Specification and analysis of communication protocols

Emilio Tuosto @ GSSI

20-24 April
and
4-7 May, 2026
Dipartimento di Informatica, Pisa

Logistics

Logistics

- Click or scan  for the web page; timetable

April 2026	May 2026
20 : 14.00-16.00	4 : 14.00-16.00
21 : 16.00-18.00	5 : 16.00-18.00
22 : 14.00-16.00	6 : 14.00-16.00
23 : 14.00-16.00	7 : 14.00-16.00

Logistics

- Click or scan  for the web page; timetable

April 2026	May 2026
20 : 14.00-16.00	4 : 14.00-16.00
21 : 16.00-18.00	5 : 16.00-18.00
22 : 14.00-16.00	6 : 14.00-16.00
23 : 14.00-16.00	7 : 14.00-16.00

- Zoom
  - Meeting ID: 838 7714 9605
  - Passcode: 867761

Logistics

- Click or scan for the web page; timetable

April 2026	May 2026
20 : 14.00-16.00	4 : 14.00-16.00
21 : 16.00-18.00	5 : 16.00-18.00
22 : 14.00-16.00	6 : 14.00-16.00
23 : 14.00-16.00	7 : 14.00-16.00

- Zoom
  - Meeting ID: 838 7714 9605
  - Passcode: 867761
- These lectures are about interactions ...so let's start interacting ☺, please
  - introduce yourself (name, research interests, known programming paradigms)
  - say what you expect from this course

Logistics

- Click or scan for the web page; timetable

April 2026	May 2026
20 : 14.00-16.00	4 : 14.00-16.00
21 : 16.00-18.00	5 : 16.00-18.00
22 : 14.00-16.00	6 : 14.00-16.00
23 : 14.00-16.00	7 : 14.00-16.00

- Zoom
  - Meeting ID: 838 7714 9605
  - Passcode: 867761
- These lectures are about interactions ...so let's start interacting ☺, please
  - introduce yourself (name, research interests, known programming paradigms)
  - say what you expect from this course
- Exam modality

Plan

Plan

Motivations

Plan

Motivations

Topics distributed coordination

- Choreographic models
- An unusual choreographic model
- Adding join patterns to actors
- A surprising (at least for me) application in economics

Plan

Motivations

Topics distributed coordination

- Choreographic models
- An unusual choreographic model
- Adding join patterns to actors
- A surprising (at least for me) application in economics

Tooling

– Motivations –

On program comprehension

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

Erlang

- Embodies the actor model [7, 1] ... back to '73!
- Message passing + functional programming
- Dynamic creation of threads
- Asynchrony by design! (FIFO buffers)

mailboxes in Erlang's jargon

All clear?

On program comprehension

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);
4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

Erlang

- Embodies the actor model [7, 1] ... back to '73!
- Message passing + functional programming
- Dynamic creation of threads
- Asynchrony by design! (FIFO buffers)

mailboxes in Erlang's jargon

All clear?

Will our program pass the test on lines 19-21?

On program comprehension

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);
4
4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

Erlang

- Embodies the actor model [7, 1] ... back to '73!
- Message passing + functional programming
- Dynamic creation of threads
- Asynchrony by design! (FIFO buffers)

mailboxes in Erlang's jargon

All clear?

Will our program pass the test on lines 19-21?

Arguably, it took some effort to get to the point.

On program comprehension

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);
4
4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

Erlang

- Embodies the actor model [7, 1] ... back to '73!
- Message passing + functional programming
- Dynamic creation of threads
- Asynchrony by design! (FIFO buffers)

mailboxes in Erlang's jargon

All clear?

Will our program pass the test on lines 19-21?

Arguably, it took some effort to get to the point.

Can we get some help?

On program comprehension

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);
4
4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

Erlang

- Embodies the actor model [7, 1] ... back to '73!
- Message passing + functional programming
- Dynamic creation of threads
- Asynchrony by design! (FIFO buffers)

mailboxes in Erlang's jargon

All clear?

Will our program pass the test on lines 19-21?

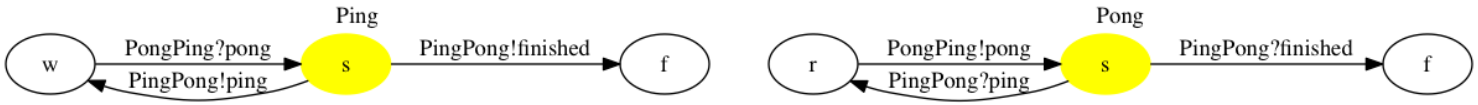
Arguably, it took some effort to get to the point.

Can we get some help?

Let's try something!

Friendlier representations with (formal) models

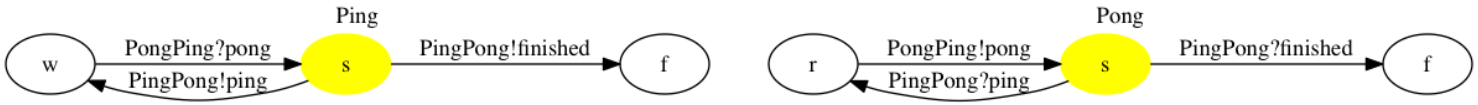
Local behaviour: communicating machines [2]



CFSMs: FIFO buffers as well

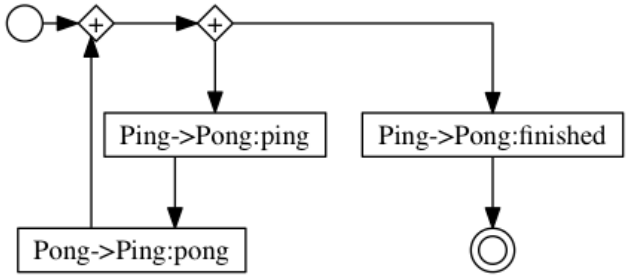
Friendlier representations with (formal) models

Local behaviour: communicating machines [2]



CFSMs: FIFO buffers as well

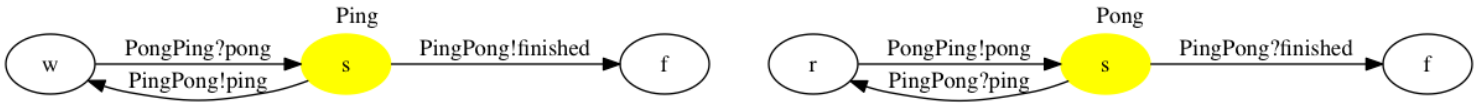
Choregraphy: global graph [3, 9]



...“synchronous” distributed workflow

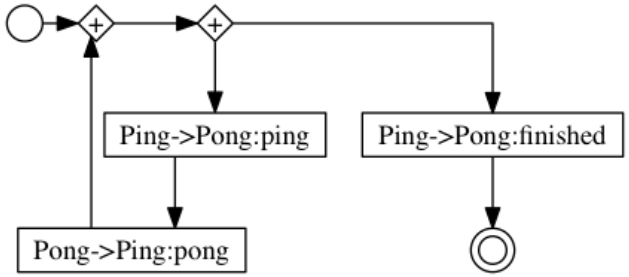
Friendlier representations with (formal) models

Local behaviour: communicating machines [2]



CFSMs: FIFO buffers as well

Choregraphy: global graph [3, 9]



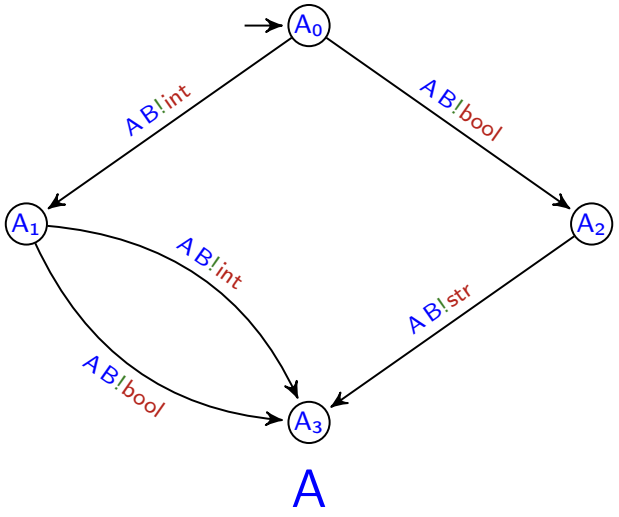
...“synchronous” distributed workflow

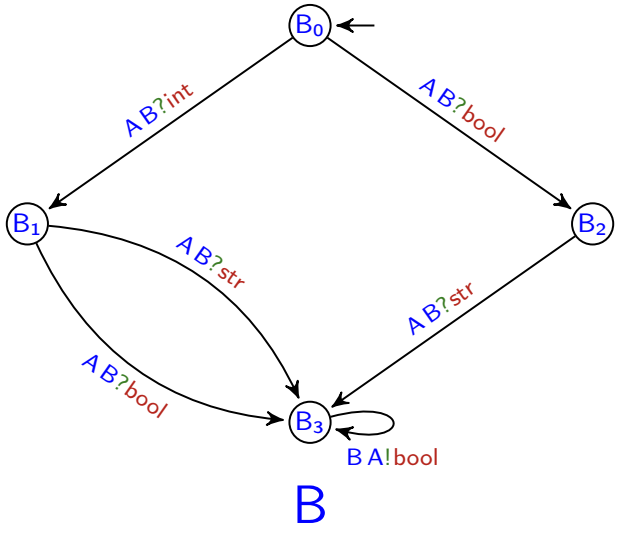
Research direction

How do we extract such models from code or documentation?

Our first formal model...informally

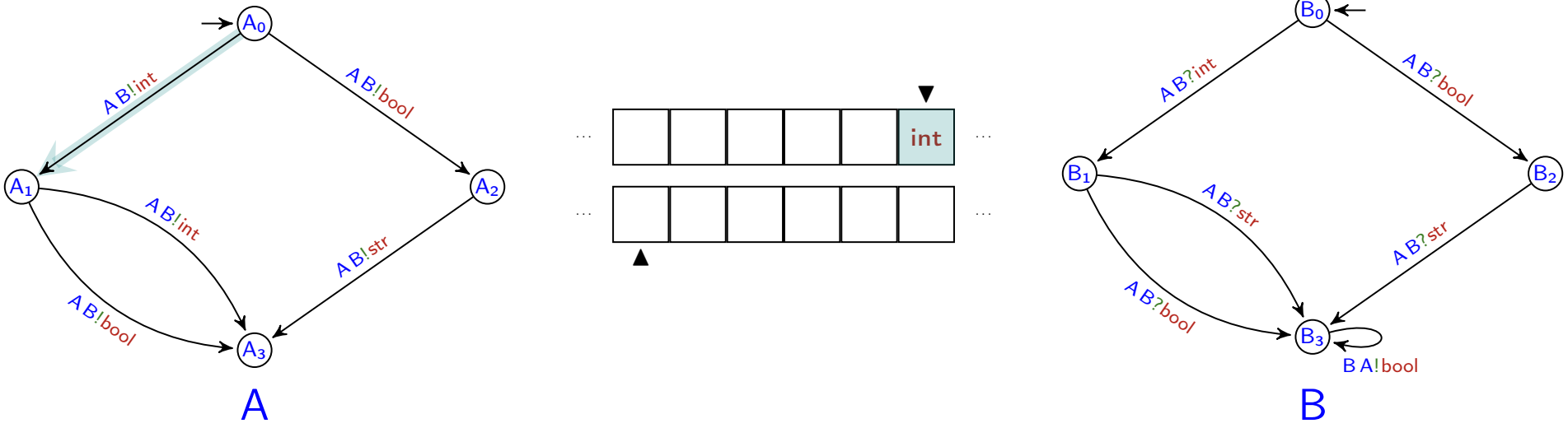
Communicating systems





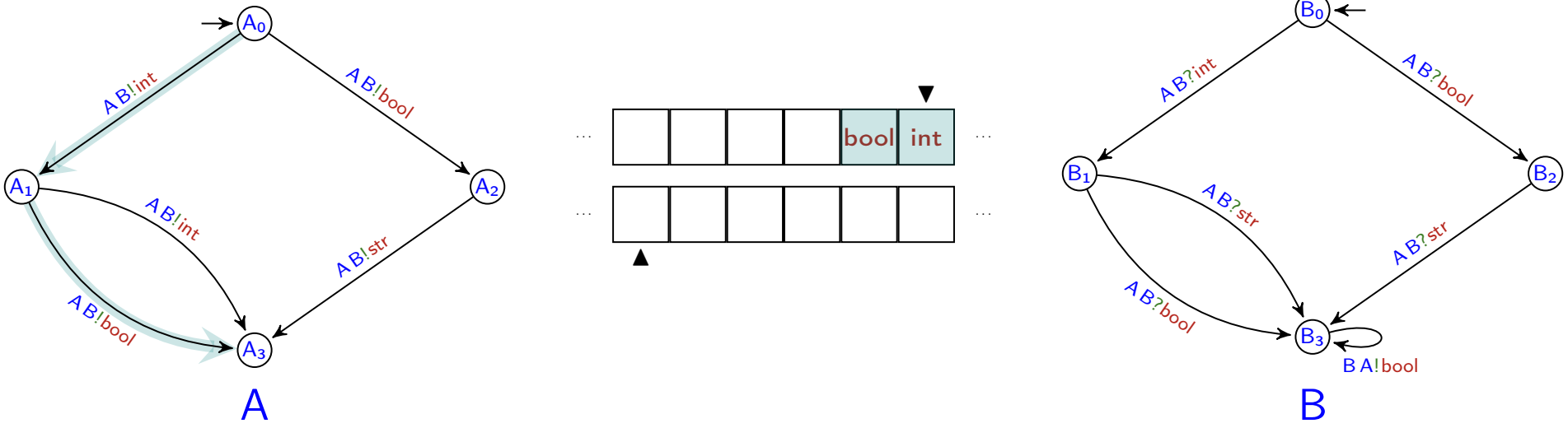
Our first formal model...informally

Communicating systems



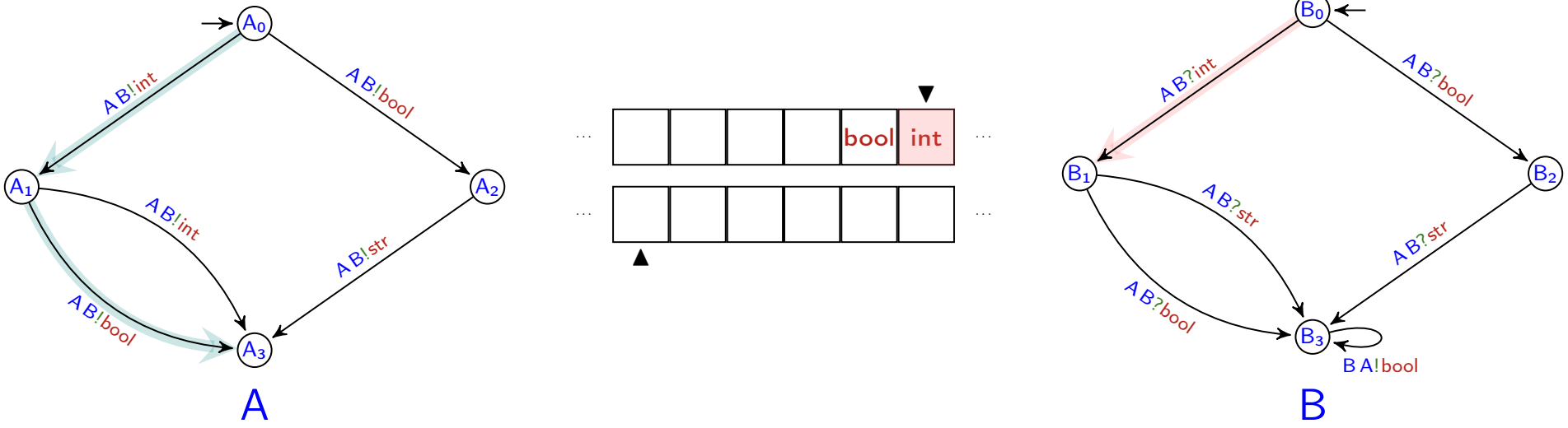
Our first formal model...informally

Communicating systems



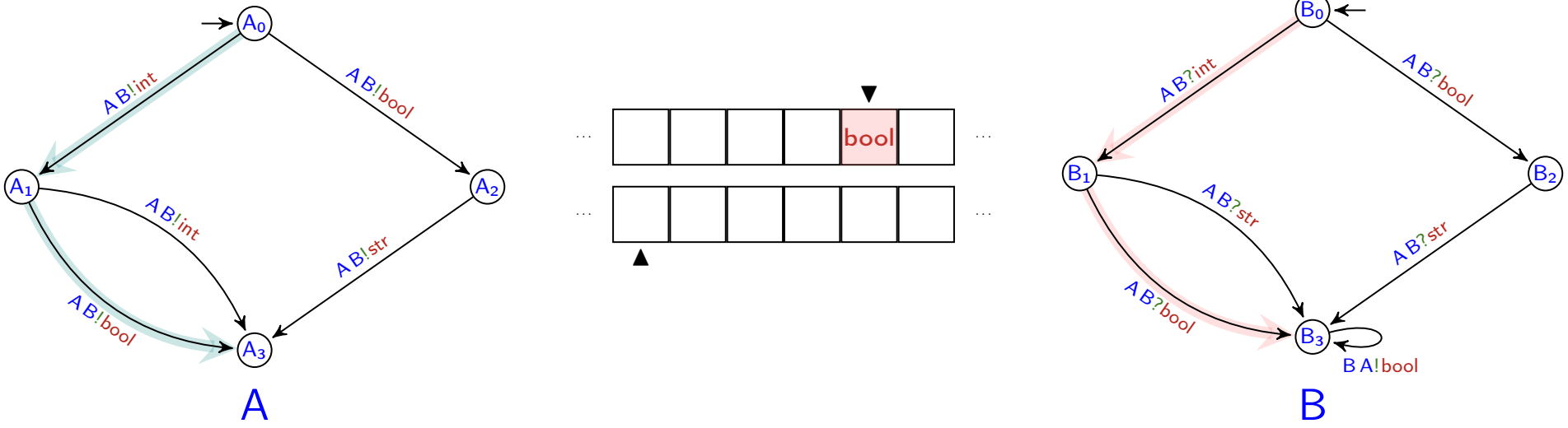
Our first formal model...informally

Communicating systems



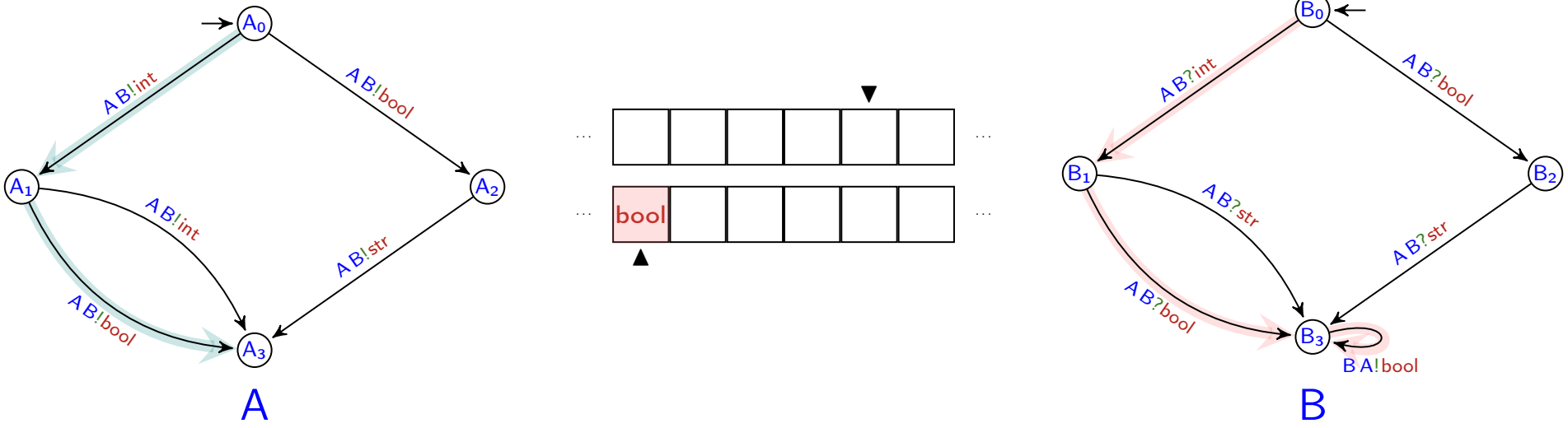
Our first formal model...informally

Communicating systems



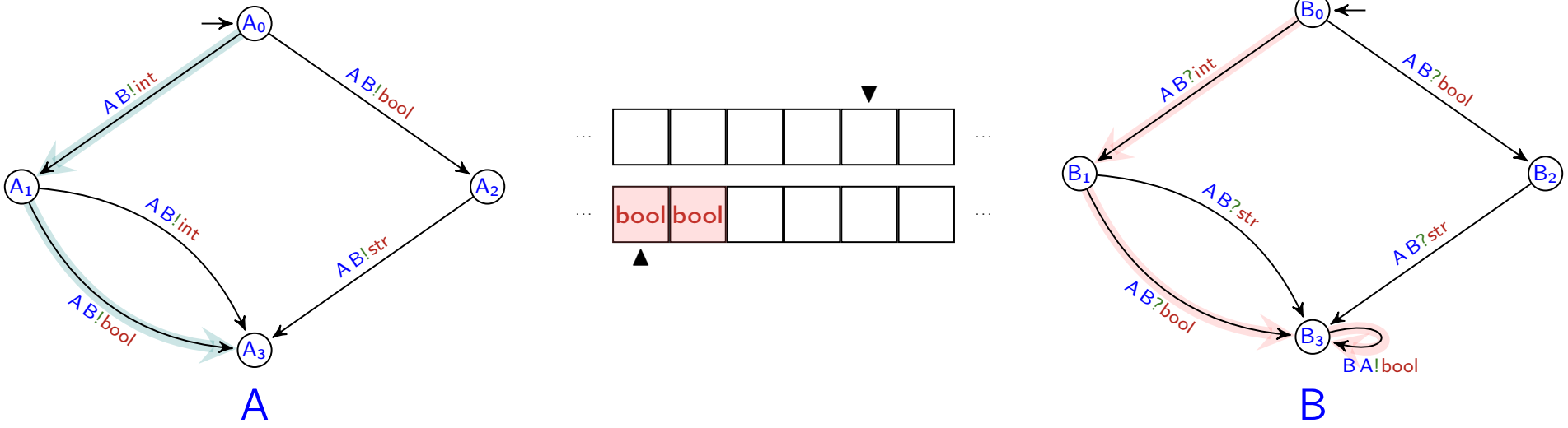
Our first formal model...informally

Communicating systems



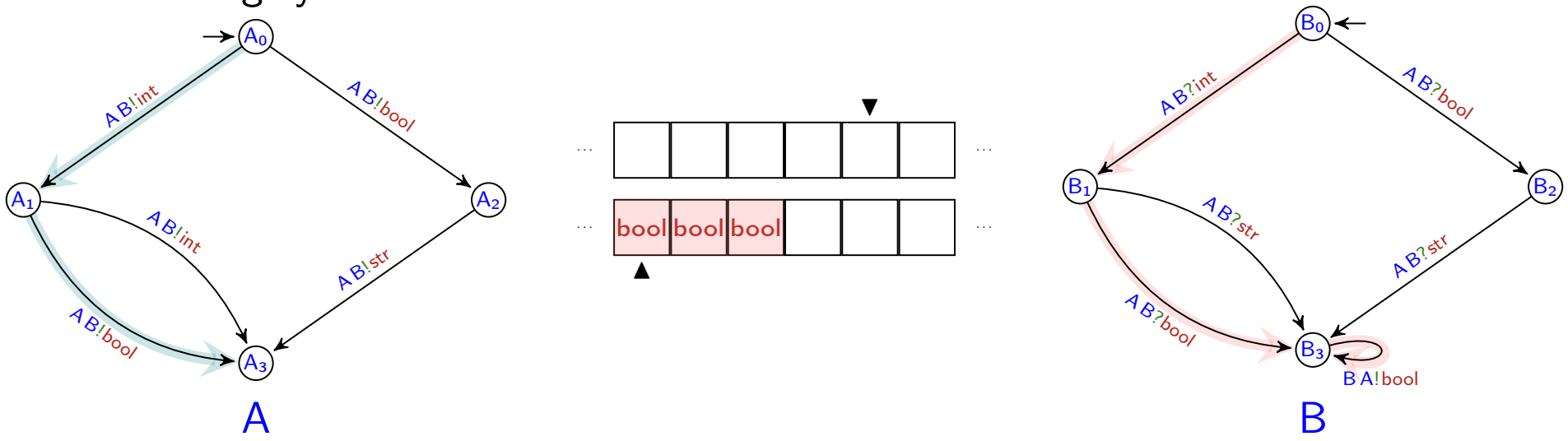
Our first formal model...informally

Communicating systems



Our first formal model...informally

Communicating systems



- a communicating finite-state machine (CFSM) is an FSA whose transitions are input/output actions executed by a single participant and whose states are all accepting
- a communicating system is a finite map assigning to a participant A a CFSM executing communications of A

On program correctness

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

So what?

On program correctness

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

So what?

- Now we have a model of the behaviour

On program correctness

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

So what?

- Now we have a model of the behaviour
- Let's use it to reason on correctness

On program correctness

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

So what?

- Now we have a model of the behaviour
- Let's use it to reason on correctness

Q:

The test was successful. But is the program correct?

A:

On program correctness

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

So what?

- Now we have a model of the behaviour
- Let's use it to reason on correctness

Q:

The test was successful. But is the program correct?

A:

No!

On program correctness

```
1  ping(0, Pong_PID) ->
2    Pong_PID ! finished,
3    io:format("ping finished~n", []);

4  ping(N, Pong_PID) ->
5    Pong_PID ! {ping, self()},
6    receive
7      pong ->
8        io:format("Ping received pong~n", [])
9    end,
10   ping(N - 1, Pong_PID).
```

```
11 pong() ->
12   receive
13     finished ->
14       io:format("Pong finished~n", []);
15     {ping, Ping_PID} ->
16       io:format("Pong received ping~n", []),
17       Ping_PID ! pong,
18       pong().
```

```
19 start() ->
20   Pong_PID = spawn(example, pong, []),
21   spawn(example, ping, [3, Pong_PID]).
```

So what?

- Now we have a model of the behaviour
- Let's use it to reason on correctness

Q:

The test was successful. But is the program correct?

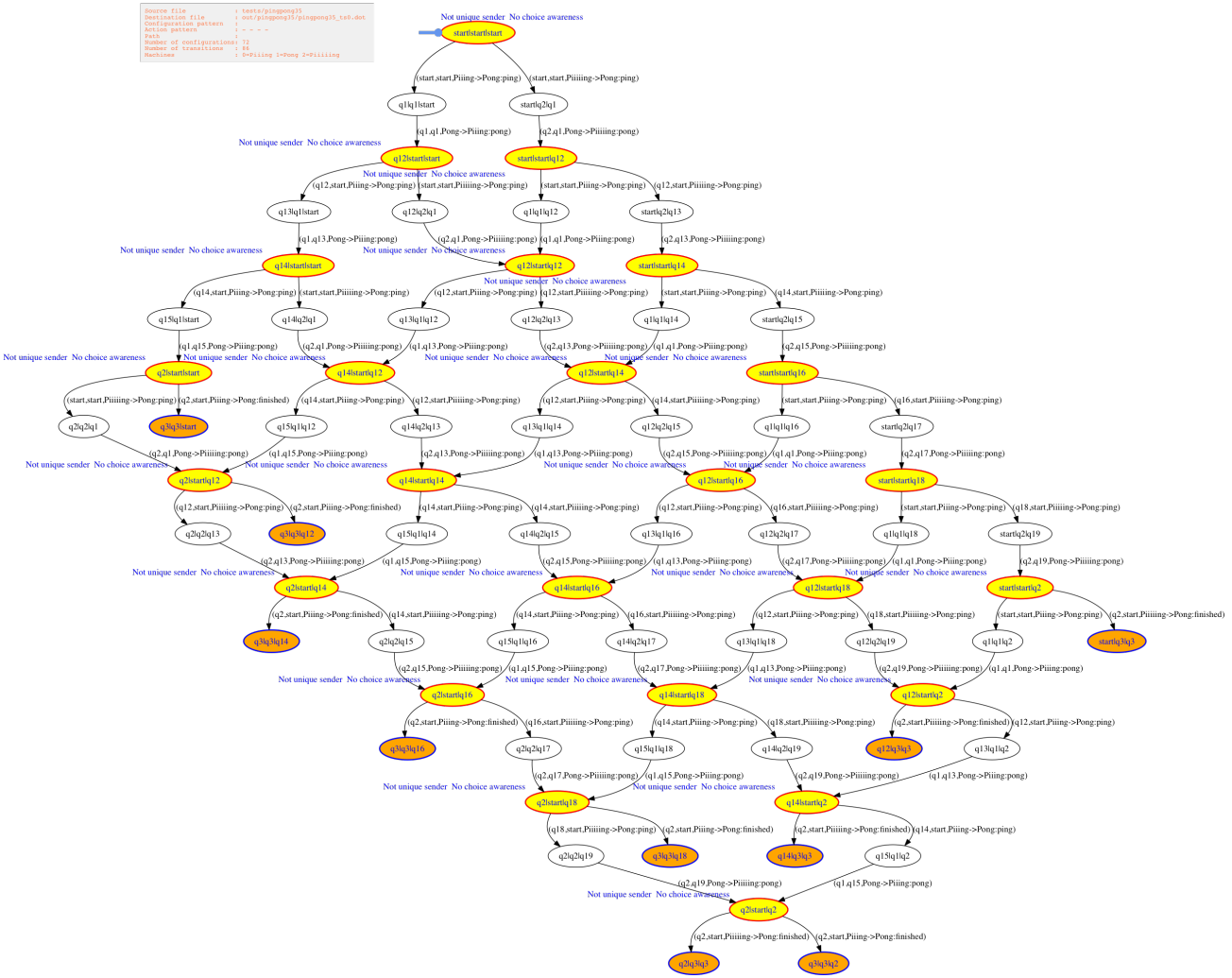
A:

No!

Exercise:

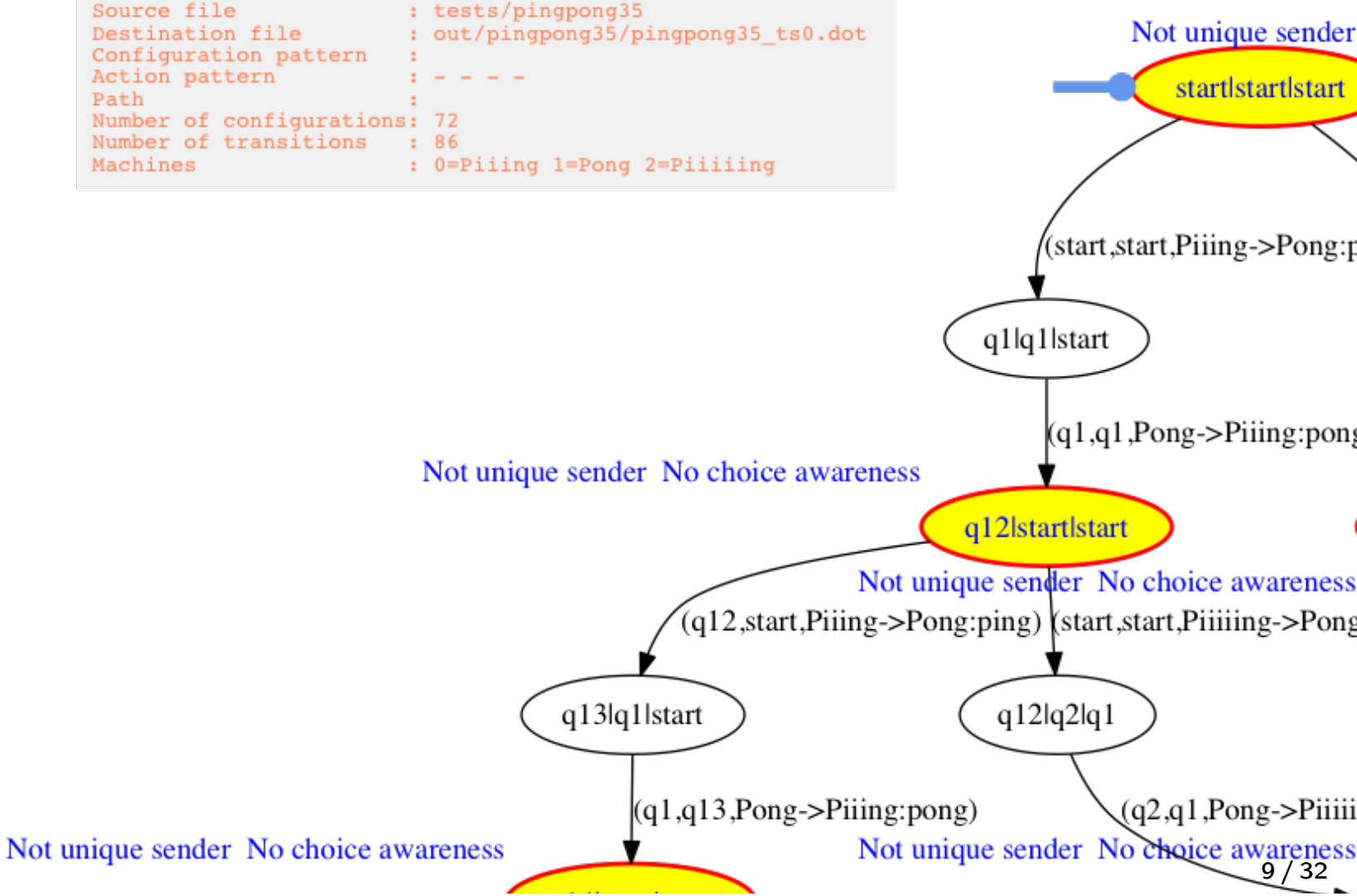
Find the bugs (there're at least 2!)

2 ping clients and 1 pong server!!!



2 ping clients and 1 pong server!!!

```
Source file              : tests/pingpong35
Destination file         : out/pingpong35/pingpong35_ts0.dot
Configuration pattern    :
Action pattern           : - - - -
Path                     :
Number of configurations: 72
Number of transitions    : 86
Machines                 : 0=Piiing 1=Pong 2=Piiiiing
```



Some reflections

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

What does 'software' actually mean?

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

What does 'software' actually mean?

My view:

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

What does 'software' actually mean?

My view:

- SW is not just 'code'

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

What does 'software' actually mean?

My view:

- SW is not just 'code'
- SW = code + docs + certificates

Some reflections

Software is an important asset nowadays in most advanced societies

- It's hardly believable that one would buy a product without proper guarantees
- Such kind of guarantees are often not required of software

What does 'software' actually mean?

My view:

- SW is not just 'code'
- SW = code + docs + certificates
  - code is formal!
  - the other addends should be formal too!

Sources of complexity

Why are such applications hard?

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

- components that dynamically
  - (dis)appears
  - engage in “conversations” (that is are reactive)

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

- components that dynamically
  - (dis)appears
  - engage in "conversations" (that is are reactive)
- dynamically reconfigurable their connections

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

- components that dynamically
  - (dis)appears
  - engage in "conversations" (that is are reactive)
- dynamically reconfigurable their connections
- can dynamically form "ensembles"

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

- components that dynamically
  - (dis)appears
  - engage in "conversations" (that is are reactive)
- dynamically reconfigurable their connections
- can dynamically form "ensembles"
- are often developed independently

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

- components that dynamically
  - (dis)appears
  - engage in "conversations" (that is are reactive)
- dynamically reconfigurable their connections
- can dynamically form "ensembles"
- are often developed independently
- failing elements
  - communications
  - connections
  - nodes

Sources of complexity

Why are such applications hard?

Because distributed applications are made of

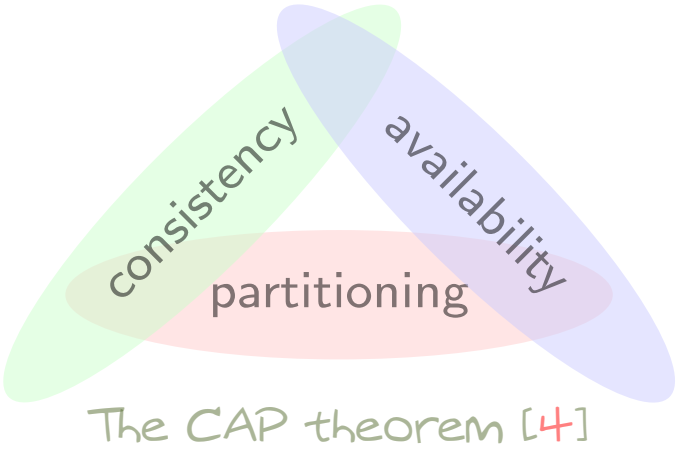
- components that dynamically
  - (dis)appears
  - engage in "conversations" (that is are reactive)
- dynamically reconfigurable their connections
- can dynamically form "ensembles"
- are often developed independently
- failing elements
  - communications
  - connections
  - nodes
- ...

Fundamental limitations

Distributed consensus

Distributed sharing
Security & coordination
Computer-assisted collaborative work
...

Fundamental limitations



The CAP theorem [4]

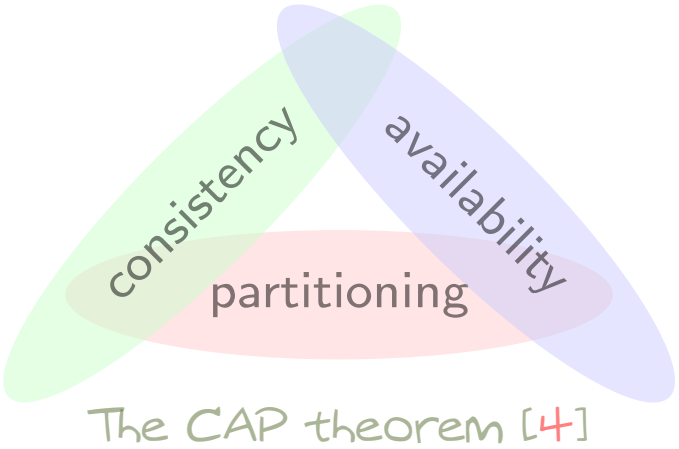
Distributed consensus

Distributed sharing
Security & coordination
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)

Fundamental limitations



The CAP theorem [4]

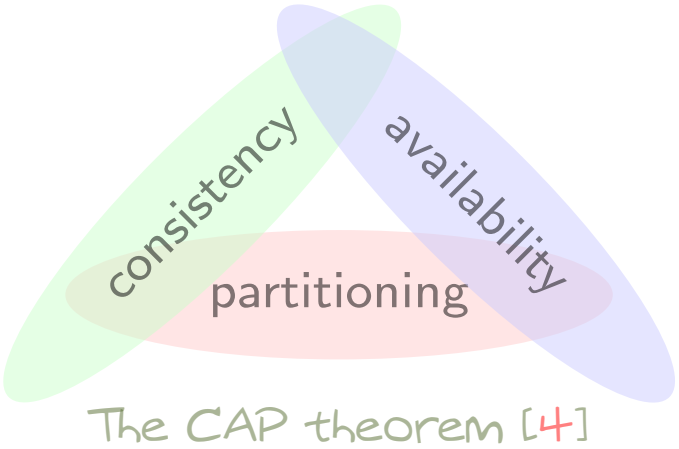
Distributed consensus

Distributed sharing
Security & coordination
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)
- Continuous consistency (with chosen tradeoffs)

Fundamental limitations



The CAP theorem [4]

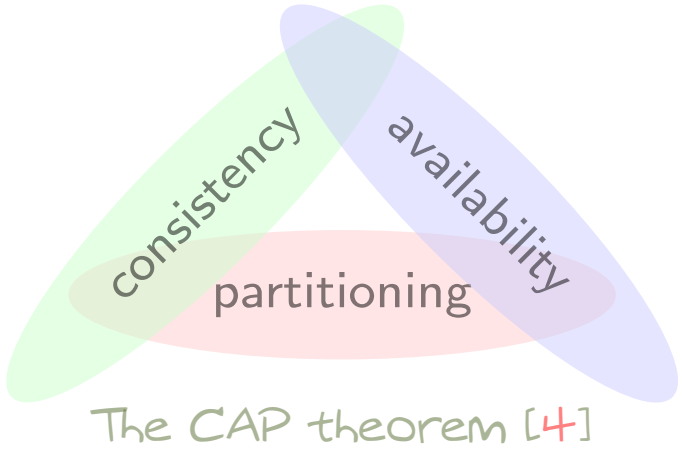
Distributed consensus

Distributed sharing
Security & coordination
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)
- Continuous consistency (with chosen tradeoffs)
- Commutative replicated data types (weakening consistency)

Fundamental limitations



The CAP theorem [4]

Distributed consensus

Distributed sharing
Security & coordination
Computer-assisted collaborative work
...

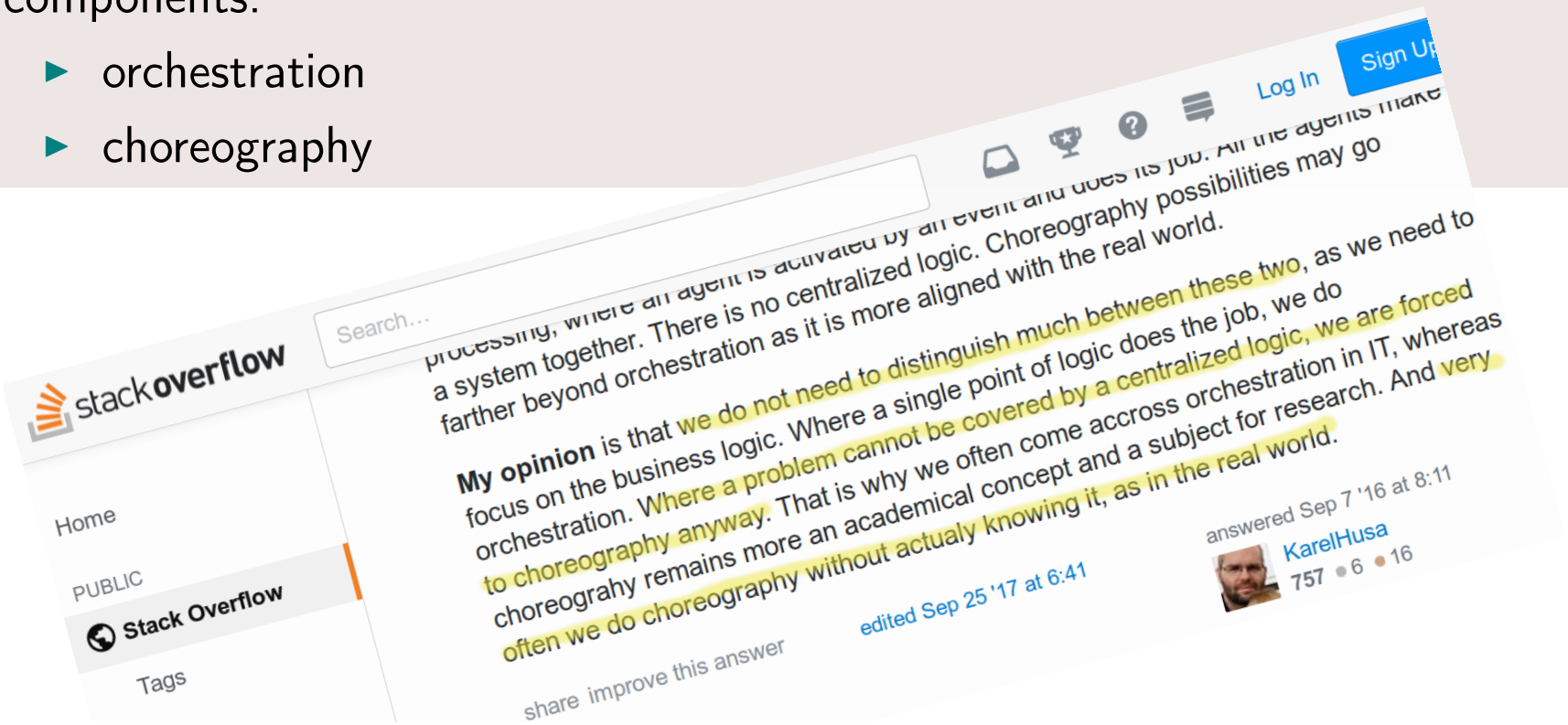
With some "solutions"

- Centralisation points (not that appealing)
- Continuous consistency (with chosen tradeoffs)
- Commutative replicated data types (weakening consistency)
- ...

Why are choreographic approaches appealing

Main design principles / programming styles for the coordination of distributed components:

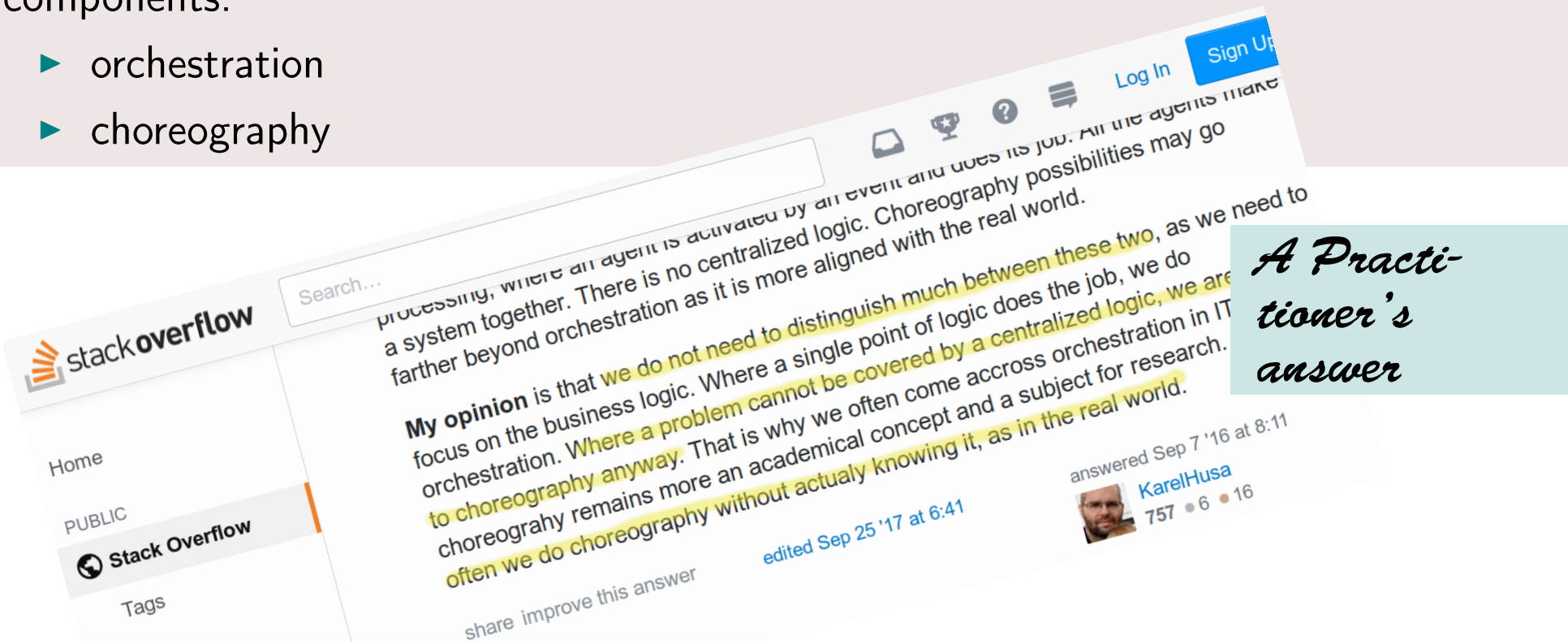
- orchestration
- choreography



Why are choreographic approaches appealing

Main design principles / programming styles for the coordination of distributed components:

- orchestration
- choreography



A Practitioner's answer

A rich research context

Many open problems

- Compositionality / Modularity
- Specs of Self-⋆ systems
- Data-driven systems
- Fault-tolerance
- Security
- ...

Many research teams

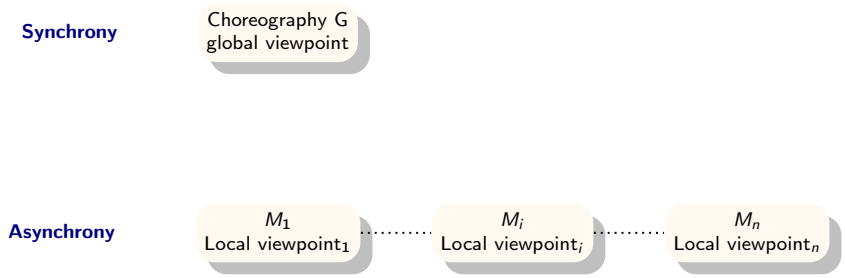
- Denmark
- Italy
- Germany
- Serbia
- Portugal
- Argentina
- Japan
- UK (including Scotland ☺)
- USA

– A first choreographic model –

"Top-down"

Quoting W3C [8]

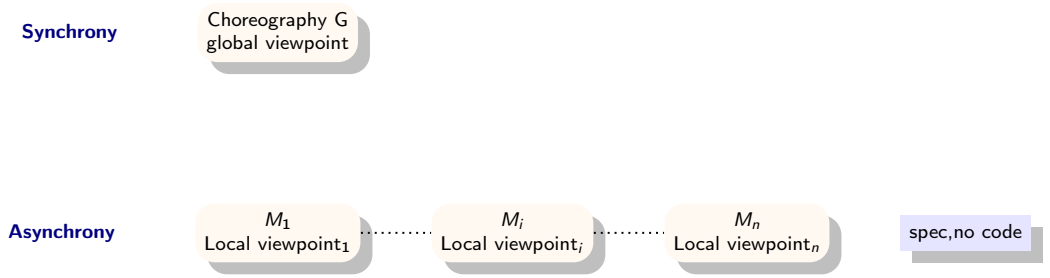
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting local systems [...]"



"Top-down"

Quoting W3C [8]

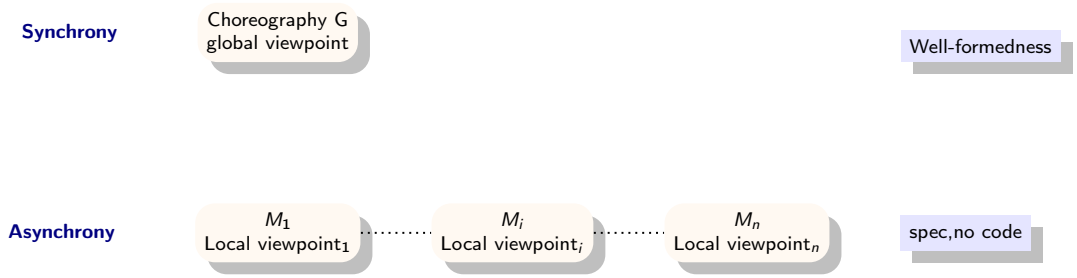
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting local systems [...]"



"Top-down"

Quoting W3C [8]

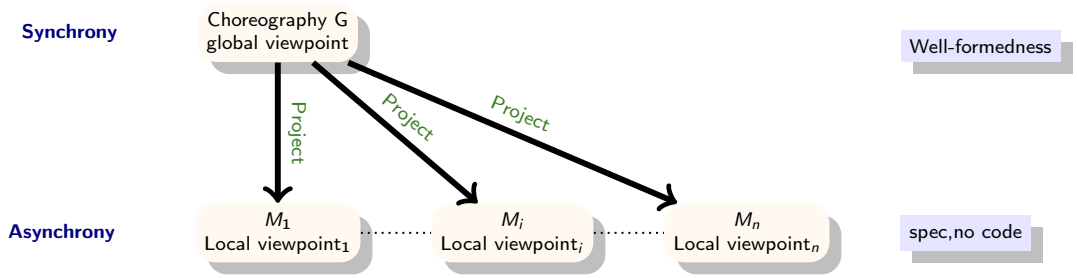
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting local systems [...]"



"Top-down"

Quoting W3C [8]

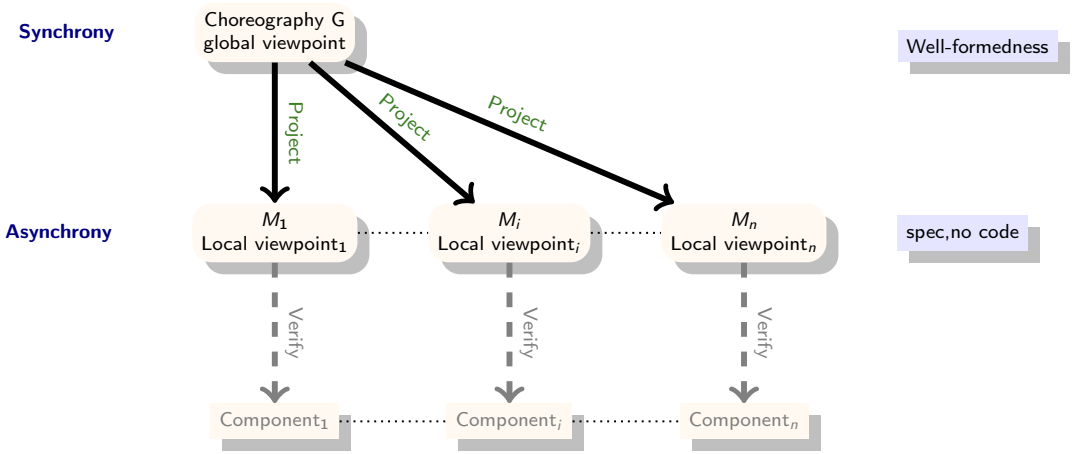
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting local systems [...]"



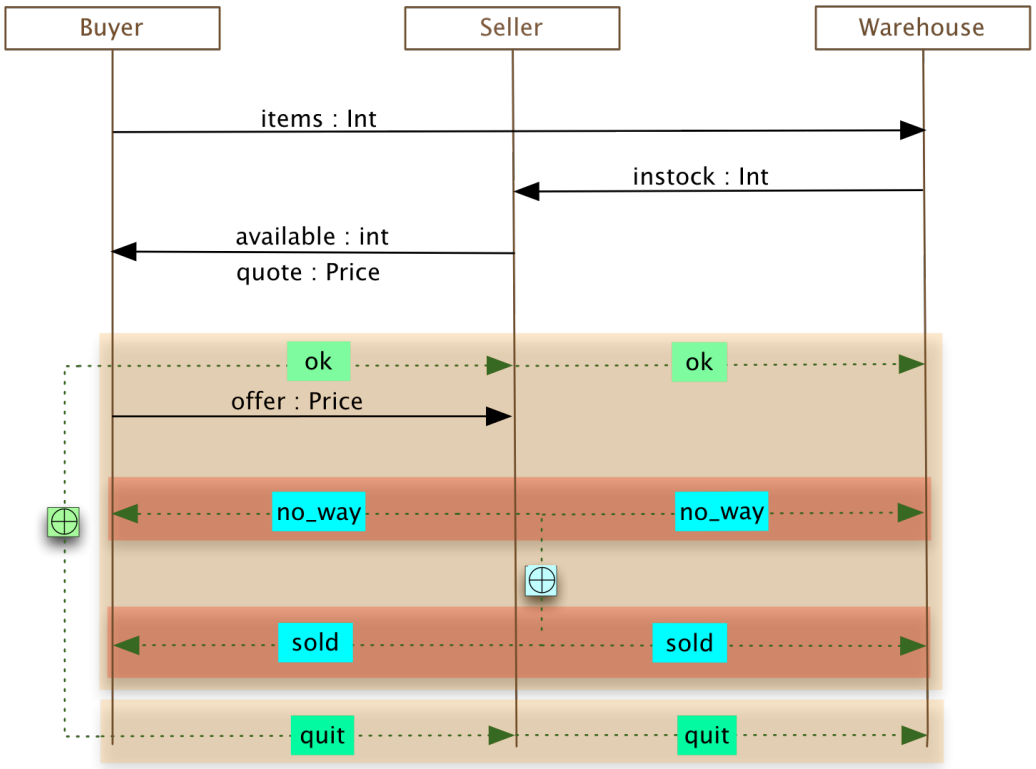
"Top-down"

Quoting W3C [8]

"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting local systems [...]"

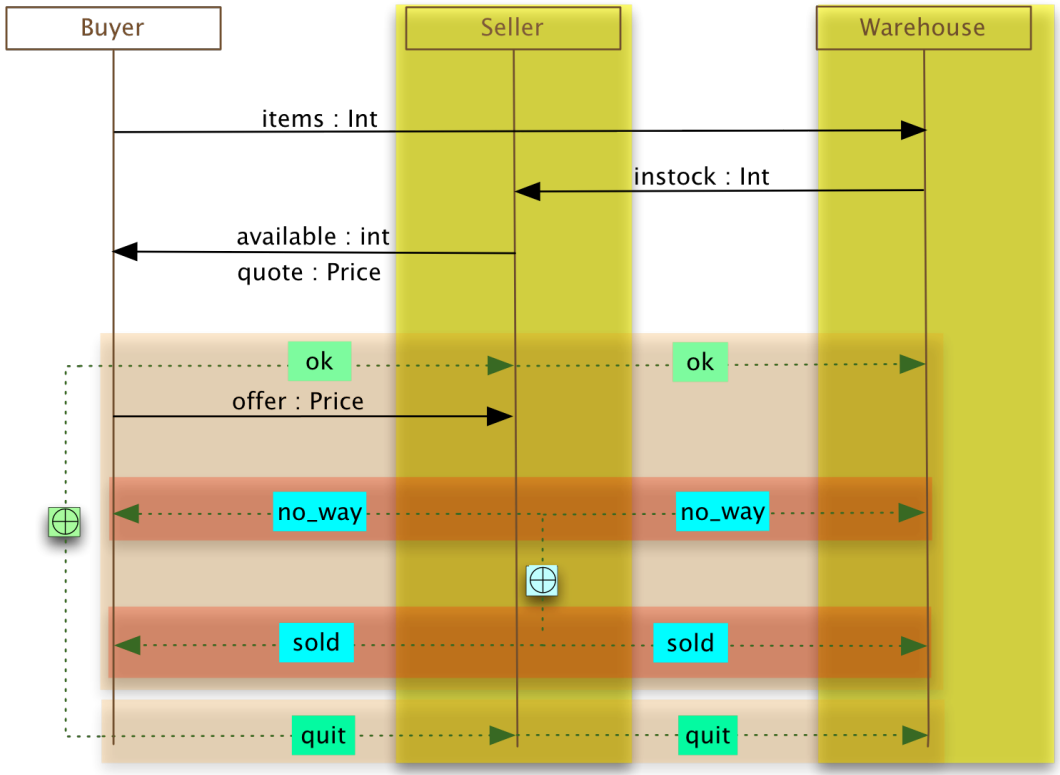


An intuitive account...



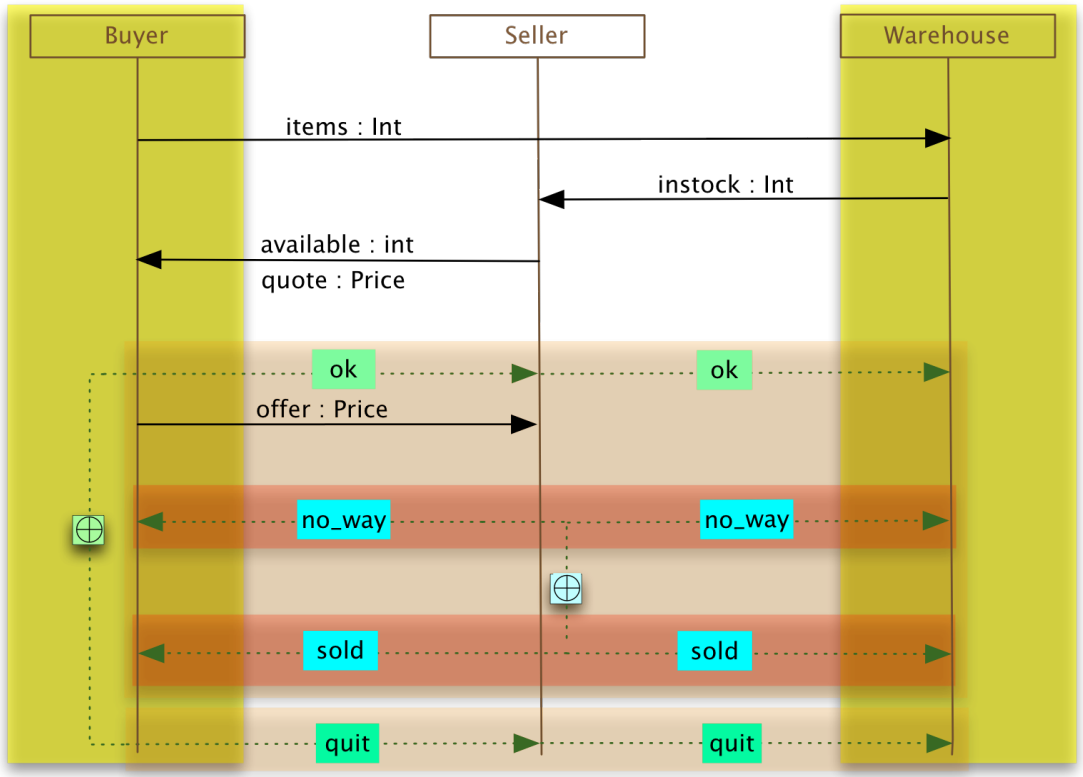
Global viewpoint

An intuitive account...



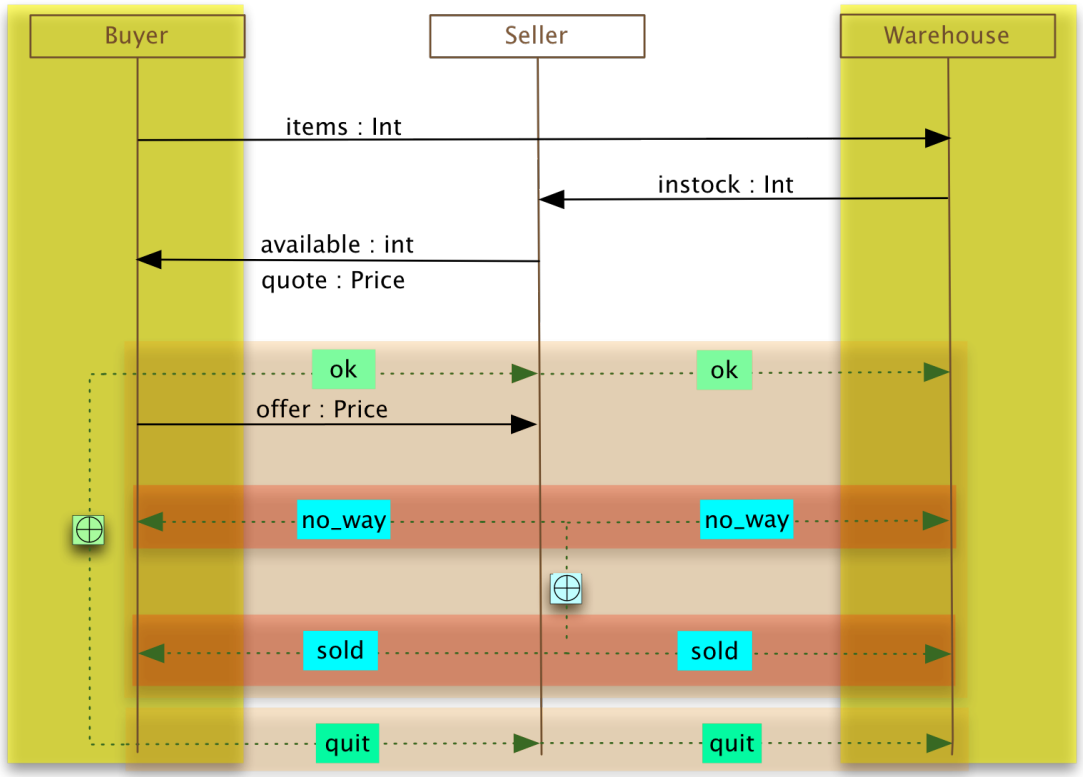
Projecting on buyer

An intuitive account...



Projecting on seller

An intuitive account...



Life is harder...recall the bugs of pingpong

Projecting on seller

Global views...a bit more formally

G-choreographies [10, 6]

$$\mathsf{G}, \mathsf{G}' \ ::= \ \circledcirc \ \mid \ \mathsf{A} \to \mathsf{B} : \mathsf{m} \ \mid \ \mathsf{G} \mid \mathsf{G}' \ \mid \ \mathsf{G}; \mathsf{G}' \ \mid \ \mathsf{G} + \mathsf{G}' \ \mid \ *\mathsf{G}$$

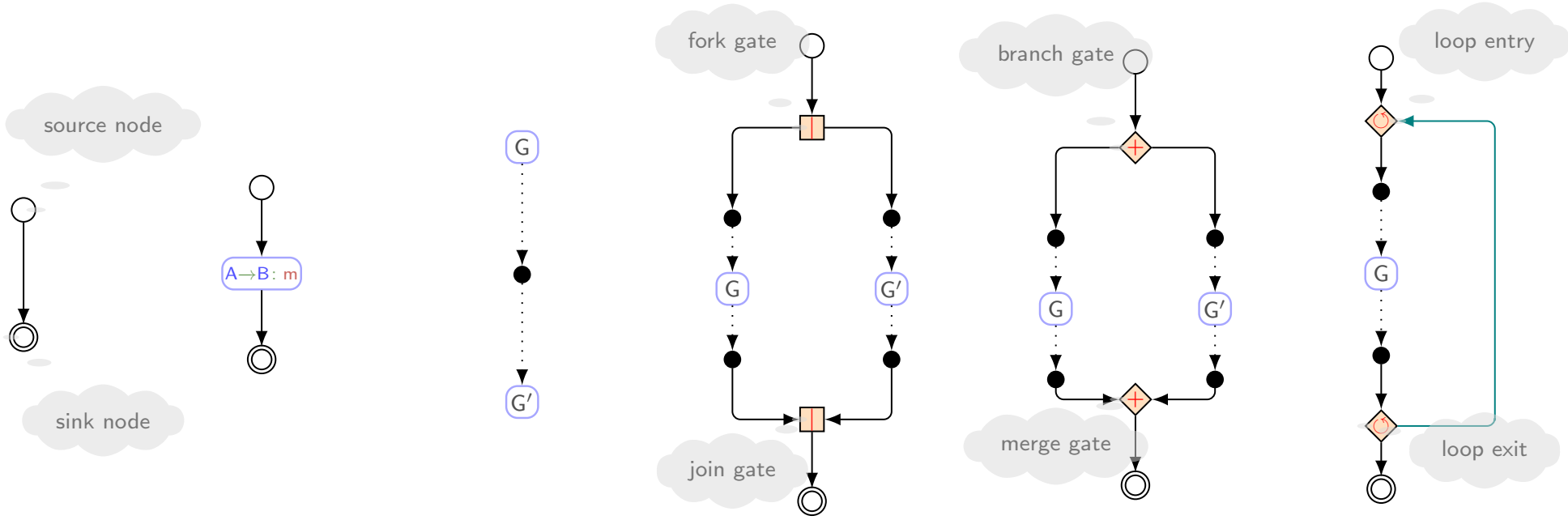
i.e., regular expressions (on an alphabet of interactions) with parallel composition

Global views...a bit more formally

G-choreographies [10, 6]

$$\mathsf{G}, \mathsf{G}' \;::=\; \circledcirc \;\mid\; \mathsf{A}\to\mathsf{B}\colon \mathsf{m} \;\mid\; \mathsf{G} \mid \mathsf{G}' \;\mid\; \mathsf{G}; \mathsf{G}' \;\mid\; \mathsf{G} + \mathsf{G}' \;\mid\; *\mathsf{G}$$

i.e., regular expressions (on an alphabet of interactions) with parallel composition



empty | interaction | sequential | parallel | branch | iteration

Global views' semantics as pomsets

A **pomset** on a set $\mathcal{E}$ of **events** is (an isomorphism class of) a **labelled partially-order** $(\mathcal{E}, \lambda, \leq)$, with

- $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function
- $\leq \quad \subseteq \quad \mathcal{E} \times \mathcal{E}$ reflexive, anti-symmetric, transitive

Our basic ingredients:

Global views' semantics as pomsets

A **pomset** on a set $\mathcal{E}$ of **events** is (an isomorphism class of) a **labelled partially-order** $(\mathcal{E}, \lambda, \leq)$, with

- $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function
- $\leq \quad \subseteq \quad \mathcal{E} \times \mathcal{E}$ reflexive, anti-symmetric, transitive

Our basic ingredients:

- a set $\mathcal{M}$ of **messages**,

Global views' semantics as pomsets

A **pomset** on a set $\mathcal{E}$ of **events** is (an isomorphism class of) a **labelled partially-order** $(\mathcal{E}, \lambda, \leq)$, with

- $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function
- $\leq \quad \subseteq \quad \mathcal{E} \times \mathcal{E}$ reflexive, anti-symmetric, transitive

Our basic ingredients:

- a set $\mathcal{M}$ of **messages**,
- a set $\mathcal{P}$ of **participants' identities**,

Global views' semantics as pomsets

A **pomset** on a set $\mathcal{E}$ of **events** is (an isomorphism class of) a **labelled partially-order** $(\mathcal{E}, \lambda, \leq)$, with

- $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function
- $\leq \quad \subseteq \quad \mathcal{E} \times \mathcal{E}$ reflexive, anti-symmetric, transitive

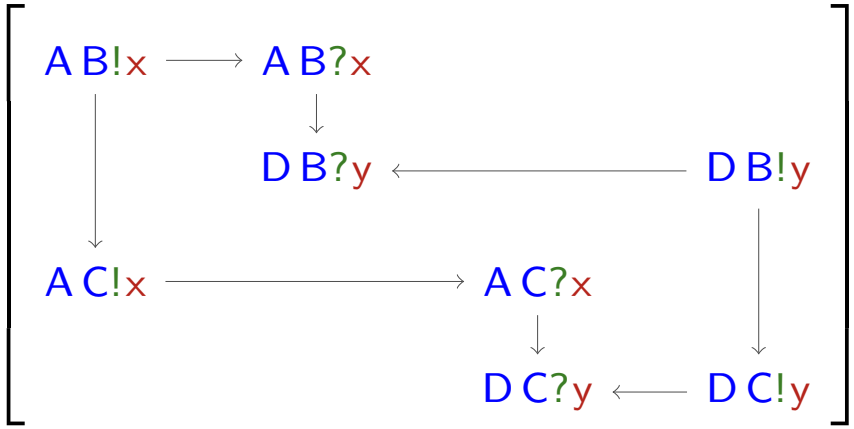
Our basic ingredients:

- a set $\mathcal{M}$ of **messages**,
- a set $\mathcal{P}$ of **participants' identities**,
- a set $\mathcal{C} = \mathcal{P} \times \mathcal{P} \setminus \{(\mathsf{A}, \mathsf{A}) \mid \mathsf{A} \in \mathcal{P}\}$ of **channels**

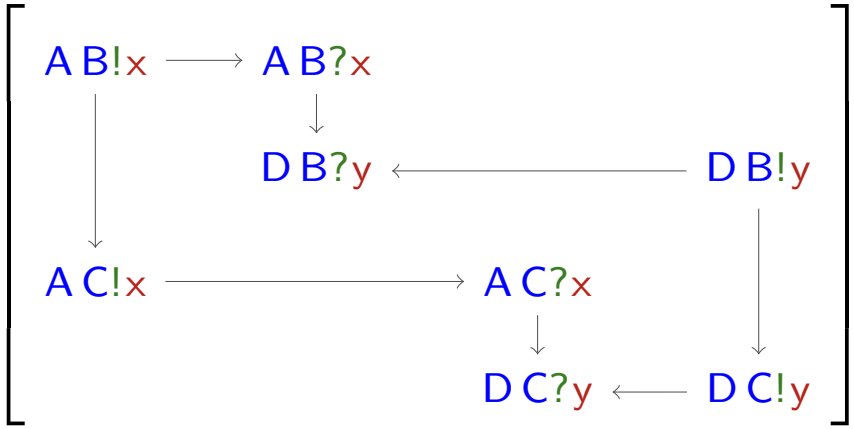
Communication events: $\mathcal{E} = \mathcal{E}^! \cup \mathcal{E}^?$ where

$$\mathcal{E}^! = \mathcal{C} \times \{!\} \times \mathcal{M}$$
$$\mathcal{E}^? = \mathcal{C} \times \{?\} \times \mathcal{M}$$

A simple pomset of communication events



A simple pomset of communication events



Exercise

Find a g-choreography that corresponds to the pomset on the left.
Another unfair question!

Pomsets semantics of g-choreographies

We define $[\![\ ]\!] : \mathsf{G} \mapsto$ set of pomsets as[1]

$$[\![\circledcirc]\!] = \{\epsilon\}$$

$$[\![\mathsf{A}\to\mathsf{B}:\ \mathsf{m}]\!] = \{[\mathsf{A}\,\mathsf{B}!\mathsf{m} \to \mathsf{A}\,\mathsf{B}?\mathsf{m}]\}$$

$$[\![\mathsf{G} \mid \mathsf{G}']\!] = \{[\text{disjoint union of } r \text{ and } r'] \mid (r, r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!]\}$$

$$[\![\mathsf{G}; \mathsf{G}']\!] = \begin{cases} \bigcup_{(r,r')\in[\![\mathsf{G}]\!]\times[\![\mathsf{G}']\!]} \{\mathsf{seq}(r, r')\} & \text{if } \forall (r, r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!] : \ ws(r, r') \\ \text{undef} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{G} + \mathsf{G}']\!] = \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] & \text{if } wb(\mathsf{G}, \mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases}$$

Pomsets semantics of g-choreographies

We define $[\![\ ]\!] : \mathsf{G} \mapsto$ set of pomsets as[1]

$$[\![\circledcirc]\!] = \{\epsilon\}$$

$$[\![\mathsf{A}\to\mathsf{B}:\mathsf{m}]\!] = \{[\mathsf{A}\,\mathsf{B}!\mathsf{m} \to \mathsf{A}\,\mathsf{B}?\mathsf{m}]\}$$

$$[\![\mathsf{G} \mid \mathsf{G}']\!] = \{[\text{disjoint union of } r \text{ and } r'] \mid (r, r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!]\}$$

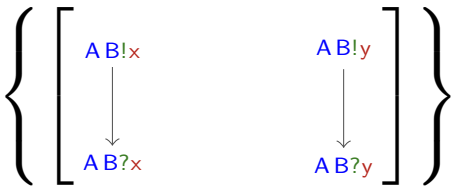
$$[\![\mathsf{G}; \mathsf{G}']\!] = \begin{cases} \bigcup_{(r,r')\in[\![\mathsf{G}]\!]\times[\![\mathsf{G}']\!]} \{\mathsf{seq}(r, r')\} & \text{if } \forall (r, r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!] : \ ws(r, r') \\ \text{undef} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{G} + \mathsf{G}']\!] = \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] & \text{if } wb(\mathsf{G}, \mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases}$$

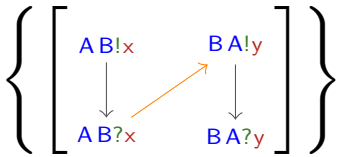
[1] Assuming single-threadness.

Some simple examples

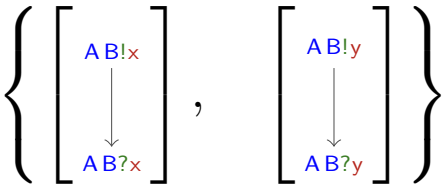
$A \to B : x \mid A \to B : y$



$A \to B : x;\ B \to A : y$



$A \to B : x + A \to B : y$



Exercise

Now you can check your answer to the exercise on slide 20.

References I

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*.
MIT Press, Cambridge, MA, USA, 1986.

[2] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines.
JACM, 30(2):323–342, 1983.

[3] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata.
In *ESOP 2012*, pages 194–213, 2012.

[4] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process.
J. ACM, 32(2):374–382, apr 1985.

[5] Roberto Guanciale and Emilio Tuosto. An abstract semantics of the global view of choreographies.
In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pages 67–82, 2016.

[6] Roberto Guanciale and Emilio Tuosto. Realisability of pomsets.
Journal of Logic and Algebraic Methods in Programming, 108:69–89, 2019.

[7] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence.
In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.

[8] Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0.
http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217. Working Draft 17 December 2004.

[9] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From Communicating Machines to Graphical Choreographies.
In *POPL15*, pages 221–232, 2015.

References II

[10] Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *Journal of Logic and Algebraic Methods in Programming*, 95:17–40, 2018. Revised and extended version of [5]. available at http://www.cs.le.ac.uk/people/et52/jlamp-with-proofs.pdf.