightarrow CO_2 + 2\ H_2O$

Search for good theories

= Search for comprehensive invariants. $e = mc^2$

Informatics should learn from this!

Even *Theoretical Biology* is behind (biological) models with nontrivial invariants (“bio mass”)!

Models in informatics

data models

models of computation

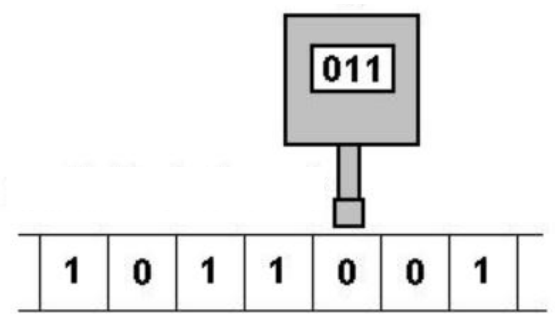
software models

system models

Symbol processing models

"the computable functions"

Turing machines



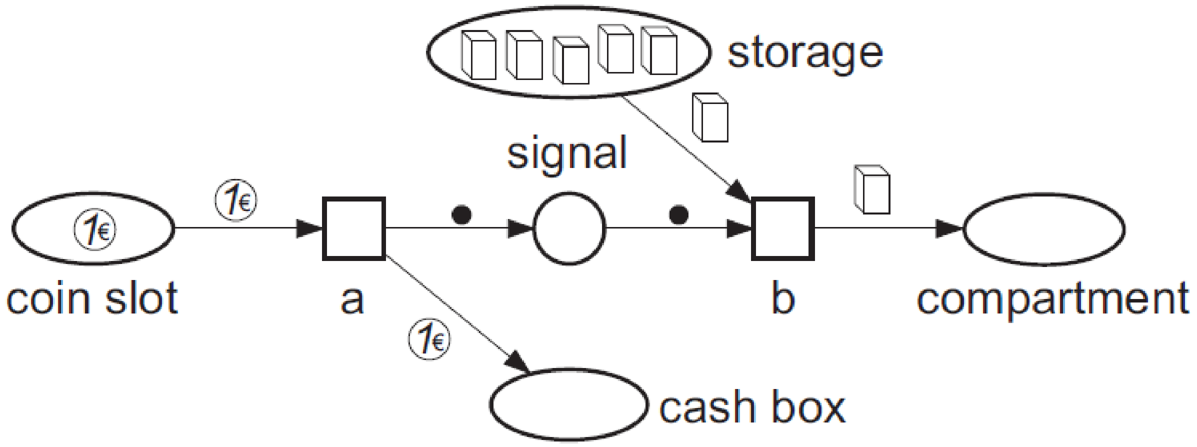
unifying, expressive, no invariants

Programs as models of algorithms

$$\textbf{while } (x < 10)\ x := x + 1$$

invariants:
Hoare Logic

Behavioral models

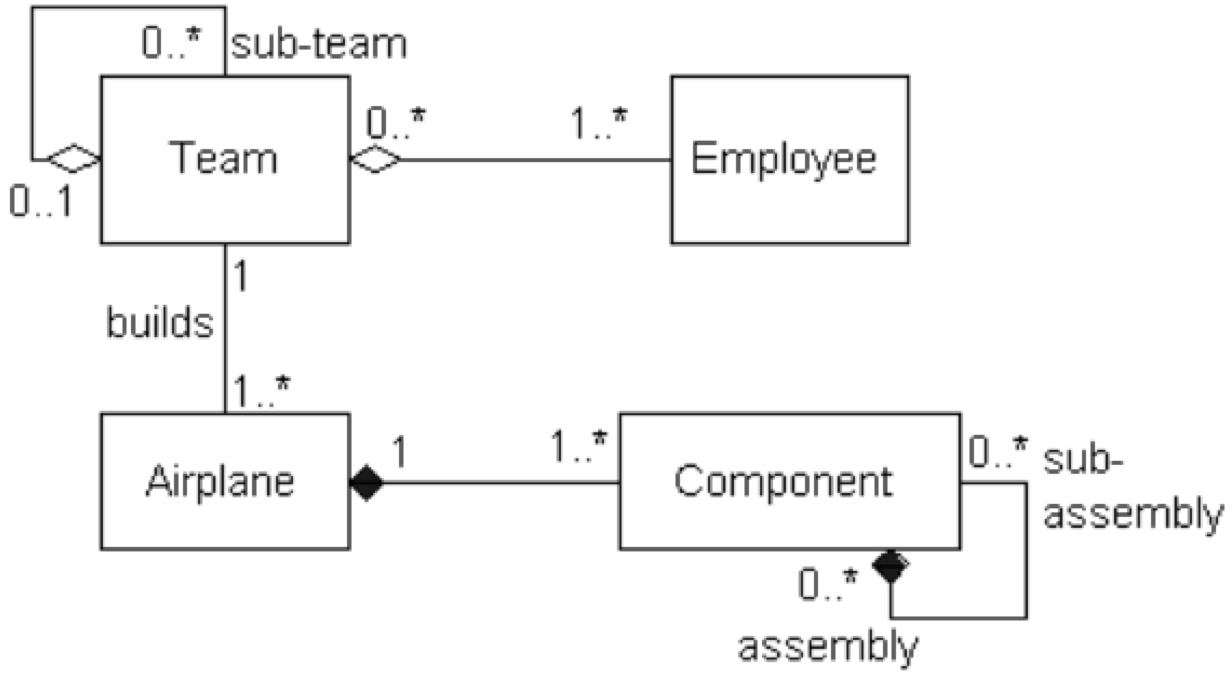


invariant:
cash box + storage = signal + 5

Petri nets have expressive invariants,
because transitions are reversible.

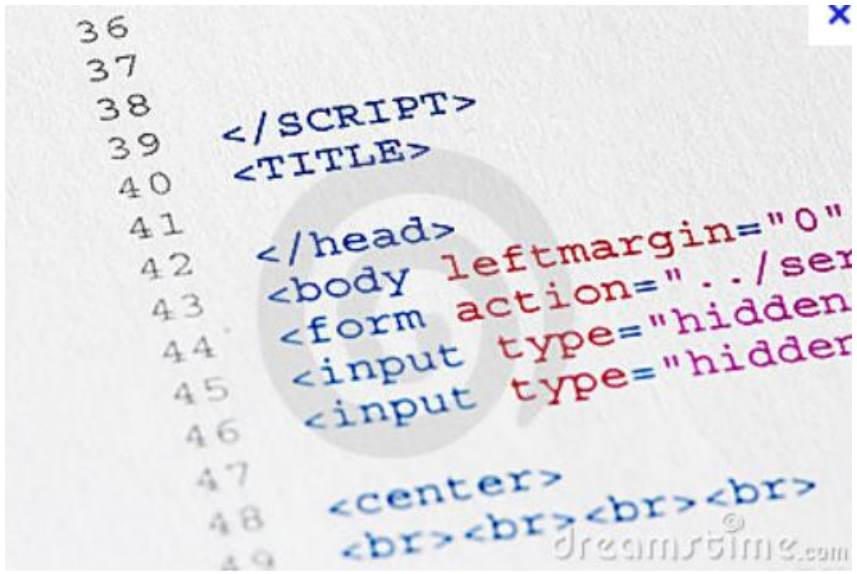
Software models

UML



not formal,
hence no invariants

… the blunt reality



```
</SCRIPT>
<TITLE>

</head>
<body leftmargin="0"
<form action="../ser
<input type="hidden
<input type="hidder

<center>
<br><br><br><br>
```

… Software engineers ignore modeling

why is it?

The software industry doesn't benefit substantially.

… because models are complicated?

no! because a software developer don't get much out of a model

Needed: more *fitting* models !

- for entire *systems*, not (only) computing components;

- allowing free choice of level of abstraction;

- representing “the implementable”
  (not “the computable”);

- including a **comprehensive** notion of “algorithm”;

- **providing much more insight than today’s models!**

What notions may be subject to such models?

information / data / documents / items / messages / contracts

copy / compose:

What aspects change?

What are the properties of a copy / a compositum?

access rights, ownership of ~

dispatch, store , disseminate ~

communicate ~

computer-mediated

Activities / tasks

what means

to cancel ~

to authorize ~

to delegate ~

to synchronize ~

to re-organize ~

More general invariants

What remains invariant when using

- a cash machine

account
+ in hand

- a garbage collector
- a communication protocol
- an elevator control?
- a telephone switching system

Prospective theorems on software models

Theorem 1: In each computerized system holds:
While computing
– without communicating –
the amount of *information* (?)
remains constant

Theorem 2: To decide an alternative =
to consume a piece of information

This talk:

1. Prelude: The grand challenge
2. In praise of models
3. Tentative basic notions
4. A notion of composition

What is a *service*?

… an algorithmic component, frequently software.

software to

- *book a journey,*
- *sell a ticket,*
- *offer cash at an ATM.*

a person

- *booking a journey,*
- *buying a ticket,*
- *withdrawing cash from an ATM.*

a technical system,

- *elevator*
- *self driving vehicle*
- *mobile phone*

an organization, providing

- *insurances*
- *medical surgery*

Three distinguishing aspects

a) A service is *always on.*

In general NOT:

Input as last time

yields output as last time

b) services interact *loosely coupled.*

In general: message passing; not handshaking.

c) A service may spawn many *instances.*

Two instances may

- temporally overlap,
- interact.

Interaction of services

Interaction is ***the*** fundamental idea of services

you can not kiss by yourself

… represented as *composition*

For services *P* and *S,*
the composition $P \oplus S$
is a service again.

Frequently, $P \oplus S$ does not interact any more

ticketing $=_{\text{def}}$
sell_ticket $\oplus$ buy_ticket

Services interact goal oriented

interacting services (instances) jointly pursue a *goal*.

They may *reach* their *goal*

you can not kiss by yourself

or *miss* it

automaton got my money

I got a ticket

automaton expects my input

I expect adviece

Frequent goal of a set of services:

to reach a final state together

Often:

services play the role of a *provider* or a *requester*,

together with a *broker*.

beauty predicates

$P \oplus S$ is *beautiful,*

in case P and S both reach their goal in $P \oplus S$

(may be, by he help of a third service).

The algebraic structure of services

Given:

- a set $\mathbb{S}$ of *services,*
- a composition operator $\mathbb{S} \times \mathbb{S}$ ♎ $\mathbb{S}$,
- a predicate $\beta \subseteq \mathbb{S}$.

This yields the algebraic structure

$$(\mathbb{S}; \oplus, \beta).$$

For $R, S \in \mathbb{S}$,

R is a *partner* of *S*,

iff $R \oplus S \in \beta$. $\beta(R \oplus S)$

Let sem(*S*) $=_{\text{def}}$ the set of

all partners of *S*.

derived notions:

S may be *substituted by S‘ :*

sem(*S*) $\subseteq$ sem(*S‘*)

R and *S* *are equivalent:*

sem(*R*) = sem(*S*)

T *adapts R* and *S:*

$R \oplus T \oplus S \in \beta$

The fundamental notions and problems

Notions	*Problems*	*Tools*
Services are *modeled.*	Formalization	
Services are *composed* ($R \oplus S$).	Formalization	
A (composed) service may be *correct (w.r.t. β).*	Verification	
Each service has a set of *partners*.	partner synthesis	
U adapts R and *S* iff $R \oplus U \oplus S$ is correct.	adapter synthesis	

tool chain

service-technology.org

next talk

by Marvin

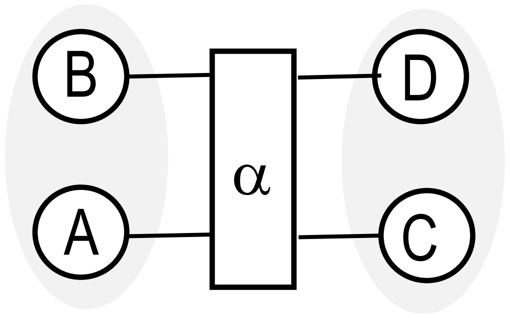
This talk:

1. Prelude: The grand challenge
2. In praise of models
3. Tentative basic notions
4. A notion of composition

An abstraction of services: components

A component has an *inner structure* and an *interface*.

Typical example:



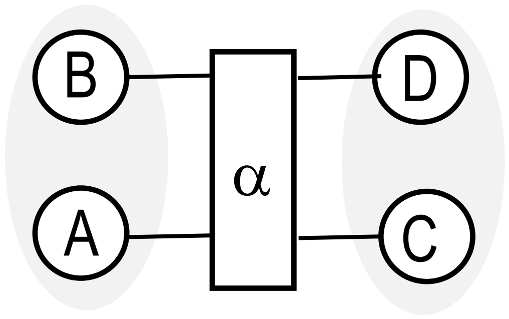
with nodes A, B, C, D
as its interface
and node α
as its inner structure.

technically:
a component is
a node labeled graph.

Some nodes
constitute ist interface

Composition

Components are intended to be *composed* along their interface.



What we want:

a relevant class ***C*** of components such that

composition of components „ 👓 " is

- *total* i.e. 👓 : ***C*** × ***C*** $\underline{\Omega}$ ***C***

  A 👓 *A* or *A* 👓 *B* 👓 *A* etc. are well defined,

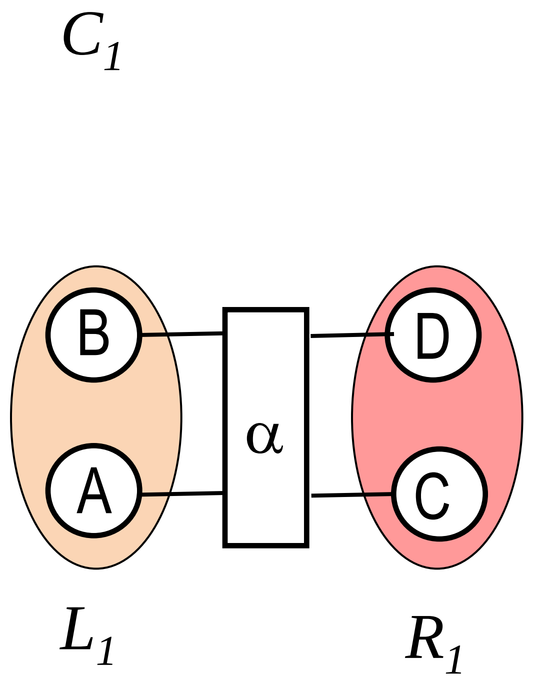
- *parameter free*, i.e. no 👓$_i$ for any kind of parameter, *i*

- *associative*, i.e. (*A* 👓 *B*) 👓 *C* = *A* 👓 (*B* 👓 *C*)

- flexible enough to cover many realistic applications.

remember: T *adapts R* and *S:*
R ⊕ *T* ⊕ *S* ∈ *β*

Components with left-right interface



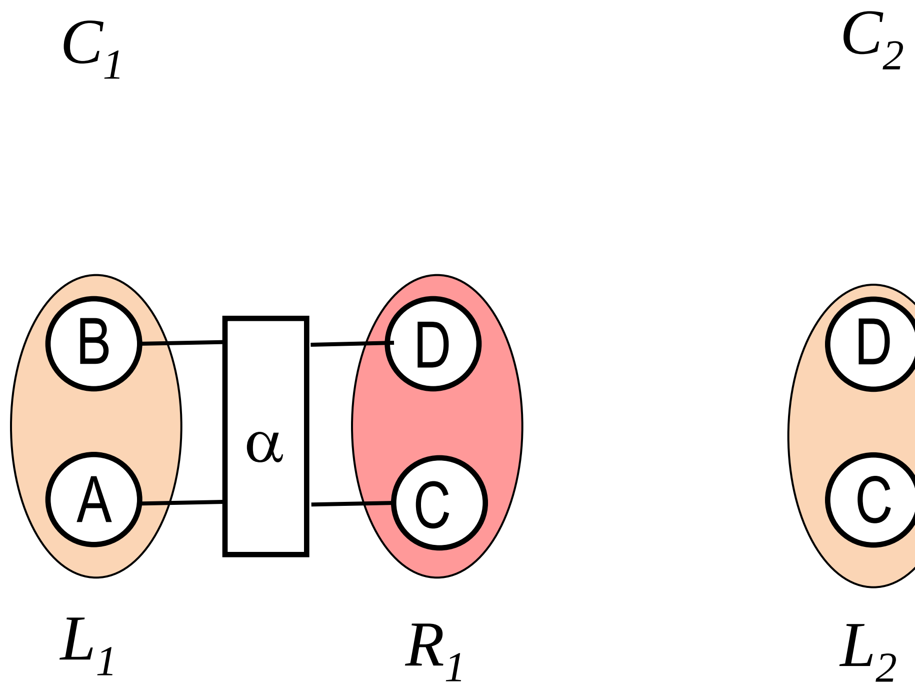
The component's *interface*:
the left and the right *port*.

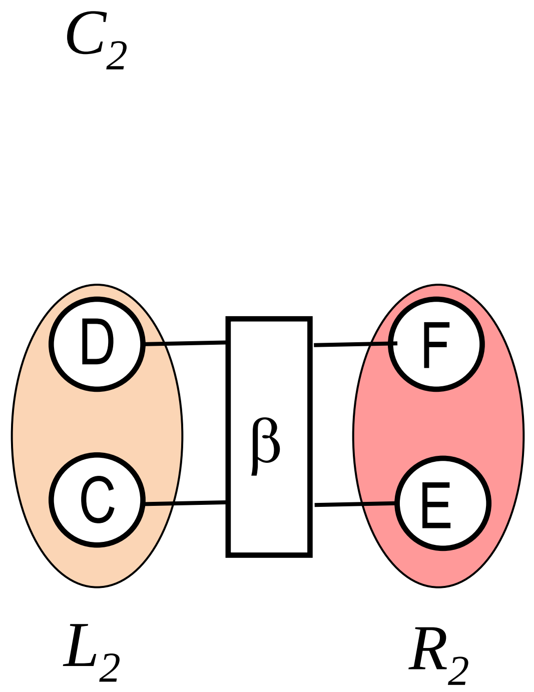
Each port: a set of (labelled) *nodes*.

Two ports are often adequate:

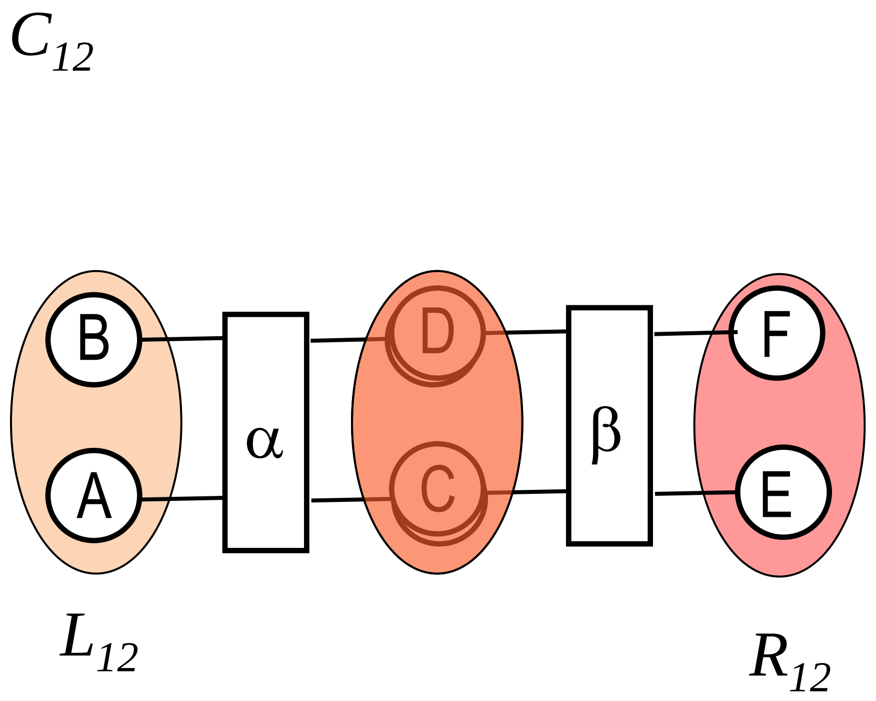
input	and	*output*
customer	and	*supplier*
provider	and	*requester*
producer	and	*consumer*
buy side	and	*sell side*

R_1 and L_2 fit perfectly

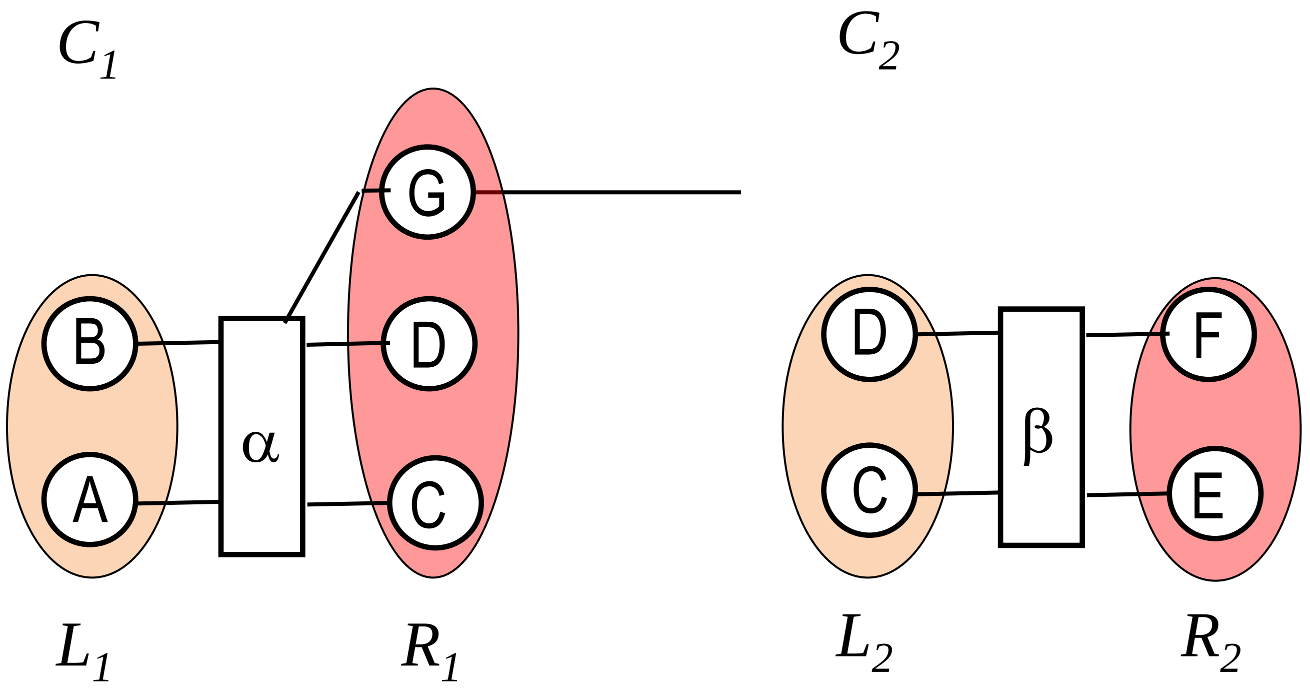




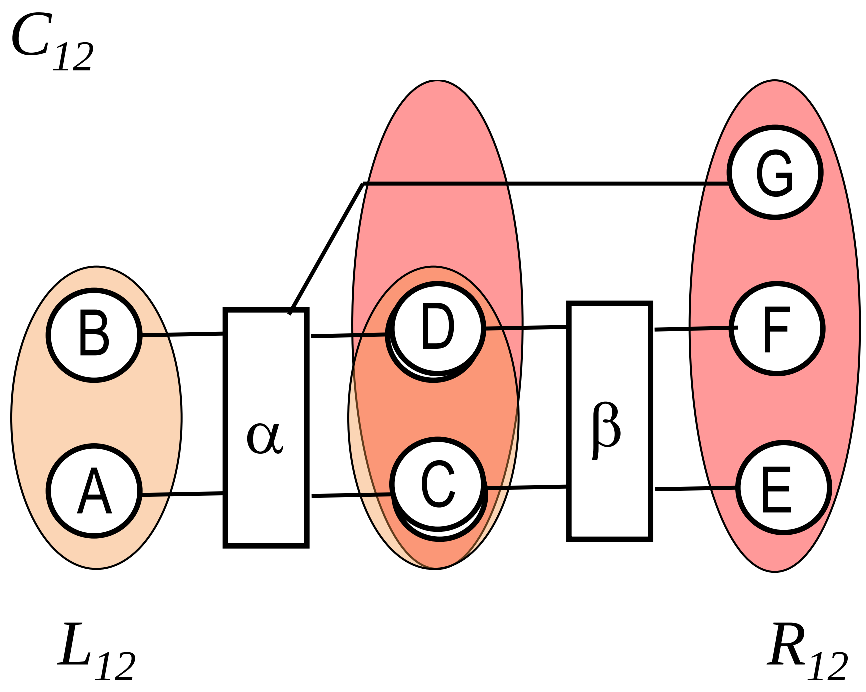
Composition C_1 👓 C_2



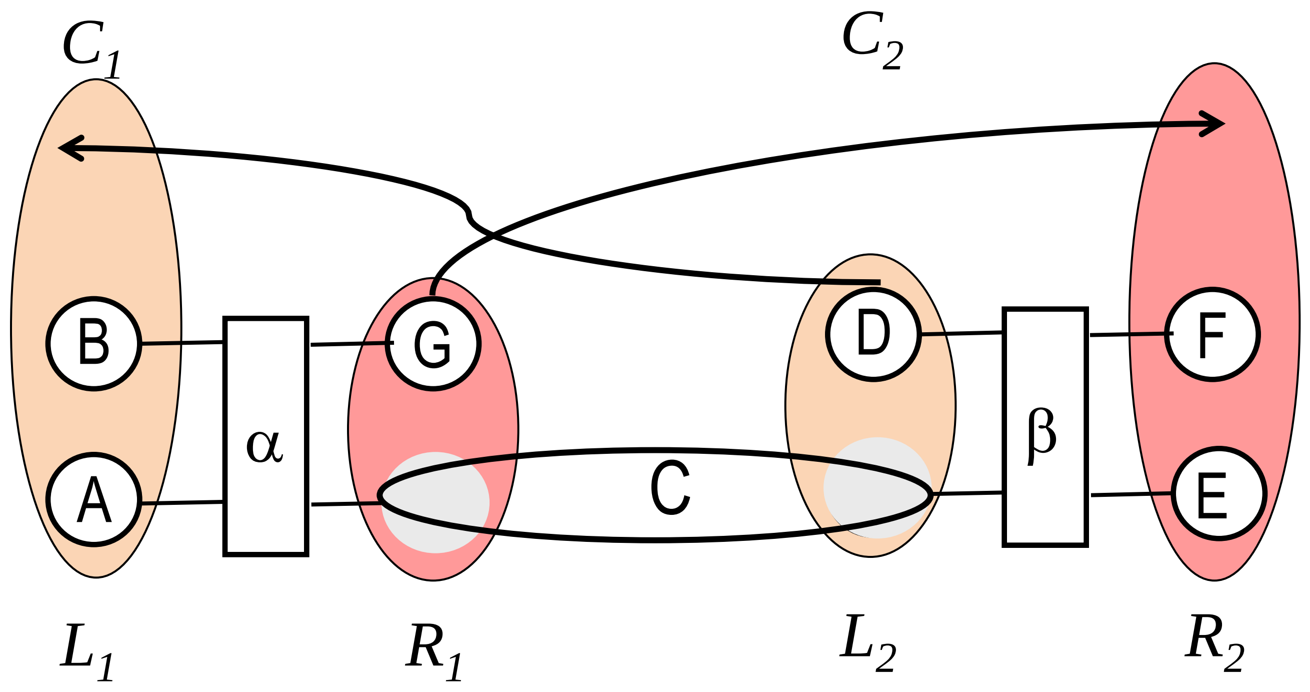
… it is not always that simple



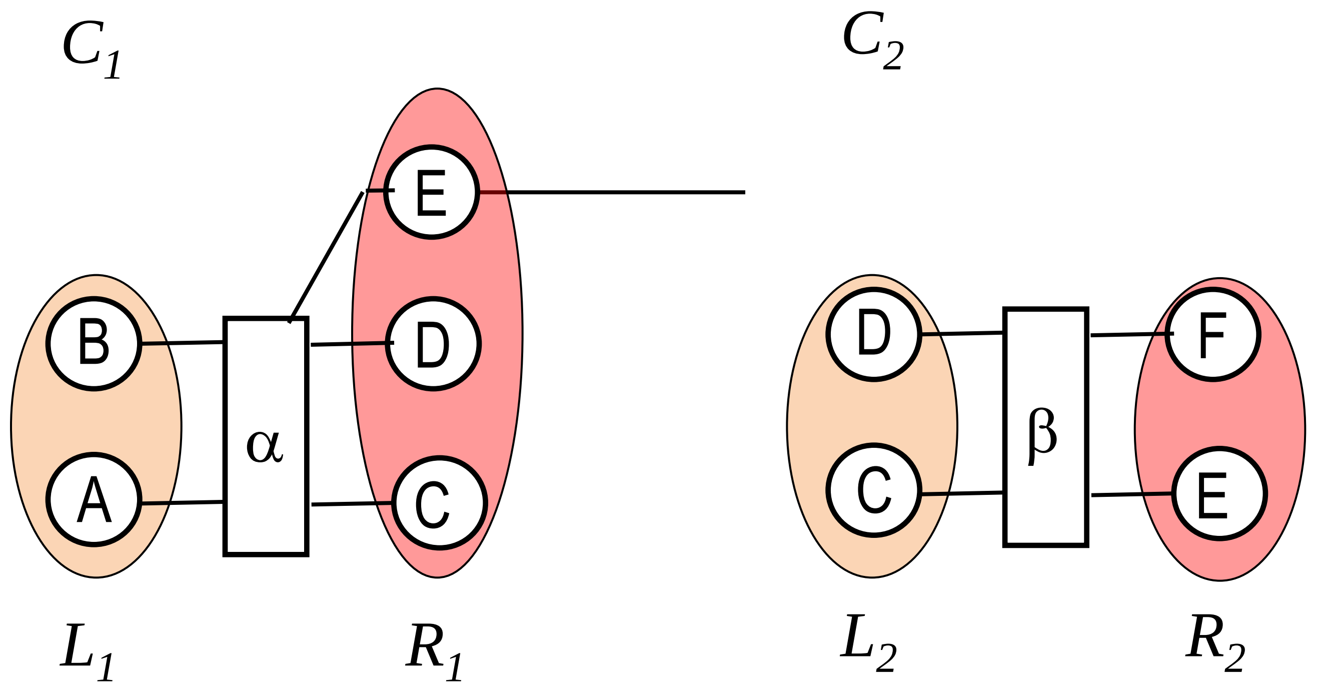
Composition C_1 👓 C_2



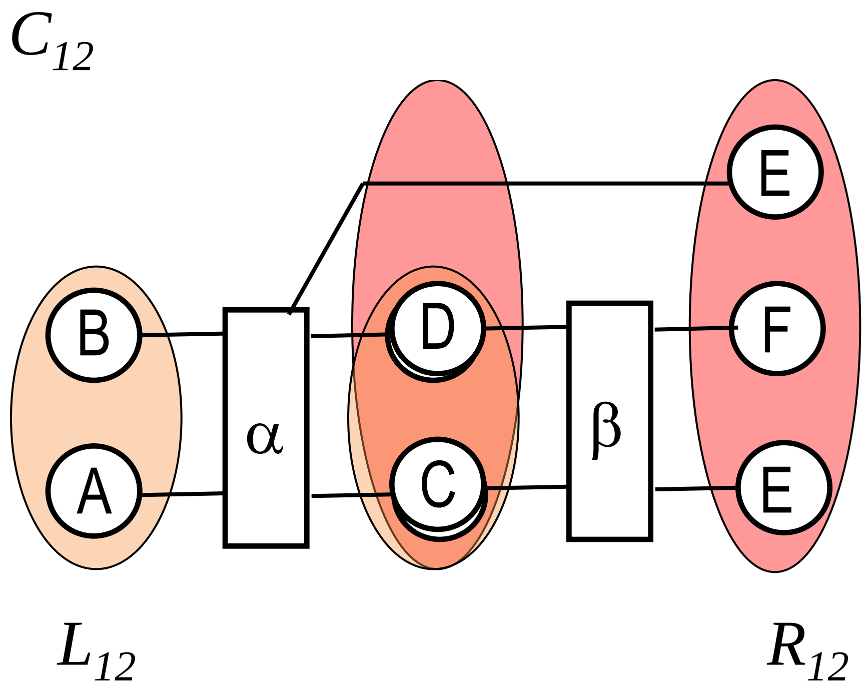
This works nicely:



… unfortunately



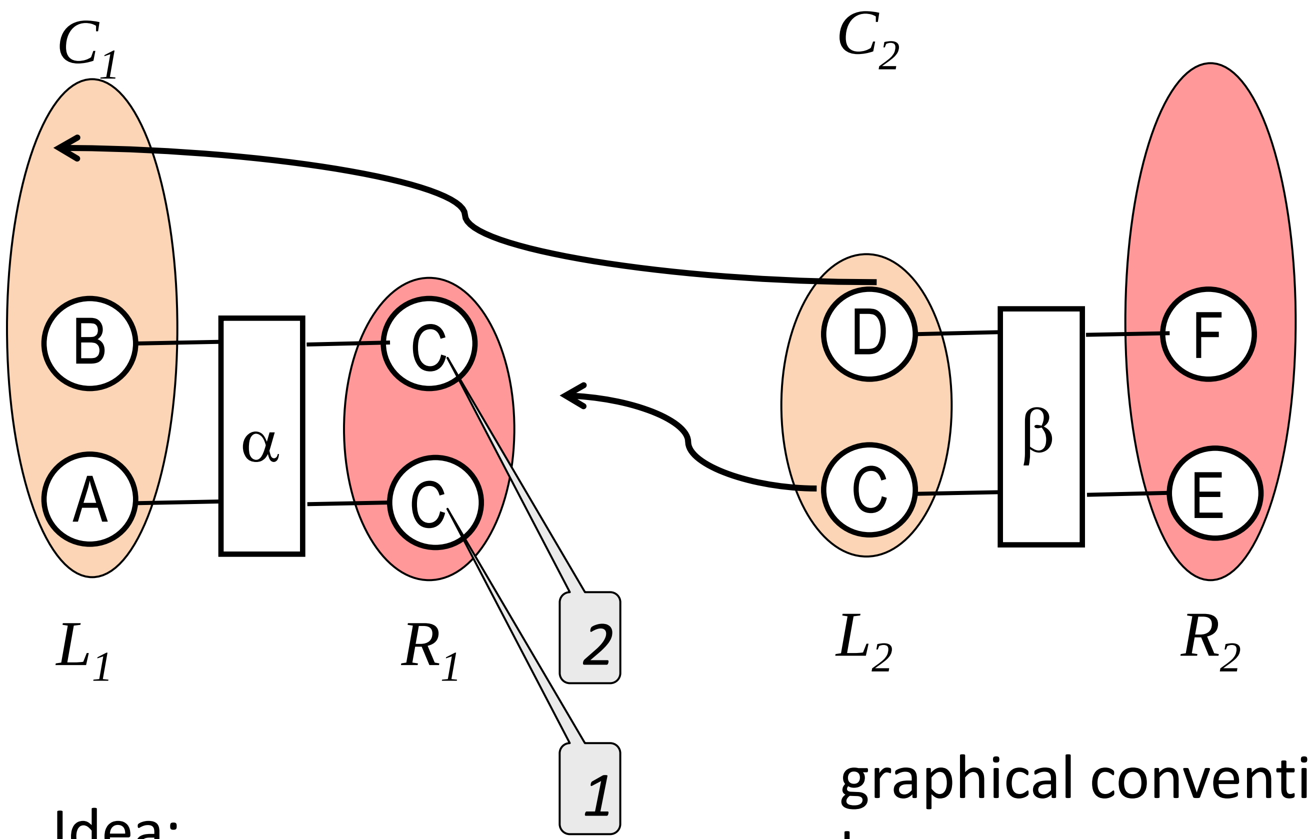
Port with multiple label



Two nodes of R_{12} are labelled alike!

You can not avoid this!

… what to do *here* ???

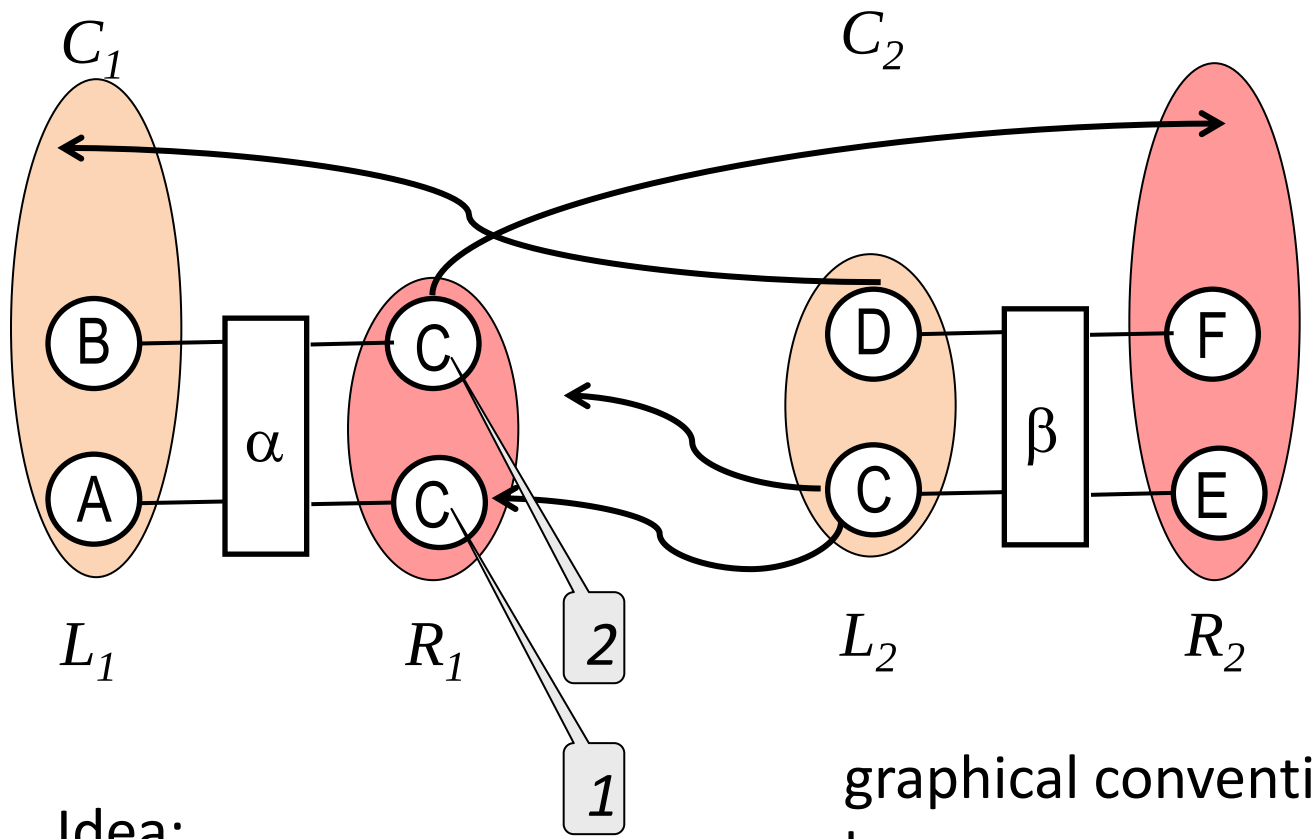


Idea:
n equally labelled nodes in one port are *indexed 1, … n* .

graphical convention:
lower < upper.

Glue
equally labelled and equally indexed nodes.

… what to do *here* ???

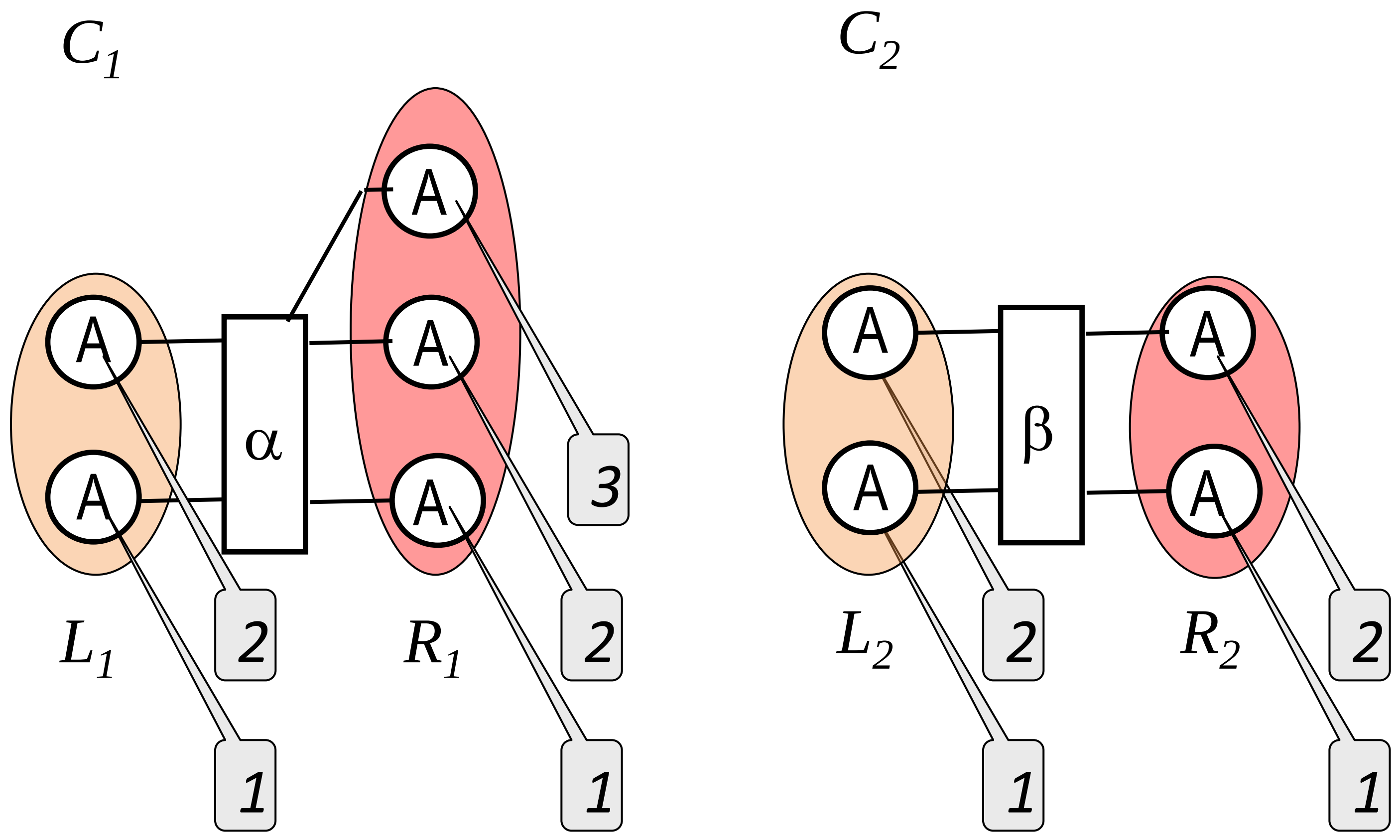


Idea:
Equally labelled nodes
in one port
are *ordered*.

graphical convention:
lower < upper.

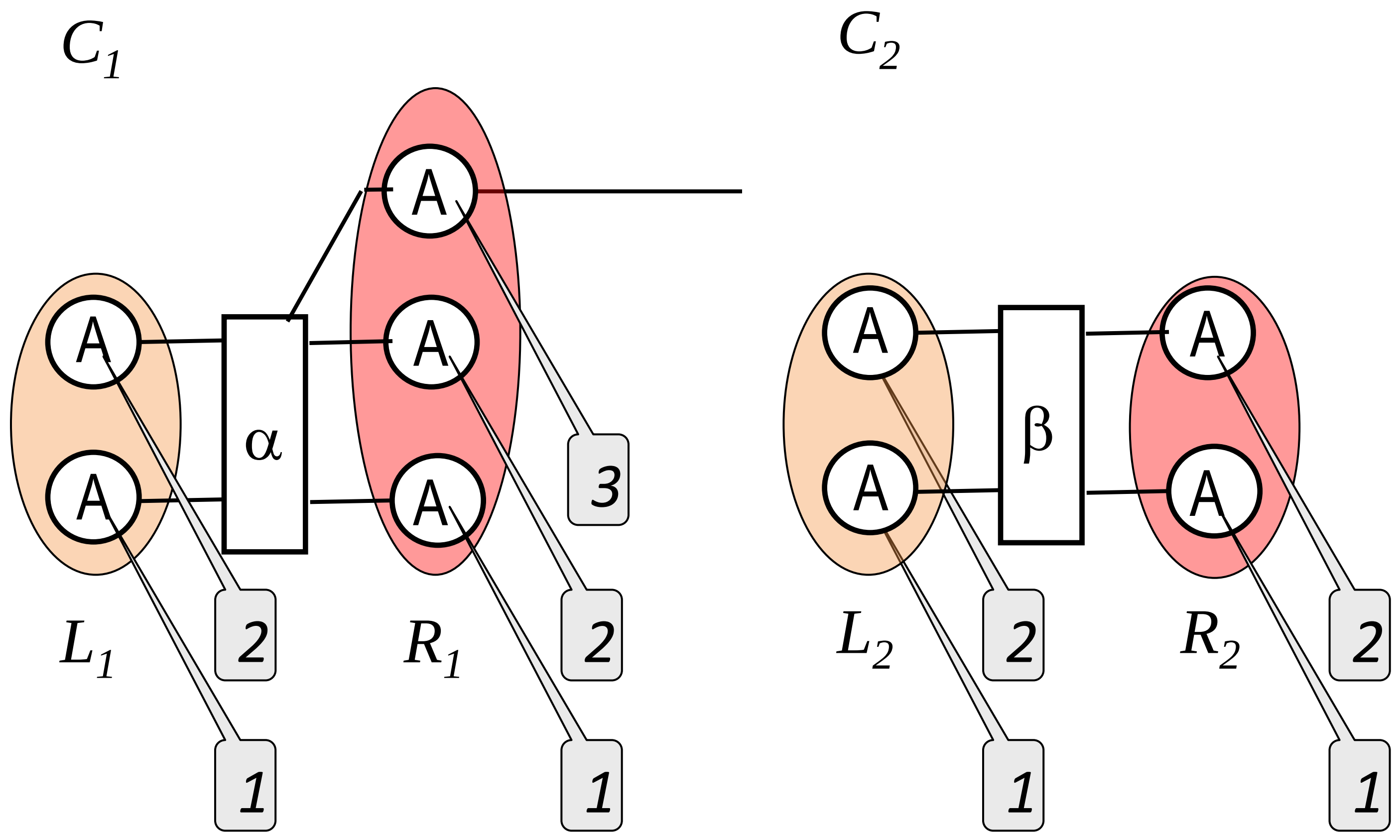
Glue
equally labelled nodes
both n-th in their order.

An extreme case



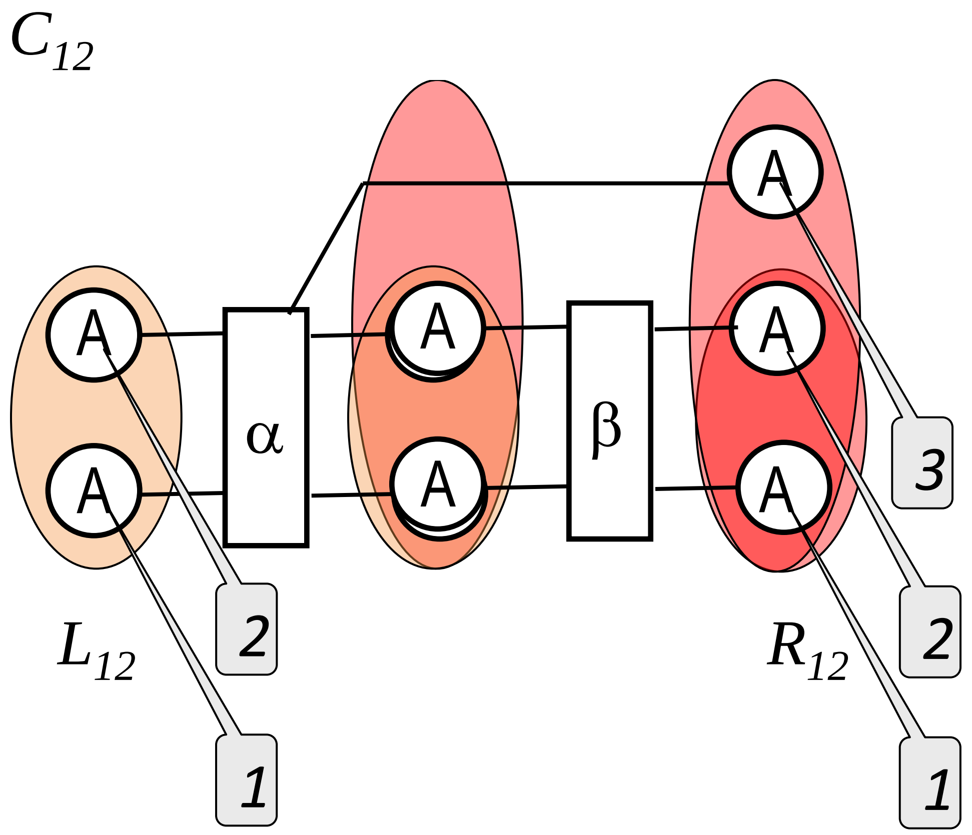
all labels alike.

An extreme case



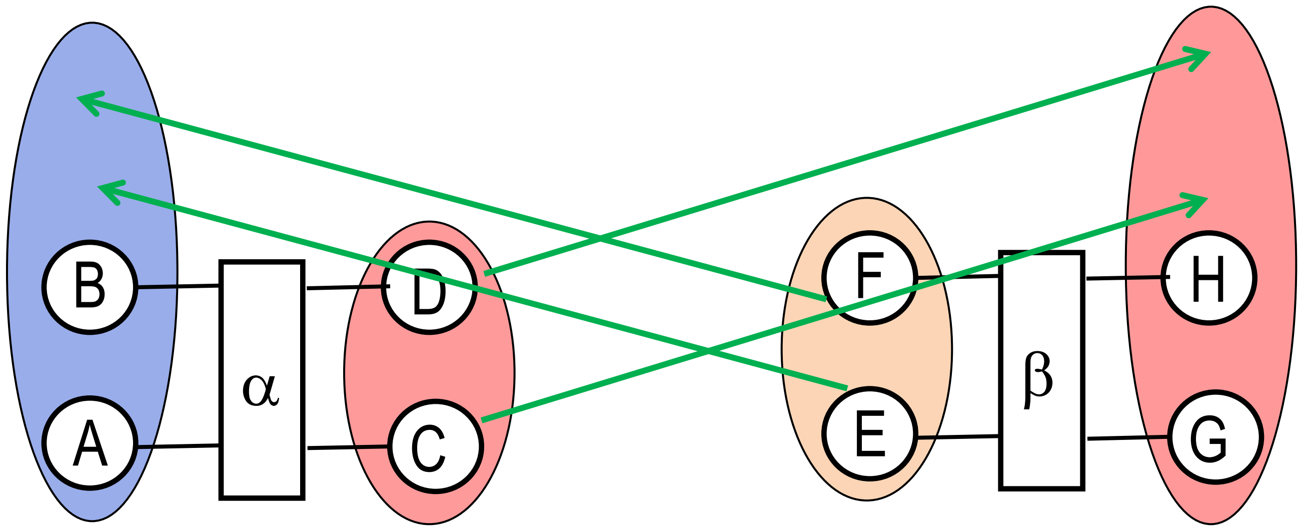
all labels alike.

An extreme case



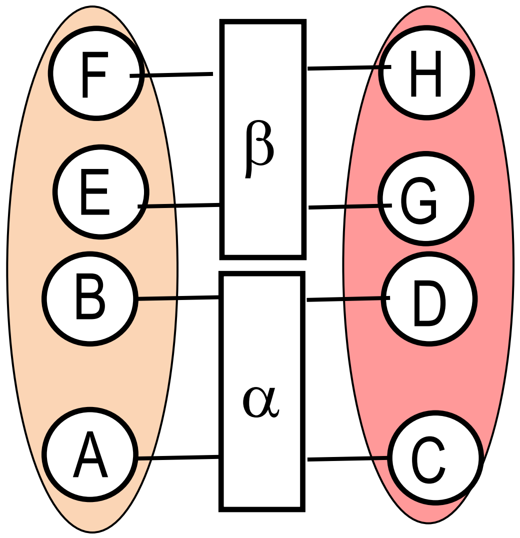
all labels alike.

… another extreme case



all labels different.

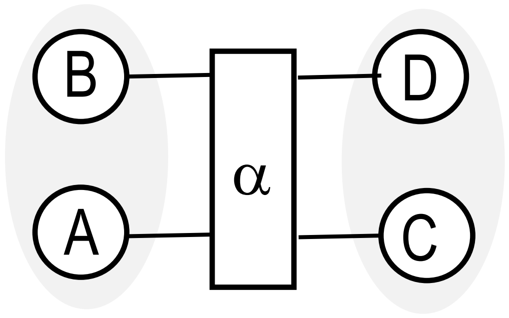
results in



1. Components: beautiful composition

A component has an *inner structure* and an *interface*.

Components are intended to be *composed* along their interface.



What we want:

a relevant class ***C*** of components such that

composition of components „ [illegible] “ is

- *total* i.e. [illegible] : ***C*** × ***C*** $\underline{\Omega}$ ***C*** **We got what we wanted …**

  A [illegible] *A* or *A* [illegible] *B* [illegible] *A* etc. are well defined,

- *parameter free*, i.e. no [illegible]$_i$ for any kind of parameter, *i*

  6 Lemmata

  13 Cases

- *associative*, i.e. (*A* [illegible] *B*) [illegible] *C* = *A* [illegible] (*B* [illegible] *C*) **took me three weeks …**

- flexible enough to cover many realistic applications

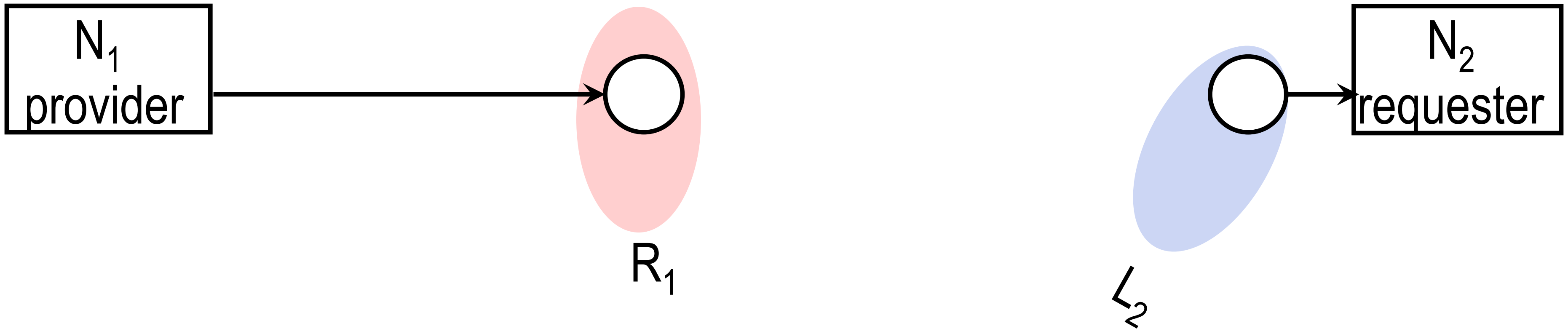
2. … we got even more:

technically:

not necessary L and R be disjoint!

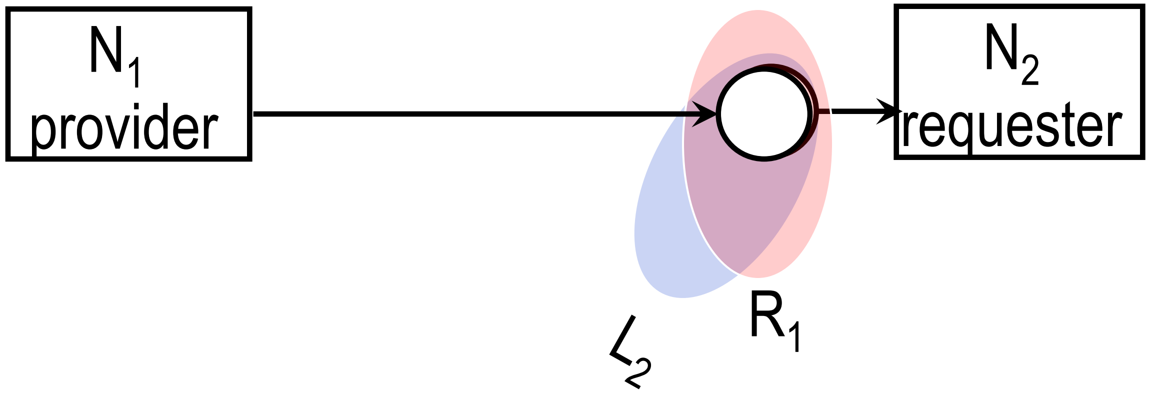
useful?

Exclusive requester



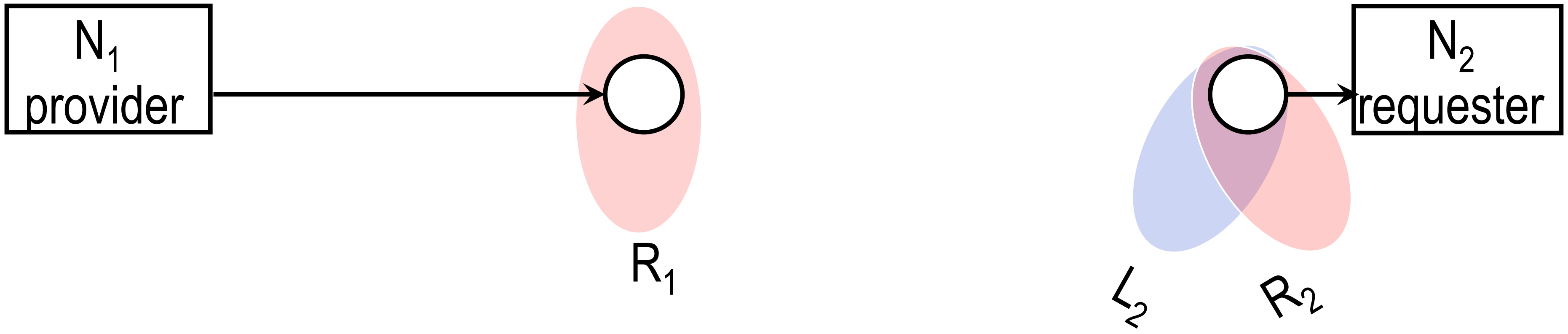
Exclusive requester

a variant:

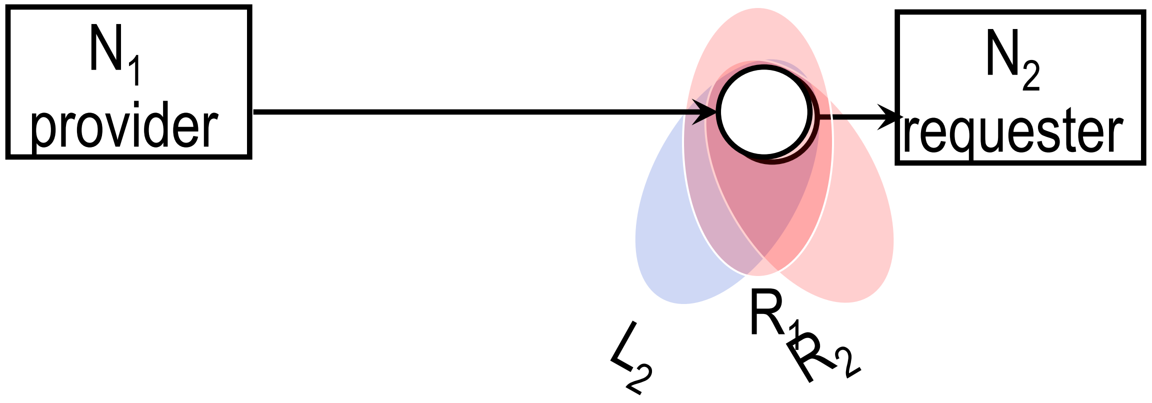


Sharing requester

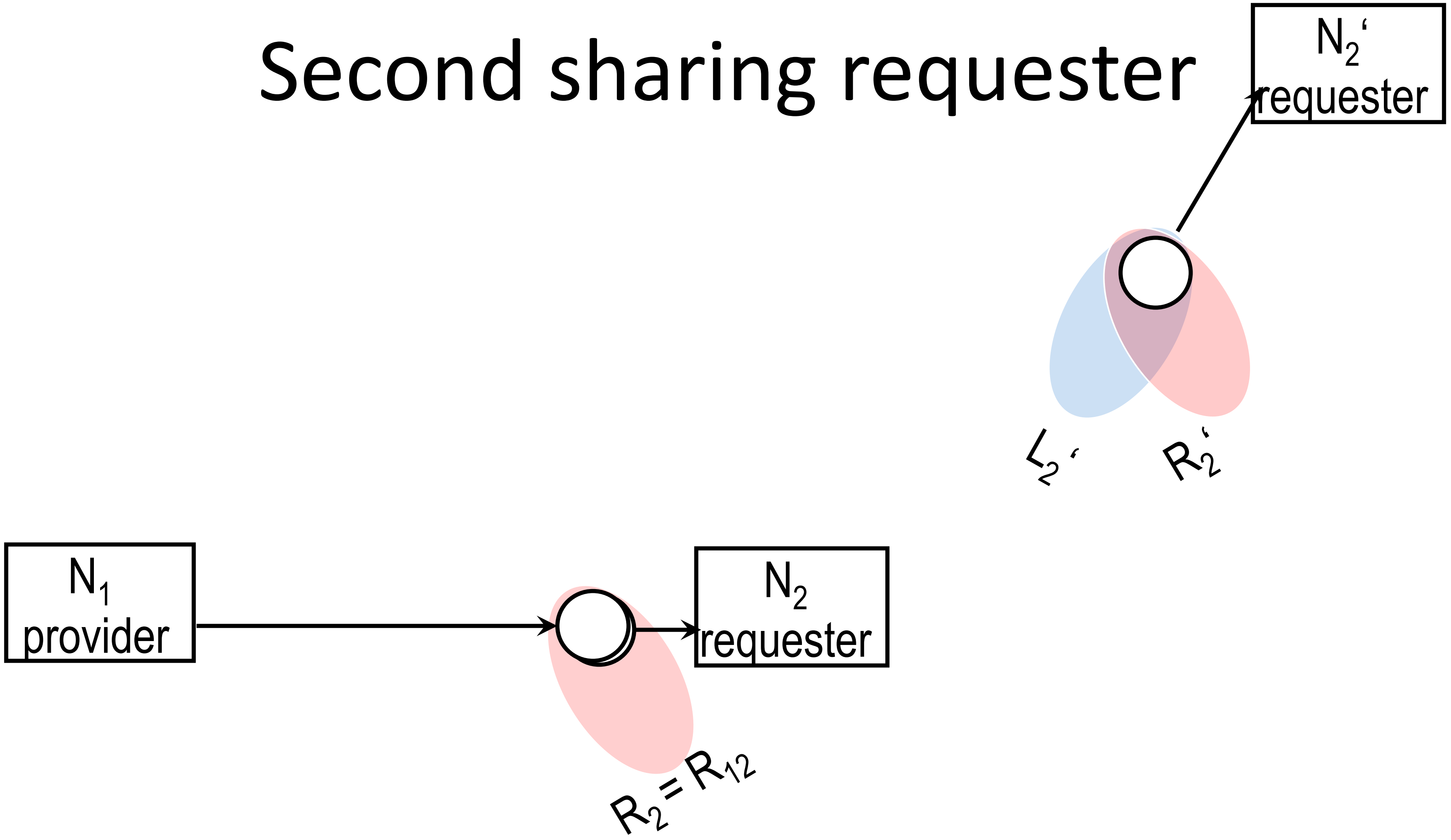
a variant:



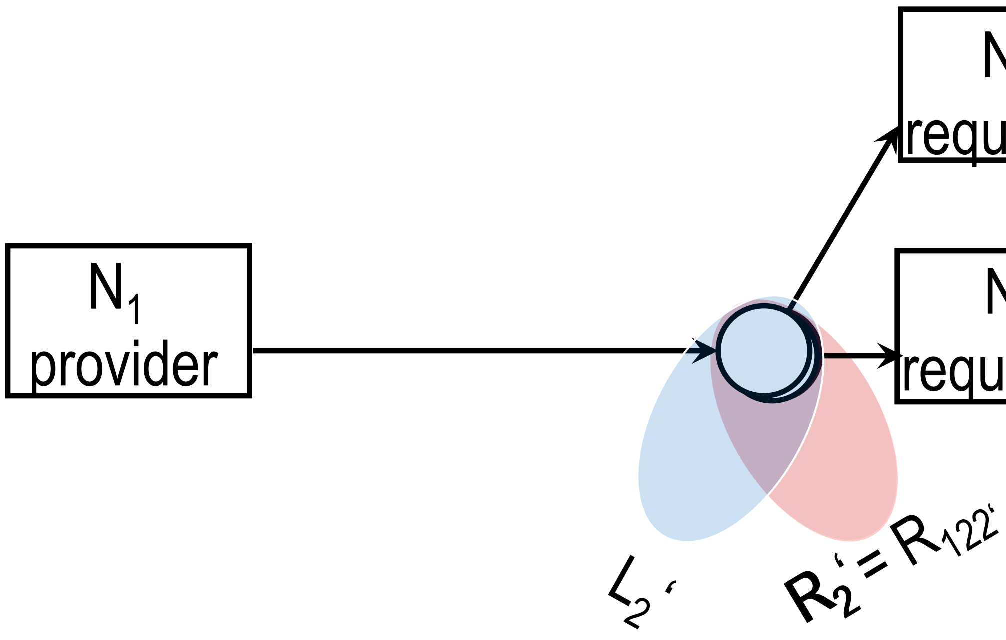
Sharing requester



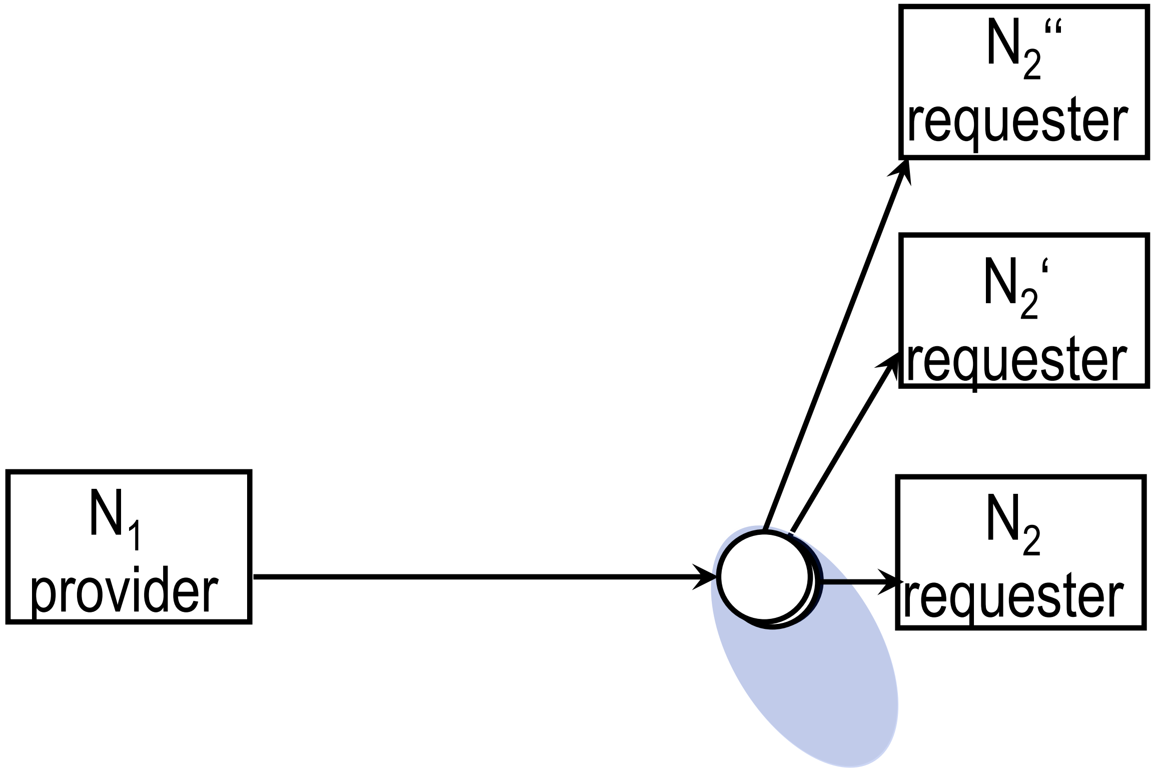
Second sharing requester



Second sharing requester

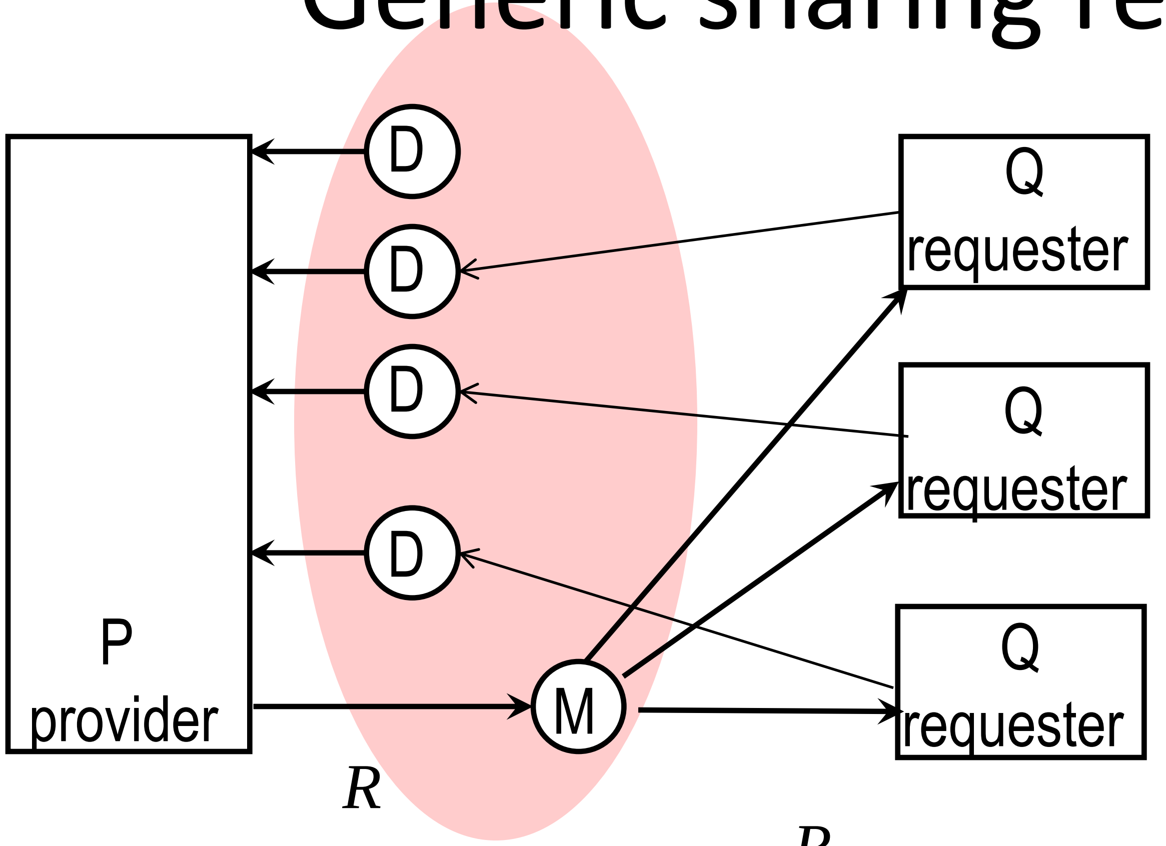


Third sharing requester



skip the primes:
N_1 👓 N_2 👓 N_2 👓 N_2

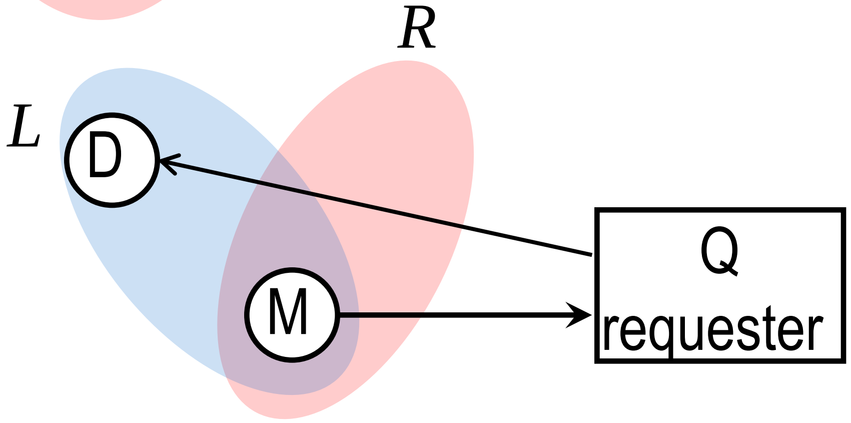
Generic sharing requesters



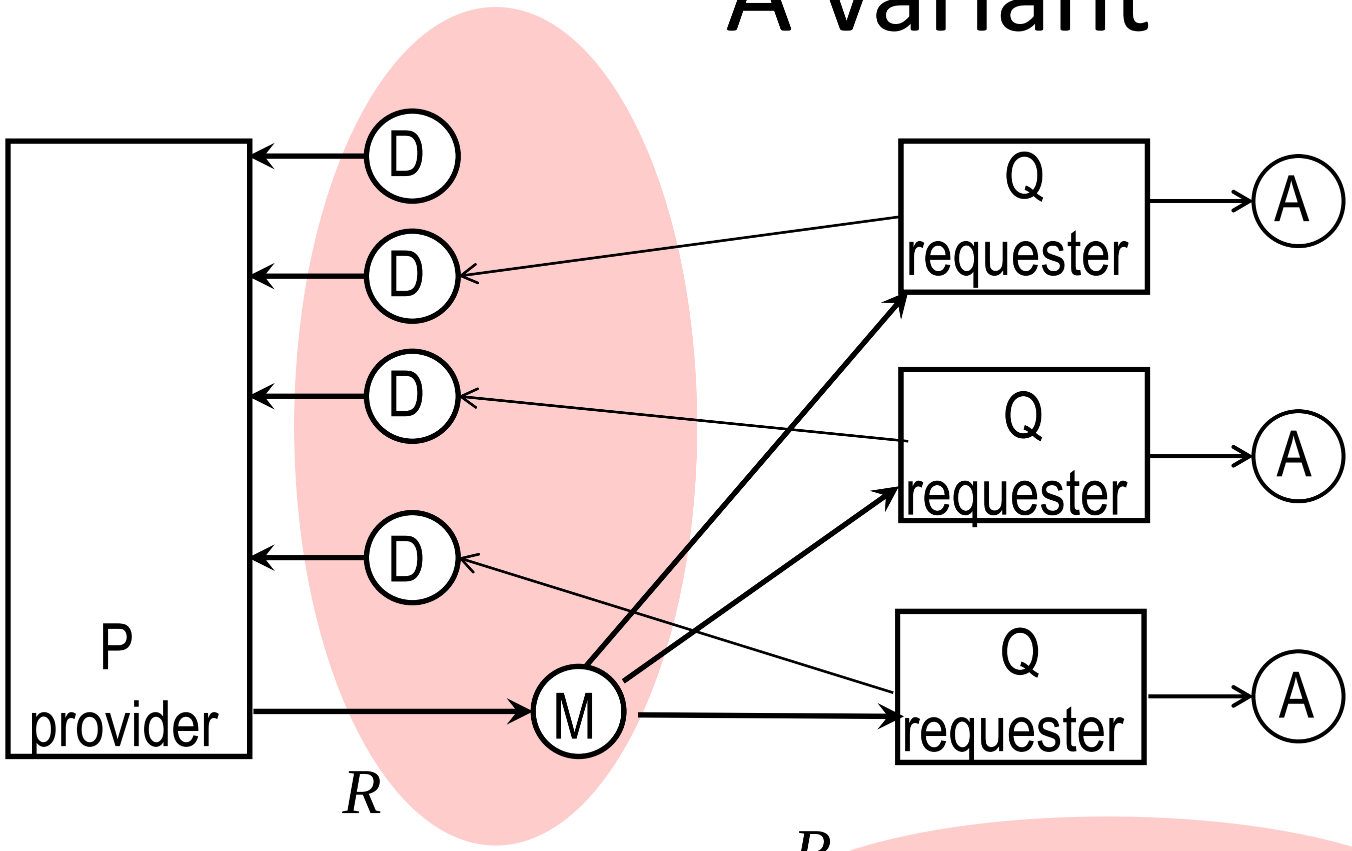
P 👓 Q
👓 Q👓
Q
P 👓 Q
👓 Q

P 👓 Q

generic
requester Q :



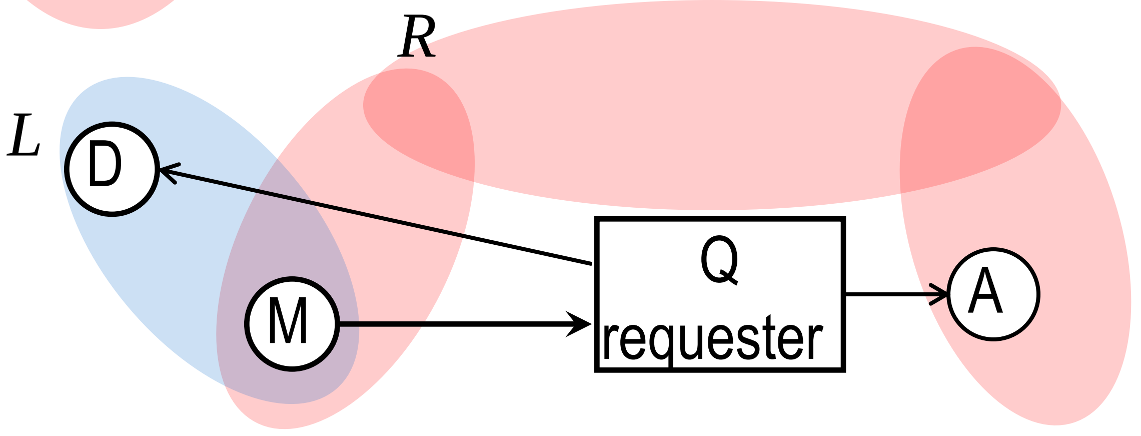
A variant



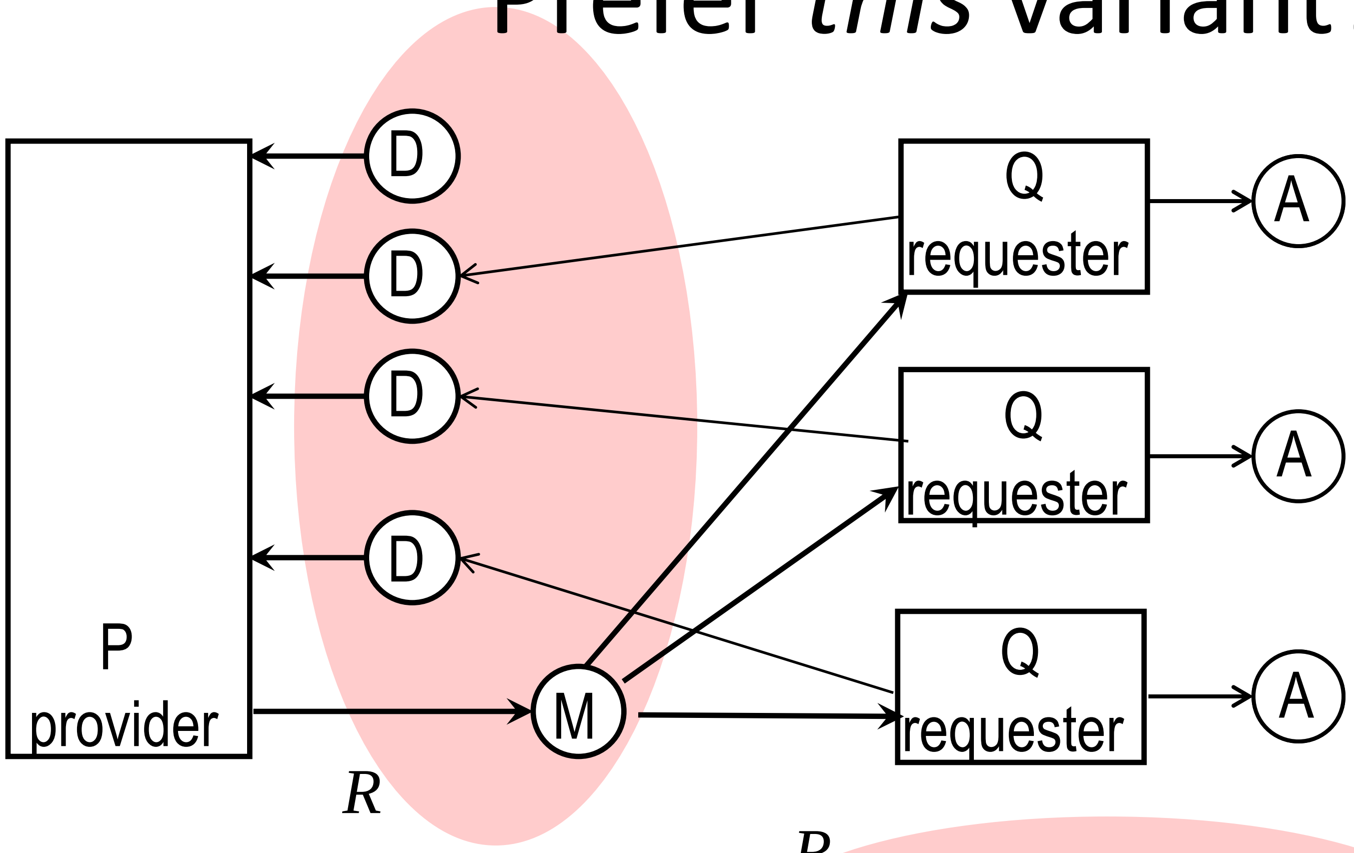
P 👓 Q
👓 Q👓
Q
P 👓 Q
👓 Q

P 👓 Q

generic
requester Q :



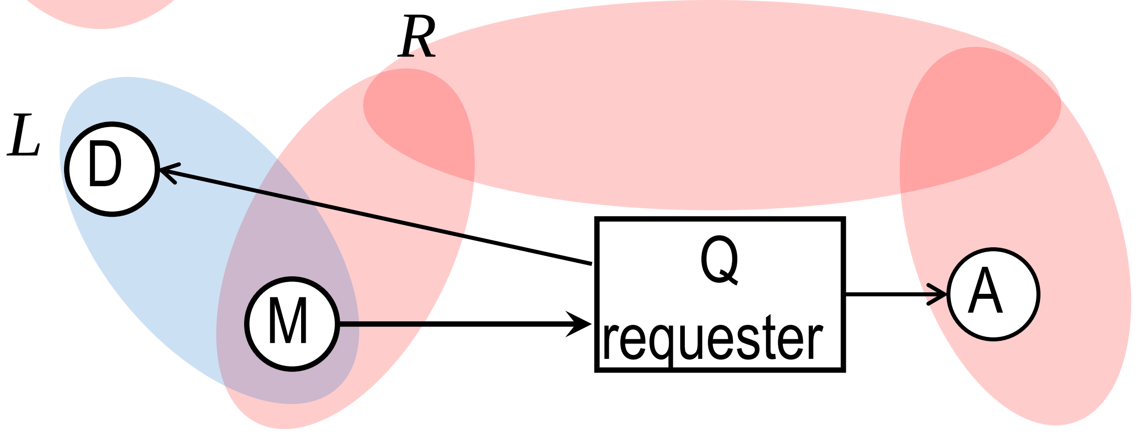
Prefer *this* variant?



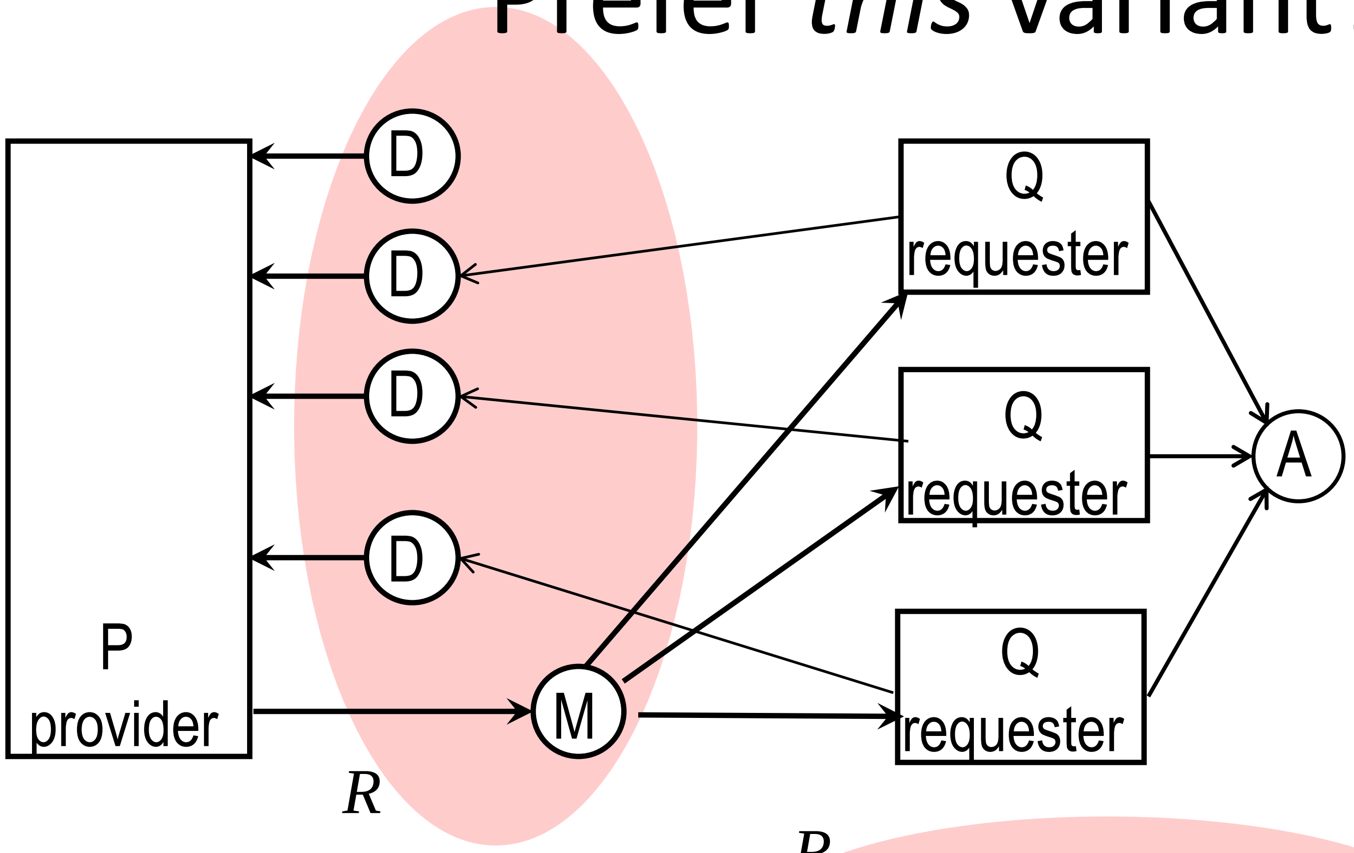
P 👓 Q
👓 Q👓
Q
P 👓 Q
👓 Q

P 👓 Q

generic
requester Q :



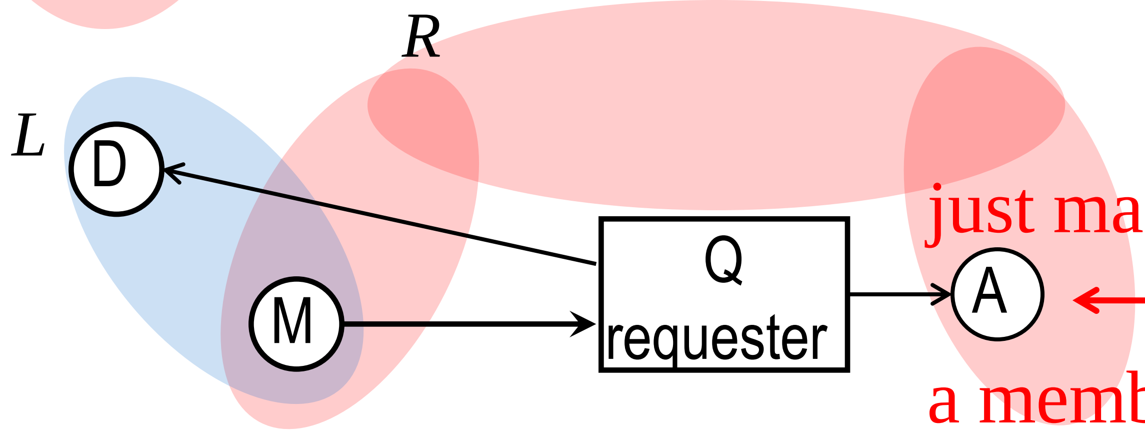
Prefer *this* variant?



P 👓 Q
👓 Q👓
Q
P 👓 Q
👓 Q

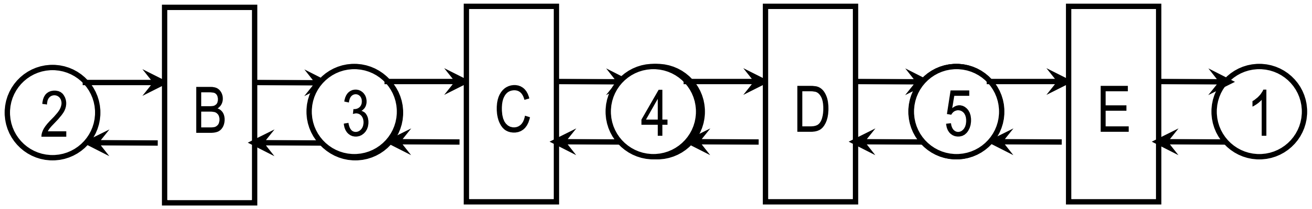
P 👓 Q

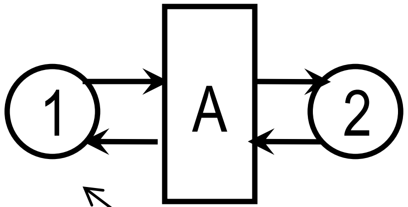
generic
requester Q :



just make A a member of *L*

Cyclic composition: The philosophers





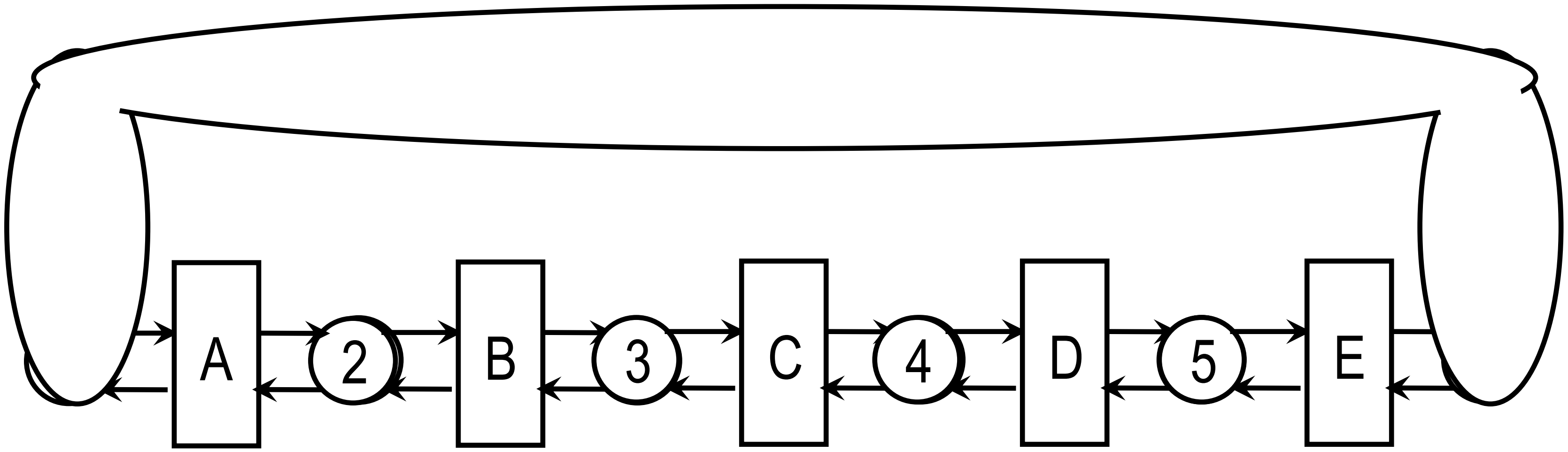
This is A👓 B 👓 C👓 D👓 E

The problem: How glue ?

Construct the *closure* (A👓 B 👓

C D E)c

Cyclic composition: The philosophers



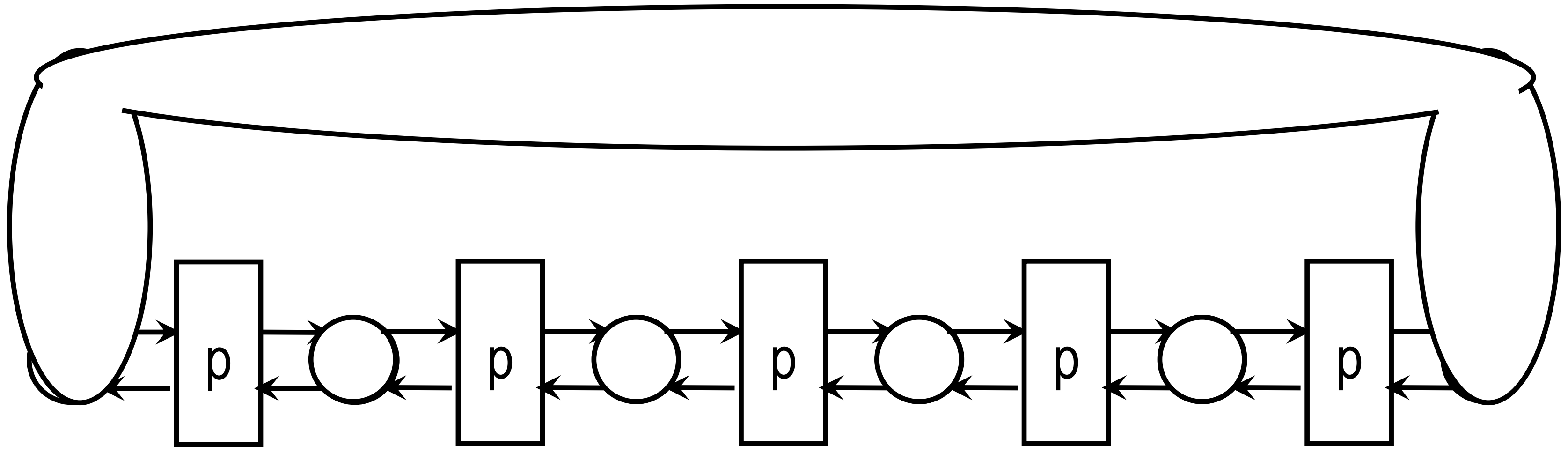
This is A👓 B 👓 C👓D👓 E

The problem: How glue ?

Construct the *closure* (A👓 B 👓

C D E)c

… with a generic philosopher



algebraic form: (p👓 p 👓 p👓
p👓 p)^c

… on your request

Don’t like labels at all? Do with ordered ports.

Prefer *one interface* instead of *two ports*? Take L = R.

However:
Order without labeling,
interface without two ports:
both not too expressive!

The algebra of services

(***C***, 👓 , ;) is a monoid. i.e. like (Σ^*, 👓 , ε)

Extend it to (***C***, 👓 , ; , ( )^c).

Study its algebraic laws!

Do formal language theory!

Build your systems accordingly!

Squeeze it all into tools!

Apply it!

This talk:

1. Prelude: The grand challenge

2. In Praise of Models

3. Tentative basic notions

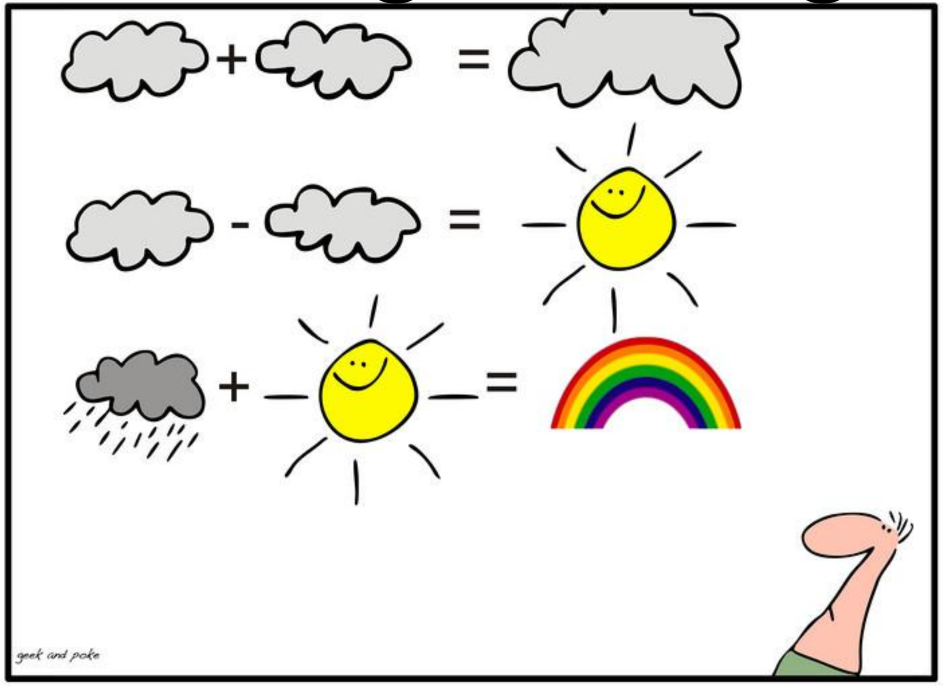
4. A notion of composition

SUMMERSOC 2015

Wednesday, July 1st, 2015

Service Orientation as a Paradigm of Programming

similar attempts: an algebra for cloud computing



CLOUD COMPUTING

the end

Humboldt Universität Informatik

Theory of Programming

Prof. Dr. W. Reisig

SOAMED

Notions	Problems	Tools
Services are *modeled.*	Formalization	tool chain
Services are *composed*. (R🖂S)		service-technology.org
A (composed) service may be *correct.*	Verification	now
Each service has a set of *partners*.	partner synthesis	by Marvin
U *adapts* R and S iff R🖂U🖂S is correct.	adapter synthesis	

Abstract

W. Reisig: Service Orientation as a Paradigm of Programming

Abstract

This contribution spans the broad spectrum from fundamental aspects of service modeling to tool-based analysis techniques of such models. We start with some fundamental considerations about the nature of service orientation as an architecture principle for software embedded systems. As a grand challenge of informatics we identify the missing theoretical foundation of modeling any kind of reactive systems, in particular service oriented computing.

In the second part we critically investigate the notion of models in general, and of services in particular. Compared to models in other sciences, we show that models in informatics frequently lack means to derive properties of a system from its model.

The third part suggests a couple of notions that may serve as a starting point for a systematic build-up of a theory of services.

In the fourth part we study in detail a particularly useful notion of composition of services.

Finally, we turn to applied aspects of service models: The tool chain as described in service-technology.org. A number of integrated tools supports the analysis of models of (Petri net based) services. Services represented in BPEL or BPMN can be analyzed via (software based) translation to Petri Nets.