Formal Methods for Communication Protocols

Harnessing distributed software design with behavioural contracts

Dragiša Žunić @ GSSI & Emilio Tuosto @ GSSI

3 - 12 March, 2026 - Novi Sad

– Motivations –

An overview of the course

Logistics

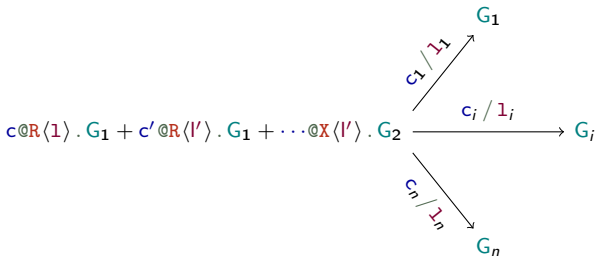
- Timetable
- These lectures are about interactions ...so do interact ☺

Motivations

Behavioural Design-by-Contracts for distributed coordination

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

A glance at prototype tools


$\mathtt{c}\,@\,\mathtt{R}\langle \mathtt{l} \rangle \,.\, \mathtt{G}_1 + \mathtt{c}'\,@\,\mathtt{R}\langle \mathtt{l}' \rangle \,.\, \mathtt{G}_1 + \cdots @\,\mathtt{X}\langle \mathtt{l}' \rangle \,.\, \mathtt{G}_2$
$\mathtt{c}_1 \,/\, \mathtt{l}_1$
$\mathtt{G}_1$
$\mathtt{c}_i \,/\, \mathtt{l}_i$
$\mathtt{G}_i$
$\mathtt{c}_n \,/\, \mathtt{l}_n$
$\mathtt{G}_n$

On program comprehension

Erlang

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

```
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]).
```

On program comprehension

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

```
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!

On program comprehension

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

```
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

On program comprehension

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

```
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

On program comprehension

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

```
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]

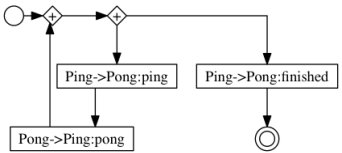
CFSMs: FIFO buffers as well

Friendlier representations with (formal) models

Local behaviour: communicating machines [2]

CFSMs: FIFO buffers as well

Choregraphy: global graph [3, 11]



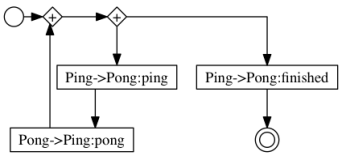
...“synchronous” distributed workflow

Friendlier representations with (formal) models

Local behaviour: communicating machines [2]

CFSMs: FIFO buffers as well

Choregraphy: global graph [3, 11]



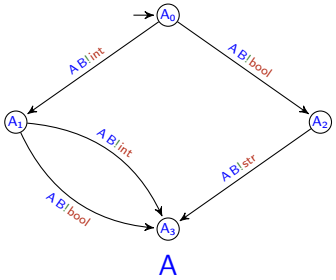
...“synchronous” distributed workflow

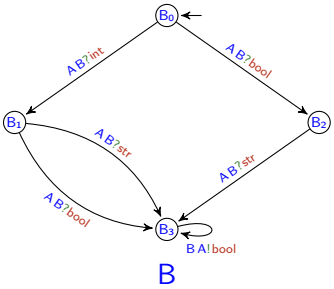
Research direction

How do we extract such models from code or documentation?

Our first formal model...informally

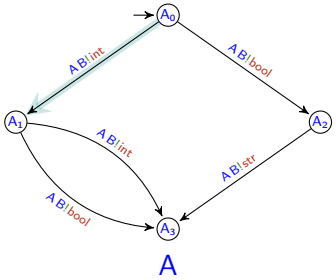
Communicating systems

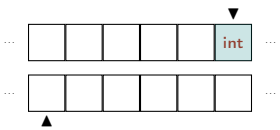


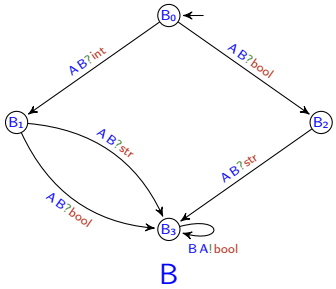


Our first formal model...informally

Communicating systems

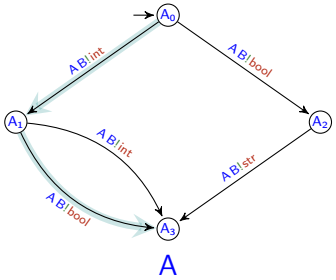


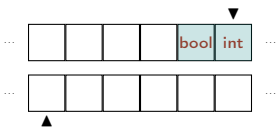


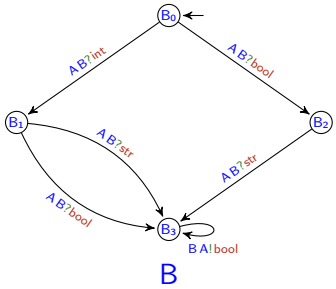


Our first formal model...informally

Communicating systems

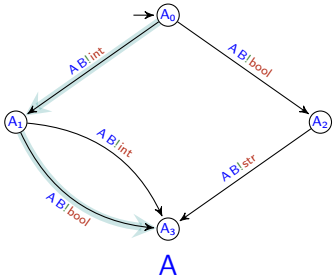


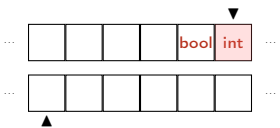


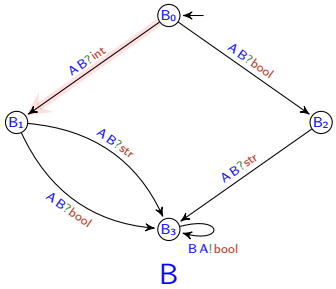


Our first formal model...informally

Communicating systems

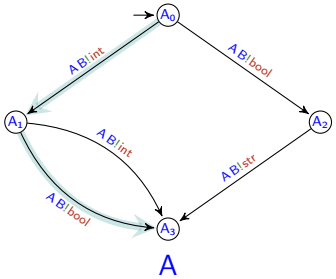


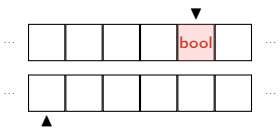


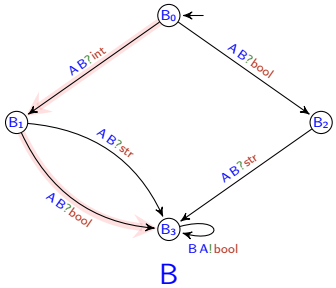


Our first formal model...informally

Communicating systems

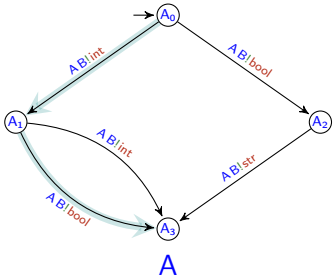


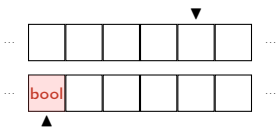


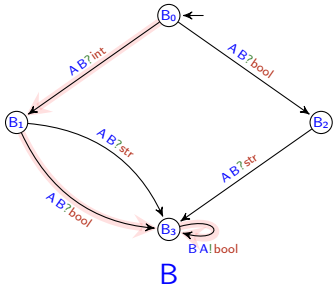


Our first formal model...informally

Communicating systems

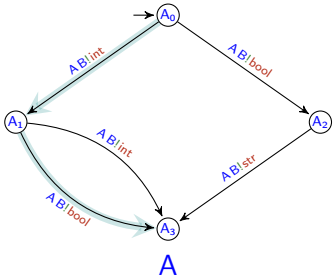


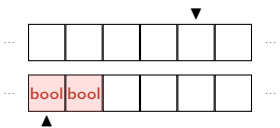


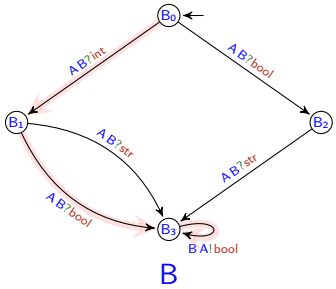


Our first formal model...informally

Communicating systems

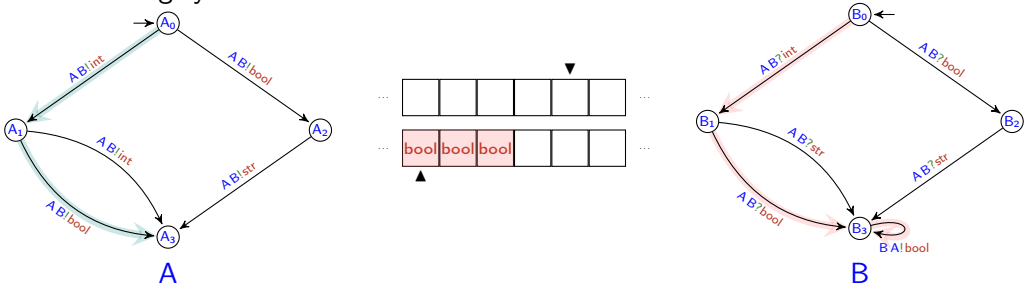






Our first formal model...informally

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

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

8  ping(0, Pong_PID) ->
9    Pong_PID ! finished,
10   io:format("ping finished~n", []);
```

```
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?

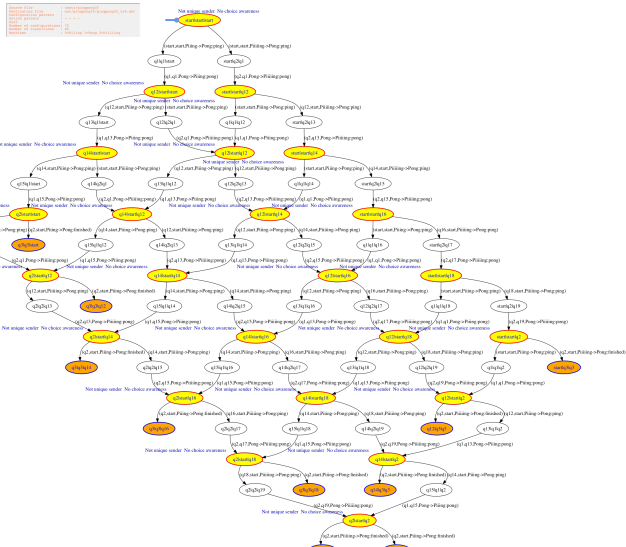
A:

No!

Exercise:

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

2 ping clients and 1 pong server!!!



Some reflections

Some reflections

What does 'software' actually mean?

Some reflections

What does 'software' actually mean?

My view:

Some reflections

What does 'software' actually mean?

My view:

- SW is not just 'code'

Some reflections

What does 'software' actually mean?

My view:

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

Some reflections

What does 'software' actually mean?

My view:

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

Some reflections

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!

Some reflections

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!

Some reflections

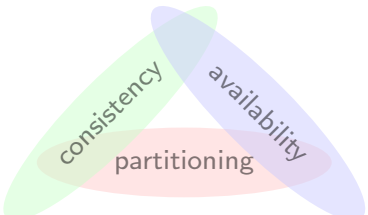
What does 'software' actually mean?

My view:

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

Would you buy a house without guarantees that its roof won't collapse? or that its heating system is working properly?

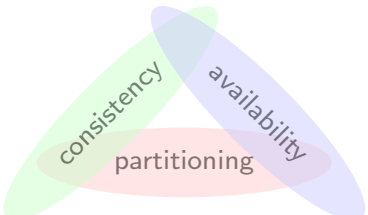
FM theory in practice



The CAP theorem

Distributed agreement
Distributed sharing
Security
Computer-assisted collaborative work
...

FM theory in practice



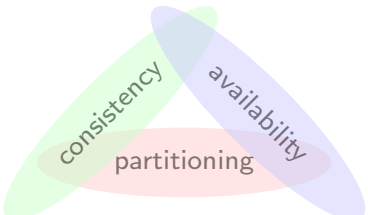
The CAP theorem

Distributed agreement
Distributed sharing
Security
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)

FM theory in practice



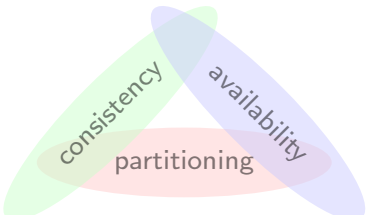
The CAP theorem

Distributed agreement
Distributed sharing
Security
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)
- Distributed consensus (with chosen weaknesses)

FM theory in practice



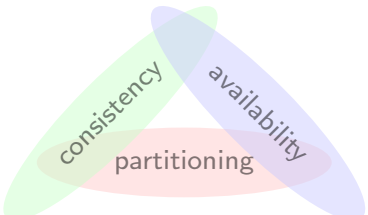
The CAP theorem

Distributed agreement
Distributed sharing
Security
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)
- Distributed consensus (with chosen weaknesses)
- Commutative replicated data types (weakening consistency)

FM theory in practice



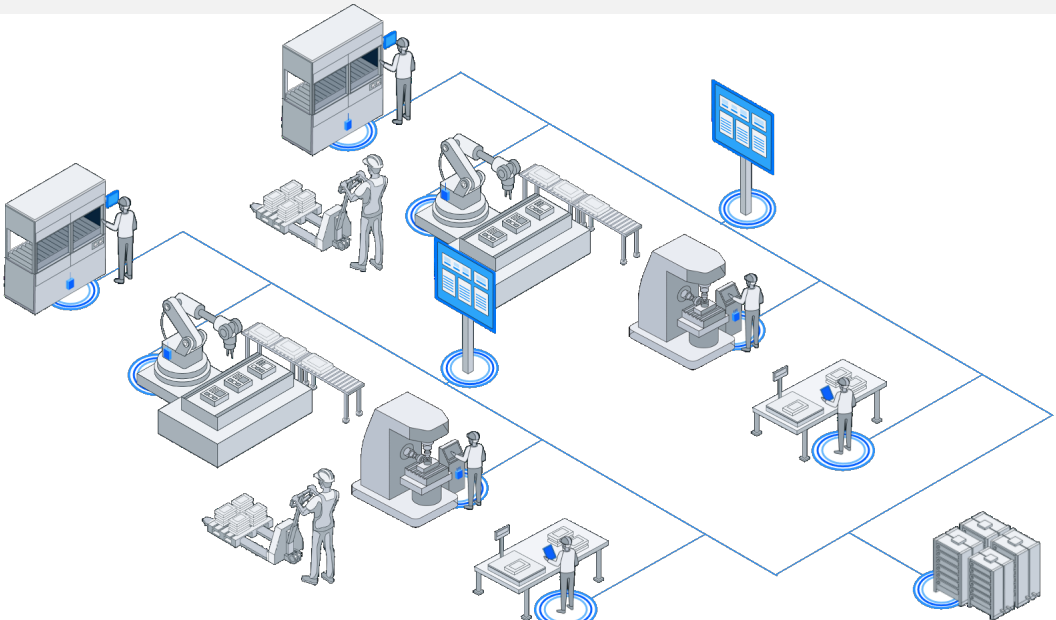
The CAP theorem

Distributed agreement
Distributed sharing
Security
Computer-assisted collaborative work
...

With some "solutions"

- Centralisation points (not that appealing)
- Distributed consensus (with chosen weaknesses)
- Commutative replicated data types (weakening consistency)
- ...

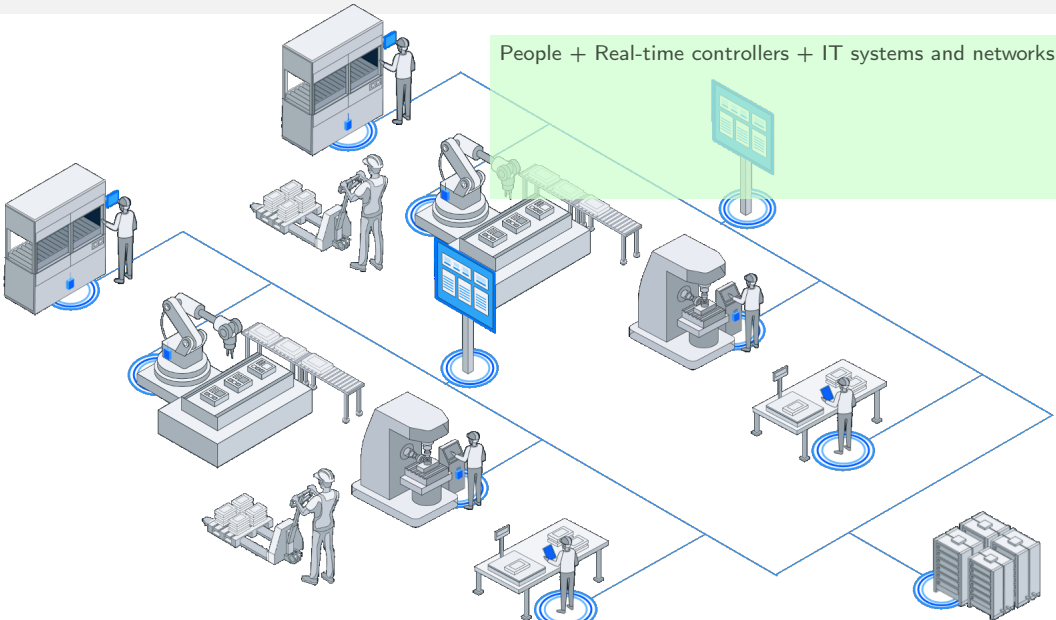
An execution model for a collaborative environment



(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

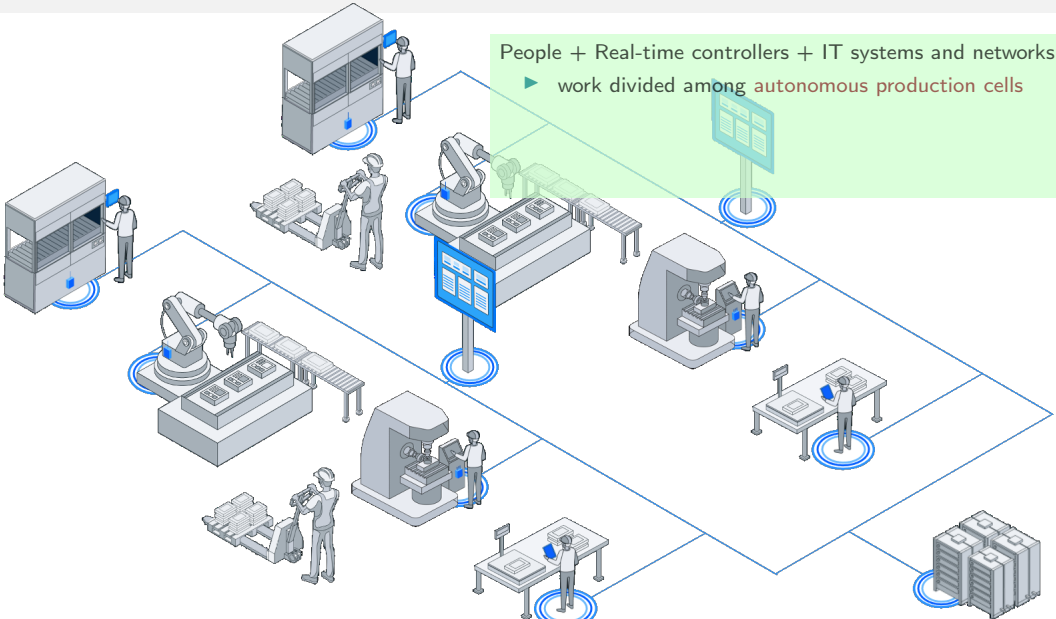


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells

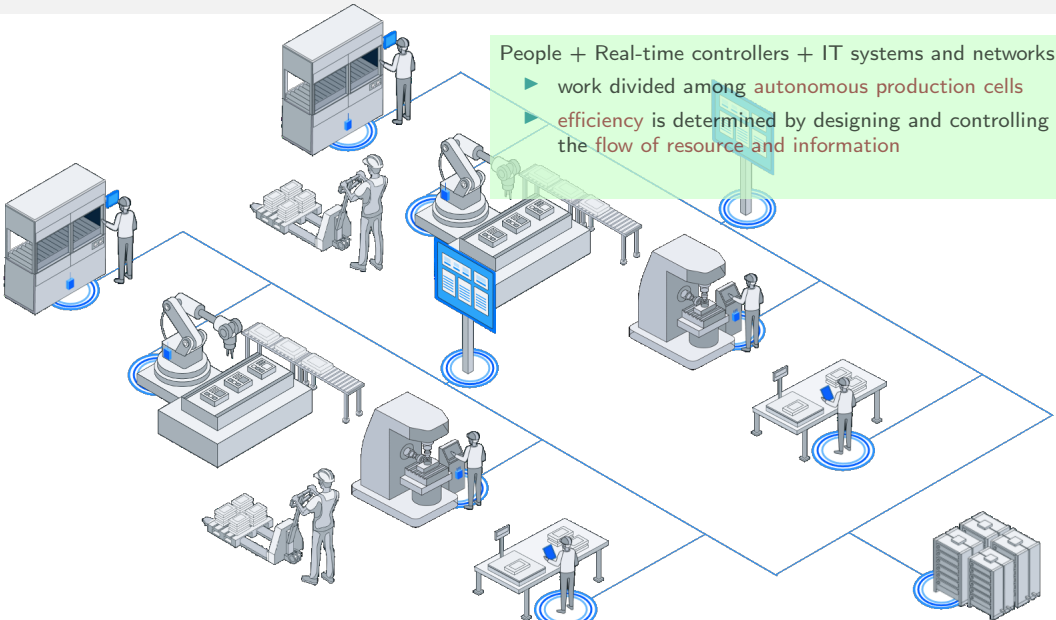


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

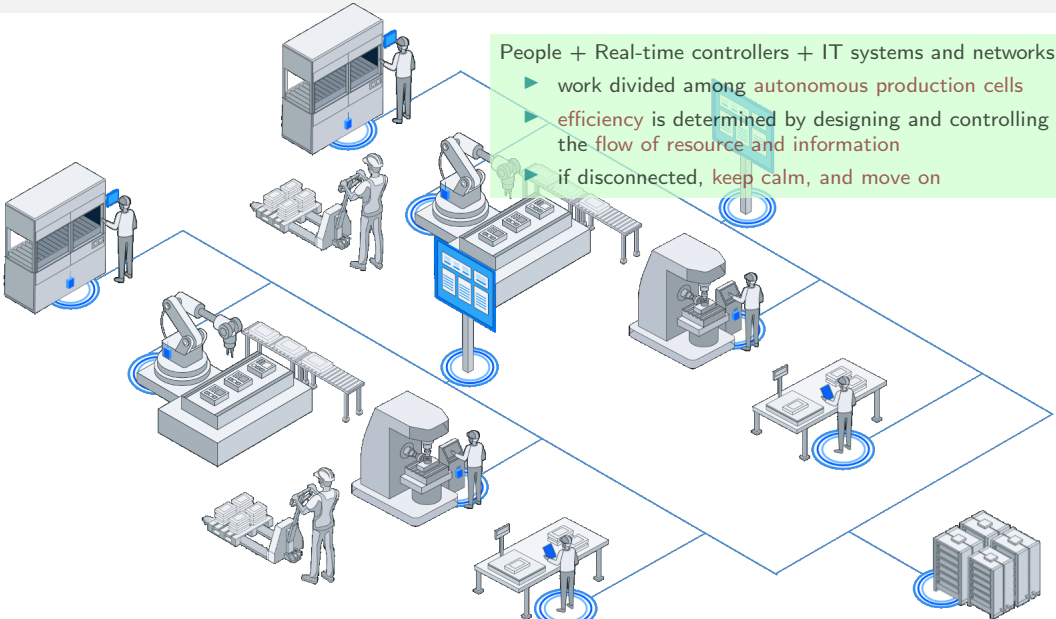
People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information



(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment



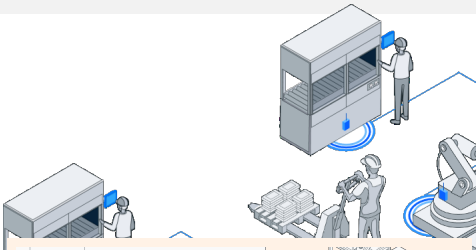
People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



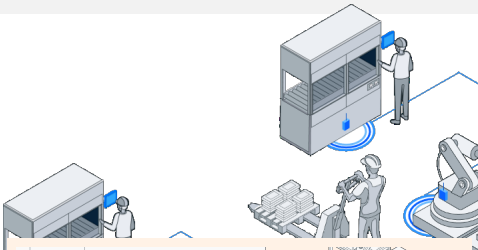
People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

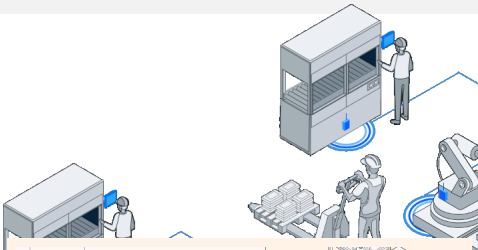
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

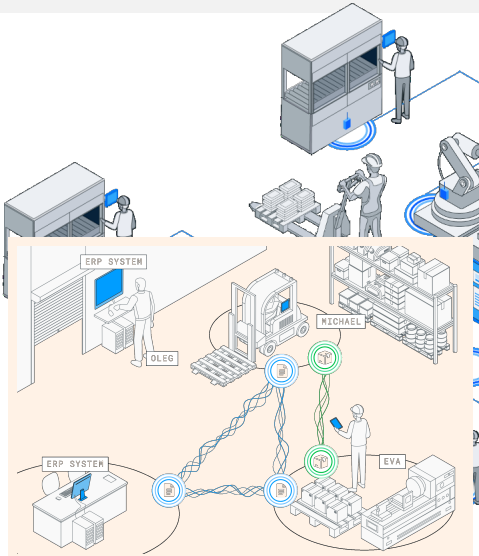
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

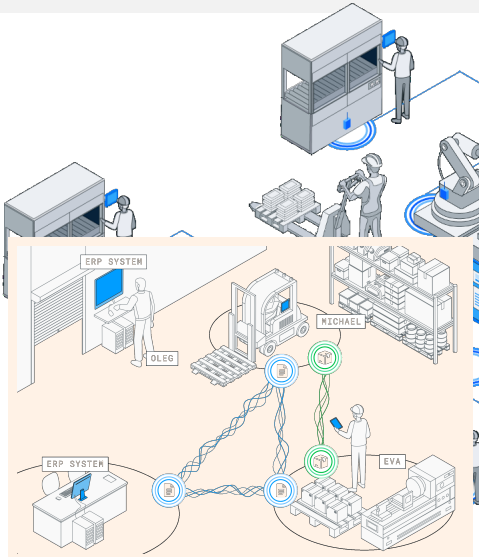
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

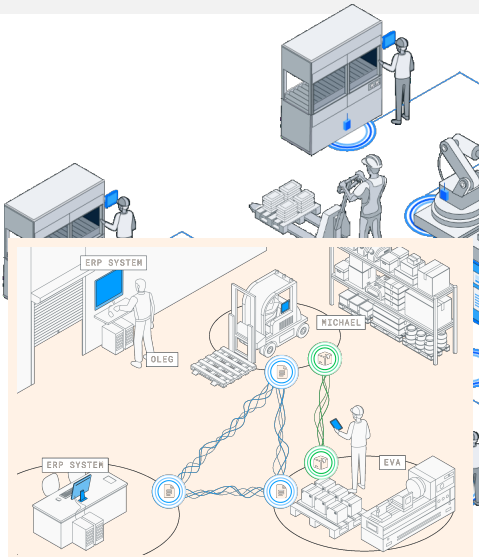
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

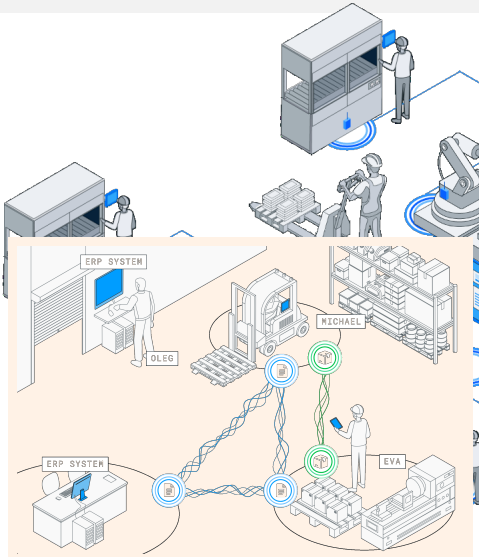
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins
- a log determines the computational state of its twin

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins
- a log determines the computational state of its twin
- replicated logs are merged

Distribute all

Are there any non-distributed applications?

Robots (e.g., rescue missions or space applications)

Collaborative applications (`https://automerge.org/`)

Home automation

Distribute all

The cloud...always?

Is it safe to let your fridge and mobile go in the cloud to interact?

Where is your data?

Where is your privacy?

Distribute all

"Anytime, anywhere..." really?

like during the AWS's outage on 25/11/2020

or almost all Google services down on 14/12/2020

DSL typical availability of 97% (& some SLA have no lower bound); checkout
`https://www.internetsociety.org/blog/2022/03/what-is-the-digital-divide/`

Distribute all

Also, taking decisions locally

can reduce downtime

shifts data ownership

gets rid of any centralization point...for real

design (well)

+

project

+

run

A motto

design (well)

+

project

+

run

Another motto

execute

+

propagate

+

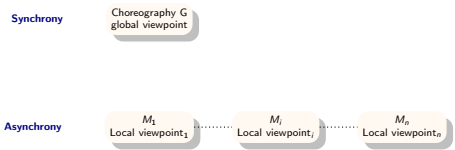
merge

– Choreographic Models –

"Top-down"

Quoting W3C [8]

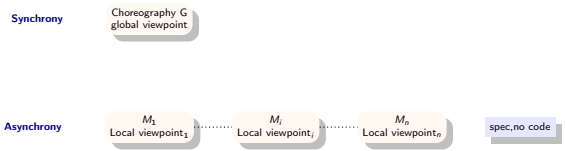
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can 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]

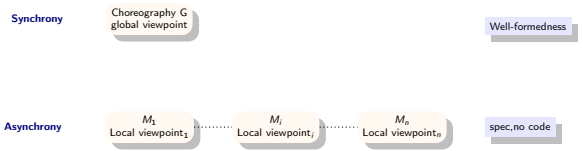
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can 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]

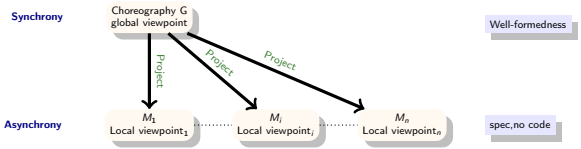
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can 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]

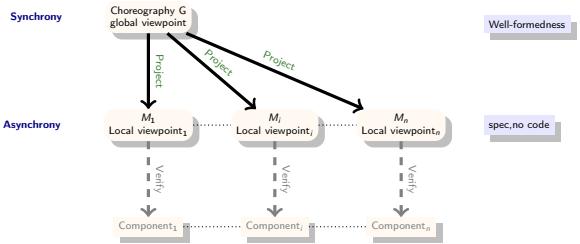
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can 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 [...]"



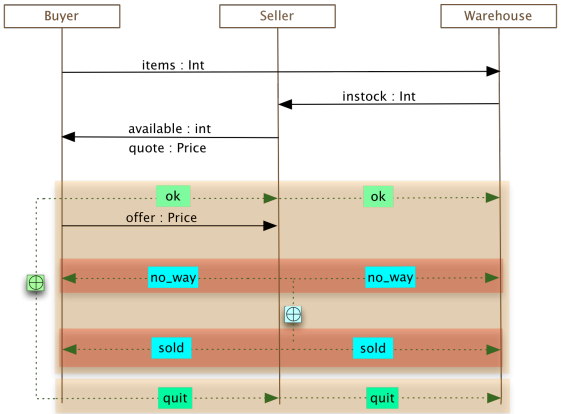
"Top-down"

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 [...]"

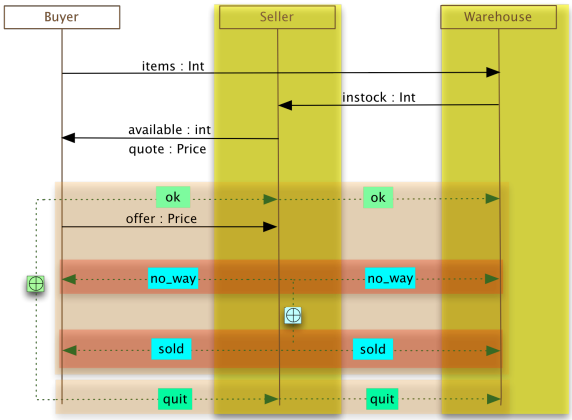


An intuitive account...



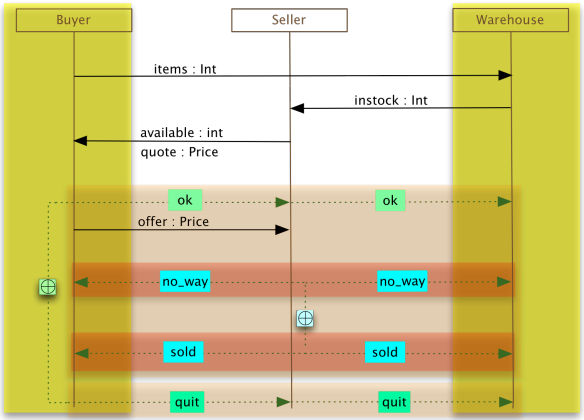
Global viewpoint

An intuitive account...



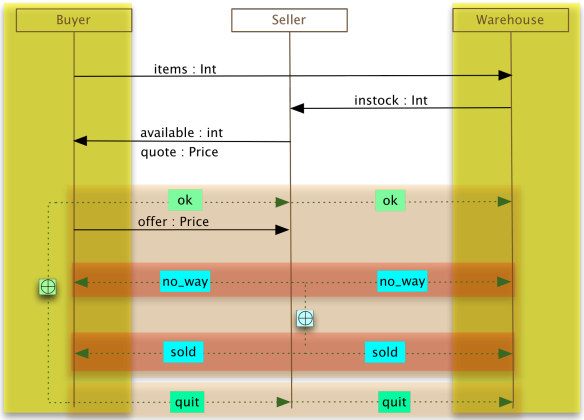
Projecting on buyer

An intuitive account...



Projecting on seller

An intuitive account...



Projecting on seller

Life is harder...recall the bugs of pingpong

Global views...a bit more formally

G-choreographies [13, 6]

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

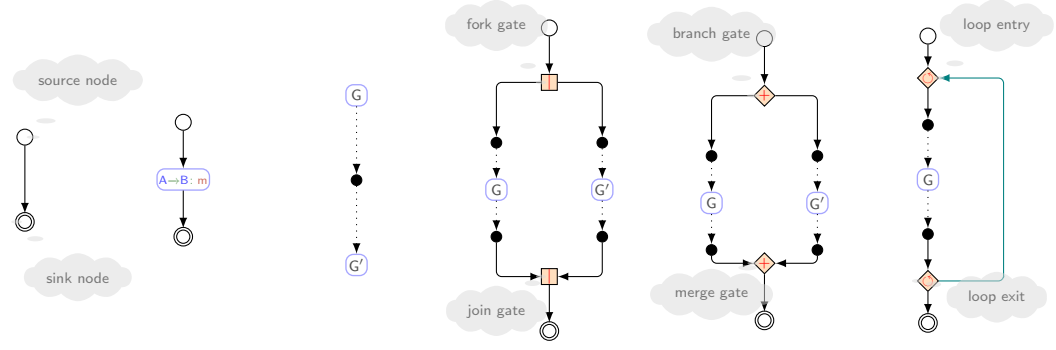
i.e., regular expressions (on an alphabet of interactions) with parallel composition

Global views...a bit more formally

G-choreographies [13, 6]

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

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



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

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 $\mathcal{C} = \mathcal{P} \times \mathcal{P} \setminus \{(A, A) \mid A \in \mathcal{P}\}$ of **channels**

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
- $\leq \ \subseteq \ \mathcal{E} \times \mathcal{E}$ reflexive, anti-symmetric, transitive

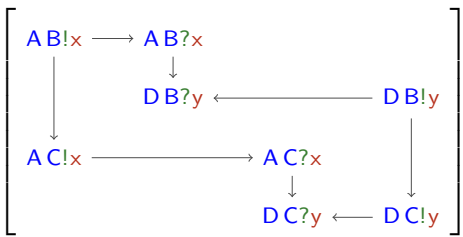
Our basic ingredients:

- 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**

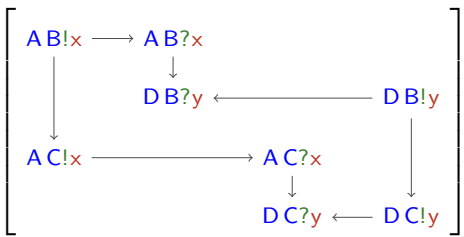
Communication events: $\mathcal{E} = \mathcal{E}^! \ \cup \ \mathcal{E}^?$ where

$$\begin{aligned} \mathcal{E}^! &= \mathcal{C} \times \{!\} \times \mathcal{M} \\ \mathcal{E}^? &= \mathcal{C} \times \{?\} \times \mathcal{M} \end{aligned}$$

A simple pomset of communication events



A simple pomset of communication events



Exercise

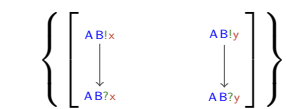
Find a g-choreography that corresponds to the pomset on the left.
Another unfair question!

Pomsets semantics of g-choreographies

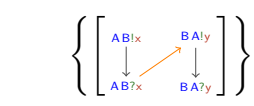
We define $[\![\ ]\!] : \mathsf{G} \mapsto$ set of pomsets as

$$
\begin{aligned}
[\![\circledcirc]\!] &= \{\epsilon\} \\
[\![\mathsf{A}{\to}\mathsf{B}\colon \mathsf{m}]\!] &= \{[\mathsf{A}\,\mathsf{B}!\mathsf{m} \to \mathsf{A}\,\mathsf{B}?\mathsf{m}]\} \\
[\![\mathsf{G} \mid \mathsf{G}']\!] &= \{[\text{disjoint union of } r \text{ and } r'] \mid (r, r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!]\} \\
[\![\mathsf{G}; \mathsf{G}']\!] &= \begin{cases} \bigcup_{(r,r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!]} \{\mathsf{seq}(r, r')\} & \text{if } \forall (r, r') \in [\![\mathsf{G}]\!] \times [\![\mathsf{G}']\!] : \ ws(r, r') \\ \text{undef} & \text{otherwise} \end{cases} \\
[\![\mathsf{G} + \mathsf{G}']\!] &= \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] & \text{if } wb(\mathsf{G}, \mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases}
\end{aligned}
$$

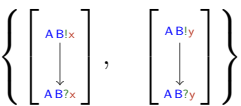
$\mathsf{A}{\to}\mathsf{B}\colon \mathsf{x} \mid \mathsf{A}{\to}\mathsf{B}\colon \mathsf{y}$



$\mathsf{A}{\to}\mathsf{B}\colon \mathsf{x};\ \mathsf{B}{\to}\mathsf{A}\colon \mathsf{y}$



$\mathsf{A}{\to}\mathsf{B}\colon \mathsf{x} + \mathsf{A}{\to}\mathsf{B}\colon \mathsf{y}$



Back to blackboard

Now we can solve the exercise on slide 20.

Well-sequencedness

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

Well-sequencedness

A→B : x; A→C : y

$r =$ [sbj A sbj B]

$r' =$ [sbj A sbj B]

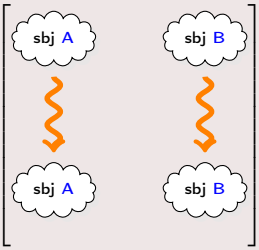
Well-sequencedness

A→B : x; A→C : y

$r =$ [sbj A sbj B]

$r' =$ [sbj A sbj B]

$\mathsf{seq}(r, r') =$



Well-sequencedness

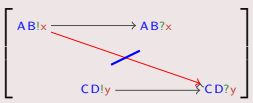
A→B : x; A→C : y

$r =$ 

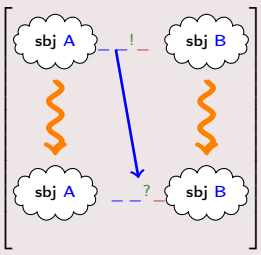
$r' =$ 

A→B : x; C→D : y

No minimal input "from the continuation"



(and no output in the continuation interfering with "left-inputs")

$ws(r, r')$ 

Well-branchedness

In a branch $G_1 + G_2$

Well-branchedness

In a branch $G_1 + G_2$

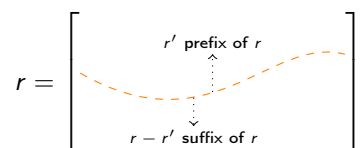
- there should be at most one active participant — non-standard

Well-branchedness

In a branch $G_1 + G_2$

- there should be at most one active participant — non-standard
- any non-active participant should be passive — standard

Problem: Participants do not necessarily "enter" a choice "immediately"
An idea: find a "common part" of the branches for which participants behaves uniformly in G_1 and G_2

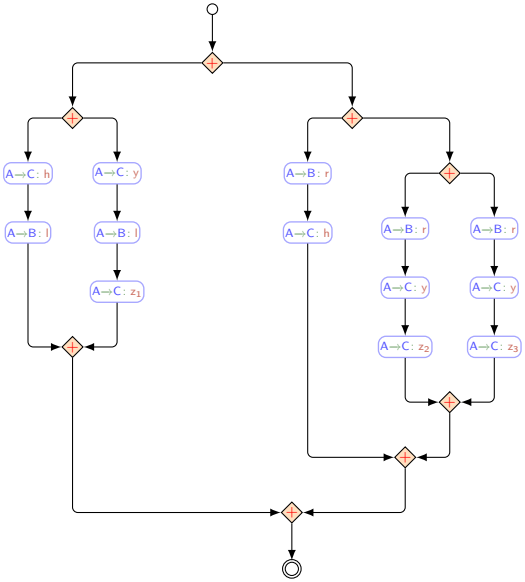


Class test

Figure out the graphical structure of the following terms and for each of them say which one is well-branched

- $G_1 = \mathsf{A}{\to}\mathsf{B}\colon \mathsf{int} + \mathsf{A}{\to}\mathsf{B}\colon \mathsf{str}$
- $G_2 = \mathsf{A}{\to}\mathsf{B}\colon \mathsf{int} + \circledcirc$
- $G_3 = \mathsf{A}{\to}\mathsf{B}\colon \mathsf{int} + \mathsf{A}{\to}\mathsf{C}\colon \mathsf{str}$
- $G_4 = \left(\begin{array}{l} \mathsf{A}{\to}\mathsf{C}\colon \mathsf{int}; \mathsf{A}{\to}\mathsf{B}\colon \mathsf{bool} \\ \quad + \\ \mathsf{A}{\to}\mathsf{C}\colon \mathsf{str}; \mathsf{A}{\to}\mathsf{C}\colon \mathsf{bool}; \mathsf{A}{\to}\mathsf{B}\colon \mathsf{bool} \end{array}\right)$

$G_{😕}$: a difficult choice



$G_{😕} = G_1 + G_2$ where

$$G_1 = \left(\begin{array}{l} A{\to}C\colon h;\ A{\to}B\colon l \\ + \\ A{\to}C\colon y;\ A{\to}B\colon l;\ A{\to}C\colon z_1 \end{array} \right)$$

$$G_2 = A{\to}B\colon r;\ A{\to}C\colon h + G_{2a} + G_{2b}$$

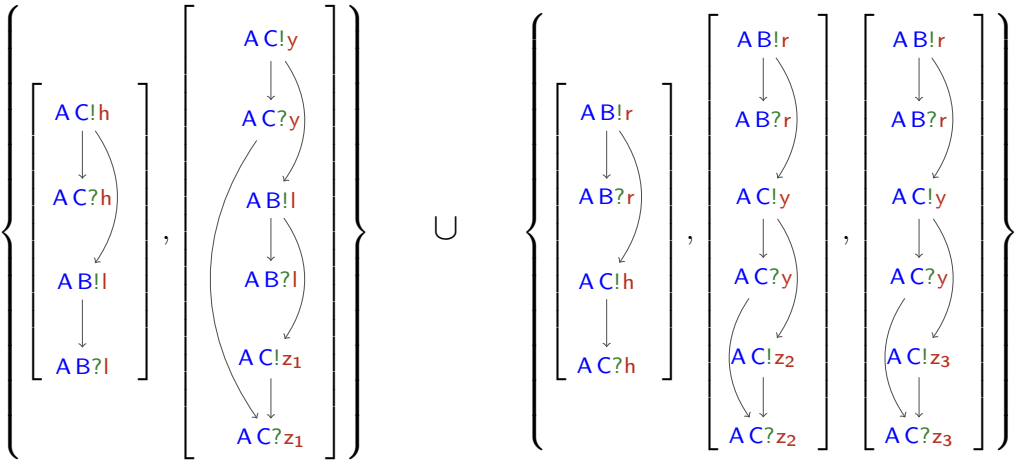
$$G_{2a} = A{\to}B\colon r;\ A{\to}C\colon y;\ A{\to}C\colon z_2$$

$$G_{2b} = A{\to}B\colon r;\ A{\to}C\colon y;\ A{\to}C\colon z_2$$

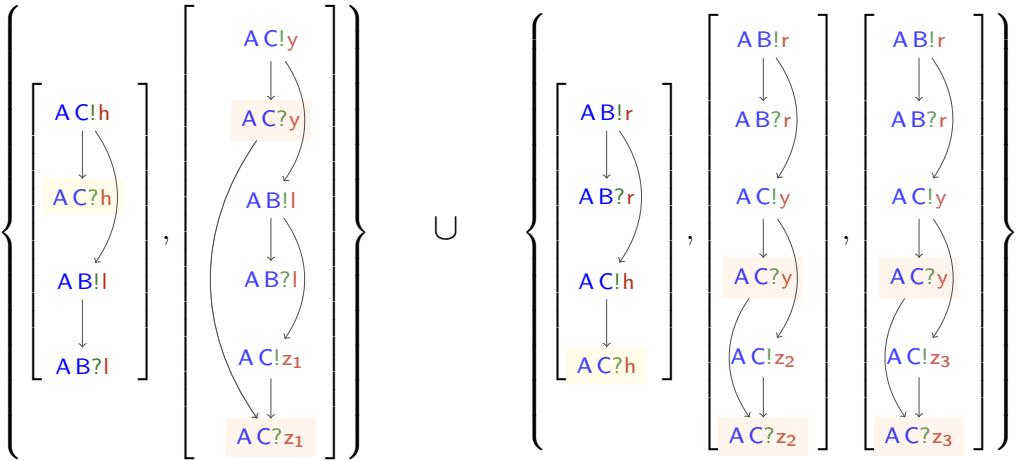
A chooses ... and

- Whatever B gets, he won't know if A and C exchanged or not h
- If C gets h, he won't know if A and B exchanged l or r

The pomsets of $G_{\text{😟}}$



The pomsets of $G_{😕}$...from C's point of view



Projecting g-choreographies

Projecting g-choreographies

Technicalities

- Functions $\_ \downarrow_{\mathsf{A}}$ yield the projection of g-choreographies on the participant A as triplets (M, q_0, q_e) with q_0 and q_e initial and terminal states respectively
- If G_1 and G_2 are sub-terms of G then we "disjointly combine" the states of $\mathsf{G}_1 \downarrow_{\mathsf{A}}$ and $\mathsf{G}_2 \downarrow_{\mathsf{A}}$; for this we define $(M, q_0, q_e) \otimes \mathbf{1}$ which transforms each state q of M in $(q, 1)$ (and likewise for $(M, q_0, q_e) \otimes \mathbf{2}$)

Base cases

$$\mathsf{G} \downarrow_{\mathsf{A}} = \begin{cases} \rightarrow (q_0) \rightarrow & \text{if } \mathsf{G} = \circledcirc \text{ or } \mathsf{G} = \mathsf{B} \rightarrow \mathsf{C} : \mathsf{m} \\ \rightarrow (q_0) \xrightarrow{\mathsf{A}\,\mathsf{B}!\mathsf{m}} (q_e) \rightarrow & \text{if } \mathsf{G} = \mathsf{A} \rightarrow \mathsf{B} : \mathsf{m}, \text{ with } q_0 \neq q_e \\ \rightarrow (q_0) \xrightarrow{\mathsf{B}\,\mathsf{A}?\mathsf{m}} (q_e) \rightarrow & \text{if } \mathsf{G} = \mathsf{B} \rightarrow \mathsf{A} : \mathsf{m}, \text{ with } q_0 \neq q_e \end{cases}$$

Projecting g-choreographies

Technicalities

- Functions $\_ \downarrow_{\mathsf{A}}$ yield the projection of g-choreographies on the participant A as triplets (M, q_0, q_e) with q_0 and q_e initial and terminal states respectively
- If G_1 and G_2 are sub-terms of G then we "disjointly combine" the states of $\mathsf{G}_1 \downarrow_{\mathsf{A}}$ and $\mathsf{G}_2 \downarrow_{\mathsf{A}}$; for this we define $(M, q_0, q_e) \otimes \mathbf{1}$ which transforms each state q of M in $(q, 1)$ (and likewise for $(M, q_0, q_e) \otimes \mathbf{2}$)

Sequential composition

$$(\mathsf{G}_1 ; \mathsf{G}_2) \downarrow_{\mathsf{A}} = \left(M_1 \sqcup \left\{ {}^{q_e^1}/_{q_0^2} \right\} M_2, q_0^1, q_e^2 \right)$$

where $(M_1, q_0^1, q_e^1) = \mathsf{G}_1 \downarrow_{\mathsf{A}} \otimes \mathbf{1}$
and $(M_2, q_0^2, q_e^2) = \mathsf{G}_2 \downarrow_{\mathsf{A}} \otimes \mathbf{2}$

Projecting g-choreographies

Technicalities

- Functions $\_ \downarrow_{\mathsf{A}}$ yield the projection of g-choreographies on the participant A as triplets (M, q_0, q_e) with q_0 and q_e initial and terminal states respectively
- If G_1 and G_2 are sub-terms of G then we "disjointly combine" the states of $\mathsf{G}_1 \downarrow_{\mathsf{A}}$ and $\mathsf{G}_2 \downarrow_{\mathsf{A}}$; for this we define $(M, q_0, q_e) \otimes \mathbf{1}$ which transforms each state q of M in $(q, 1)$ (and likewise for $(M, q_0, q_e) \otimes \mathbf{2}$)

Choice

$$(\mathsf{G}_1 + \mathsf{G}_2) \downarrow_{\mathsf{A}} = \left(\left\{ {}^{q_e^2}/_{q_e^1} \right\} M_1 \sqcup \left\{ {}^{q_0^1}/_{q_0^2} \right\} M_2, q_0^1, q_e^2 \right)$$

where $(M_1, q_0^1, q_e^1) = \mathsf{G}_1 \downarrow_{\mathsf{A}} \otimes \mathbf{1}$
and $(M_2, q_0^2, q_e^2) = \mathsf{G}_2 \downarrow_{\mathsf{A}} \otimes \mathbf{2}$

Projecting g-choreographies

Technicalities

- Functions $\_ \downarrow_{\mathsf{A}}$ yield the projection of g-choreographies on the participant A as triplets (M, q_0, q_e) with q_0 and q_e initial and terminal states respectively
- If G_1 and G_2 are sub-terms of G then we "disjointly combine" the states of $\mathsf{G}_1 \downarrow_{\mathsf{A}}$ and $\mathsf{G}_2 \downarrow_{\mathsf{A}}$; for this we define $(M, q_0, q_e) \otimes \mathbf{1}$ which transforms each state q of M in $(q, 1)$ (and likewise for $(M, q_0, q_e) \otimes \mathbf{2}$)

Parallel composition

$$(\mathsf{G}_1 \mid \mathsf{G}_2) \downarrow_{\mathsf{A}} = (M_1 \times M_2, (q_0^1, q_0^2), (q_e^1, q_e^2))$$

where $(M_1, q_0^1, q_e^1) = \mathsf{G}_1 \downarrow_{\mathsf{A}} \otimes \mathbf{1}$
and $(M_2, q_0^2, q_e^2) = \mathsf{G}_2 \downarrow_{\mathsf{A}} \otimes \mathbf{2}$

Exercise

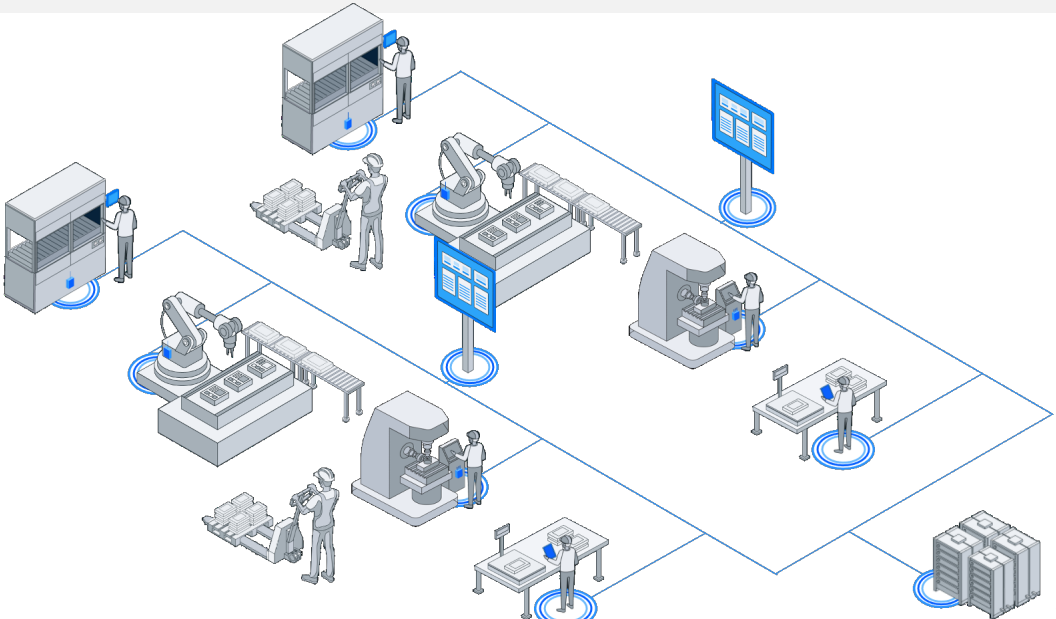
Verify that the projection of the g-choreography on 6 are the CFSM on the same slide.

– Swarms and Swarm Protocols –

design (well)
+
project
+
run

execute
+
propagate
+
merge

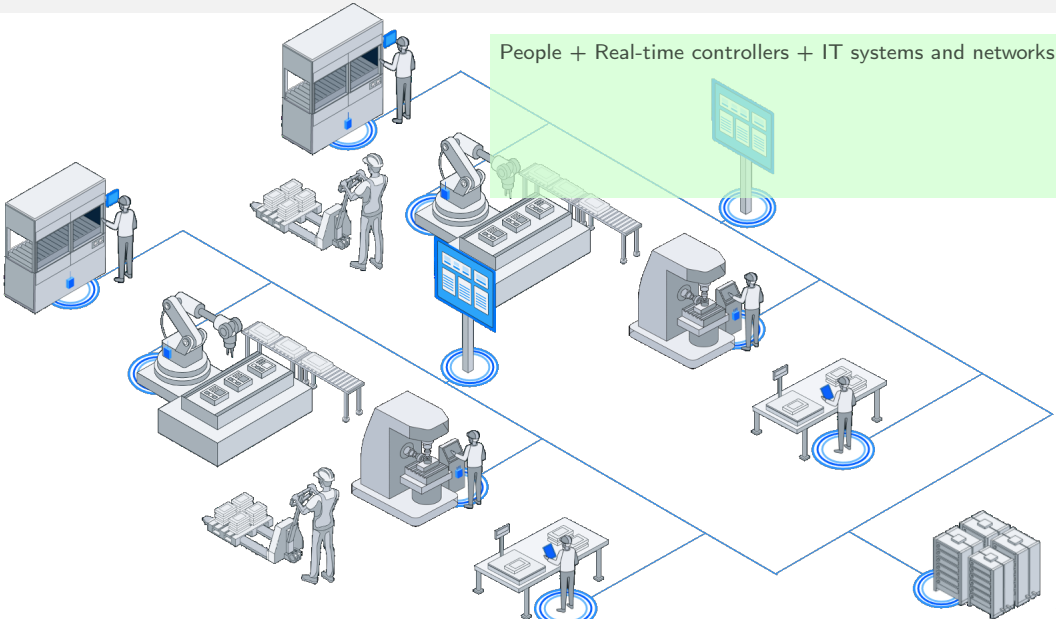
An execution model for a collaborative environment



(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

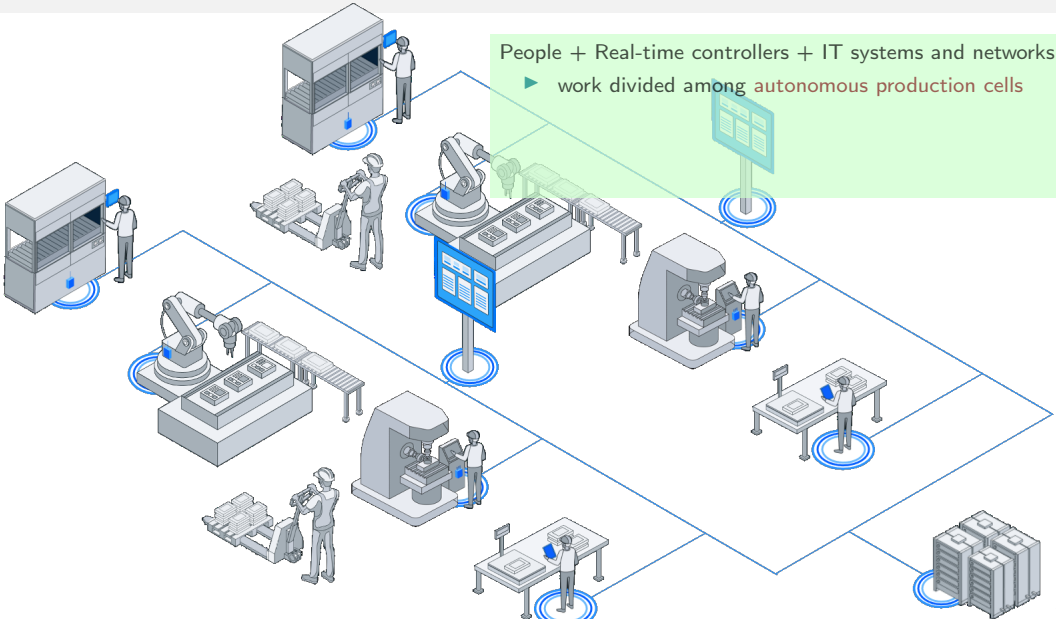


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells

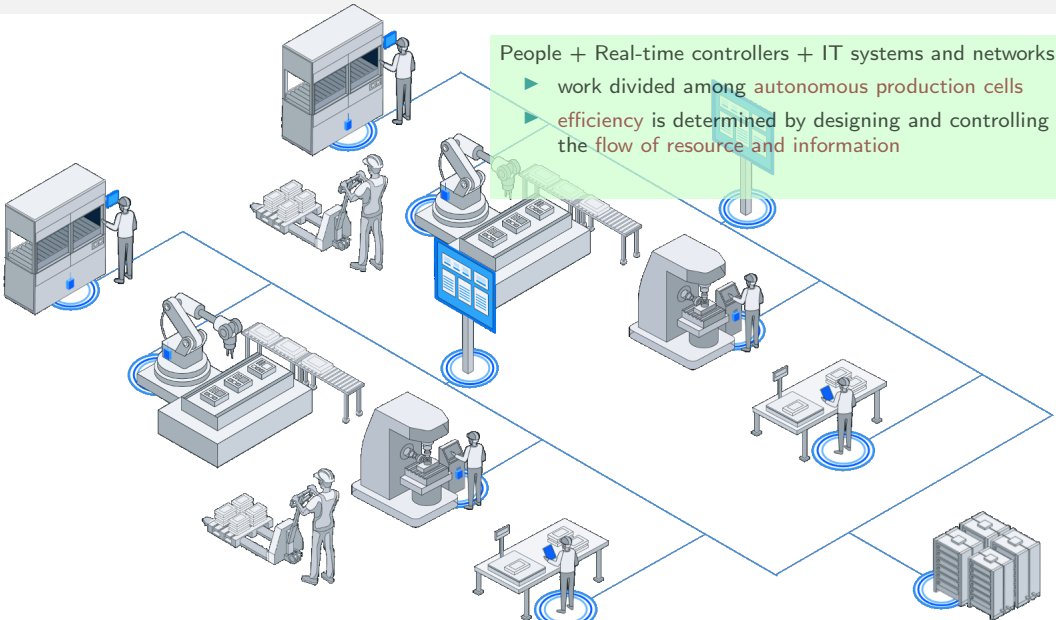


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information

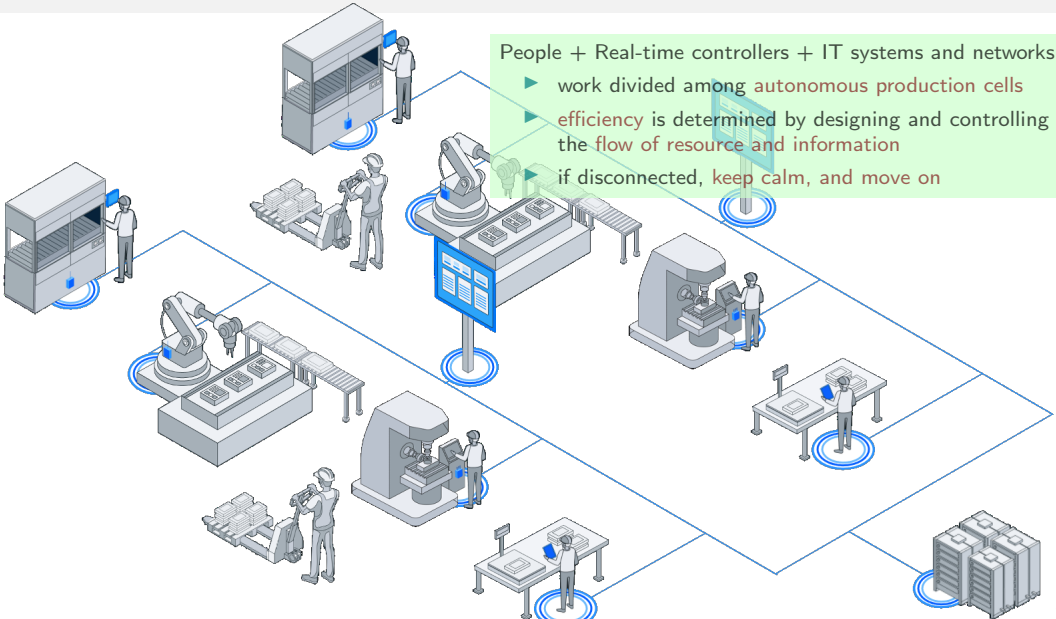


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

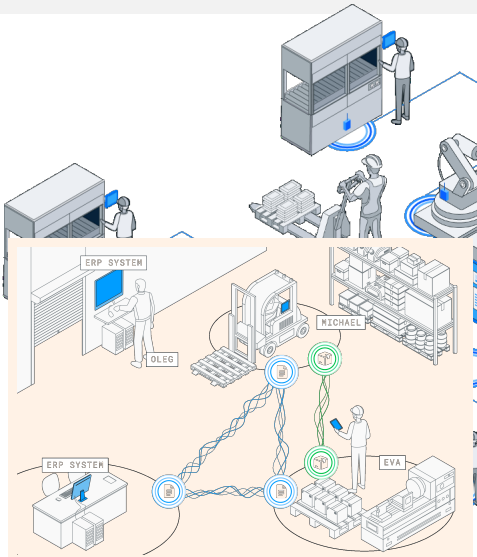


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

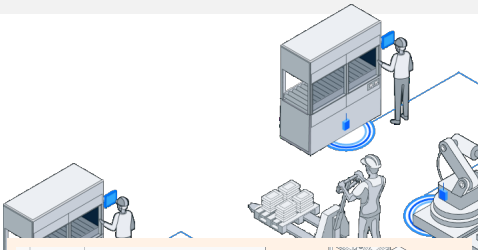


Execution model

(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator

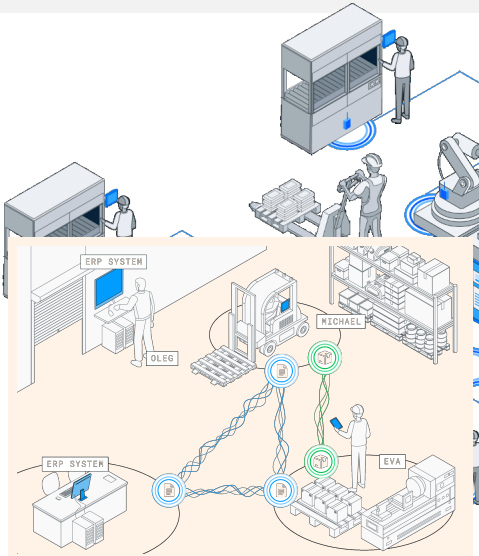
An execution model for a collaborative environment

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

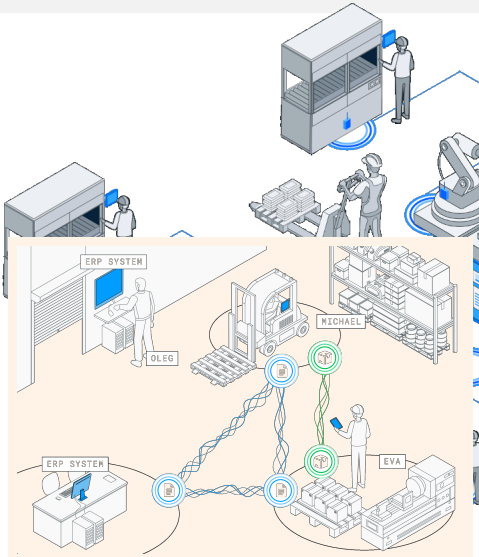
- local twin for each device/operator
- twins are replicated where needed



(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

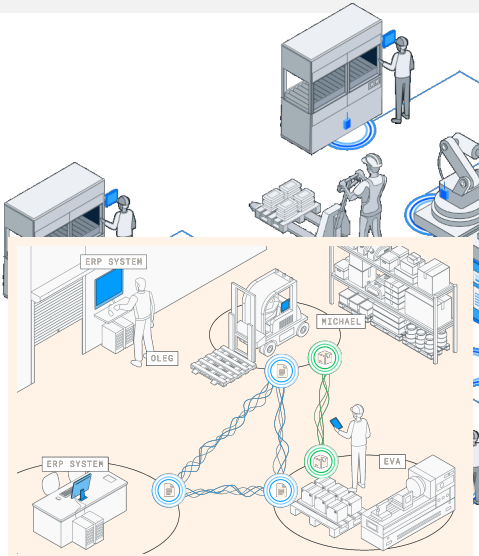
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

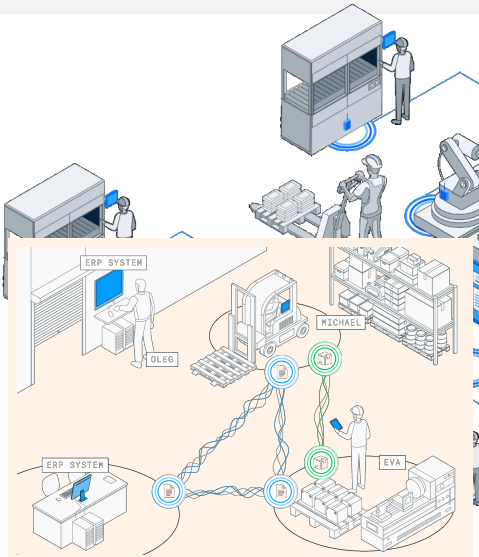
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

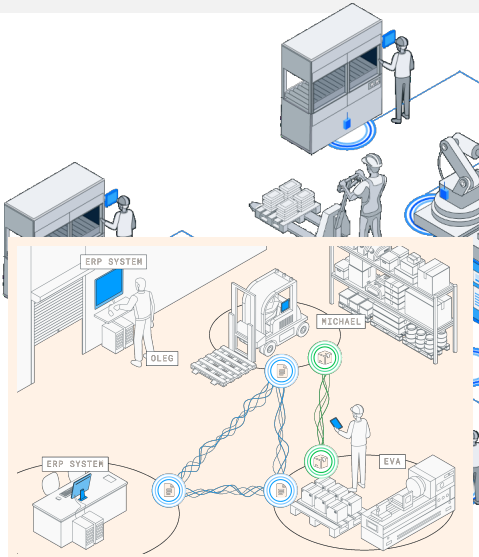
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins
- a log determines the computational state of its twin

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins
- a log determines the computational state of its twin
- replicated logs are merged

Local-first principles

Being partial to availability

Local-first principles

Being partial to availability

Thou shalt be autonomous

Local-first principles

Being partial to availability

Thou shalt be autonomous
Thou shalt collaborate

Local-first principles

Being partial to availability

Thou shalt be autonomous
Thou shalt collaborate
Thou shalt recognise and embrace conflicts

Local-first principles

Being partial to availability

- Thou shalt be autonomous
- Thou shalt collaborate
- Thou shalt recognise and embrace conflicts
- Thou shalt resolve conflicts

Local-first principles

Being partial to availability

- Thou shalt be autonomous
- Thou shalt collaborate
- Thou shalt recognise and embrace conflicts
- Thou shalt resolve conflicts
- Thou shalt be consistent eventually

Local-first principles

Being partial to availability

Thou shalt be autonomous
Thou shalt collaborate
Thou shalt recognise and embrace conflicts
Thou shalt resolve conflicts
Thou shalt be consistent eventually

Challenges

Specify application-level protocols where decisions

Local-first principles

Being partial to availability

Thou shalt be autonomous
Thou shalt collaborate
Thou shalt recognise and embrace conflicts
Thou shalt resolve conflicts
Thou shalt be consistent eventually

Challenges

Specify application-level protocols where decisions don't require consensus

Local-first principles

Being partial to availability

Thou shalt be autonomous
Thou shalt collaborate
Thou shalt recognise and embrace conflicts
Thou shalt resolve conflicts
Thou shalt be consistent eventually

Challenges

Specify application-level protocols where decisions
don't require consensus
are based on stale local states

Local-first principles

Being partial to availability

Thou shalt be autonomous
Thou shalt collaborate
Thou shalt recognise and embrace conflicts
Thou shalt resolve conflicts
Thou shalt be consistent eventually

Challenges

Specify application-level protocols where decisions
don't require consensus
are based on stale local states
yet, collaboration has to be successful

State of the art [9]

Some assumptions

- peers are not malicious
- peers can progress at all times...even under partial knowledge
- communication infrastructure is reliable
- purity: inconsistencies resolved by "replaying" executions (invertible or compensatable actions)

Research directions

Generalisations dropping any of the assumptions above

Ingredients (I): events & logs

Events e

$src(e)$

Logs $e_1 \cdot e_2 \ldots$

Ingredients (I): events & logs

Events $\quad \vdash \quad e \quad : \quad t$

$$src(e)$$

Logs $\quad \vdash \quad e_1 \cdot e_2 \ldots \quad : \quad t_1 \cdot t_2 \ldots$

Ingredients (I): events & logs

Events $\quad \vdash \quad e \quad : \quad t$

$$src(e)$$

Logs $\quad \vdash \quad e_1 \cdot e_2 \ldots \quad : \quad t_1 \cdot t_2 \ldots$

order induced by $\ell = e_1 \cdots e_n$ $\quad e_i <_\ell e_j \iff i < j$

Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)

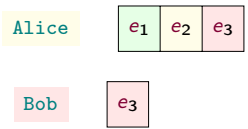
Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)
Events are appended to the logs of machines in **two phases**:

Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)
Events are appended to the logs of machines in **two phases**:

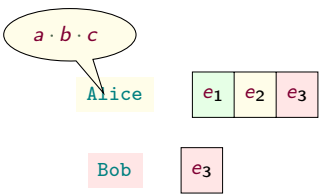
1st Phase: emitted events are appended to the local log of the emitting machine



Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)
Events are appended to the logs of machines in **two phases**:

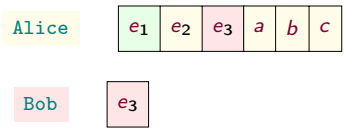
1st Phase: emitted events are appended to the local log of the emitting machine



Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)
Events are appended to the logs of machines in **two phases**:

1st Phase: emitted events are appended to the local log of the emitting machine

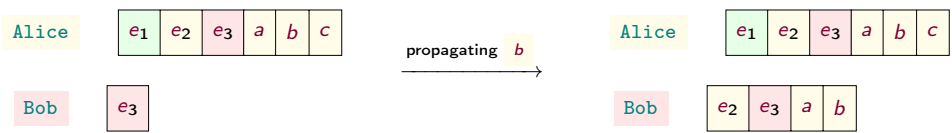


Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)
Events are appended to the logs of machines in **two phases**:

1st Phase: emitted events are appended to the local log of the emitting machine

2nd Phase: newly emitted events are shipped to other machines

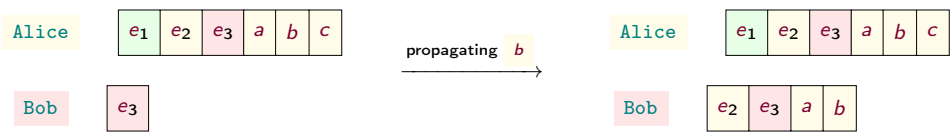


Ingredients (II): log shipping

Machines **emit** logs upon execution of commands (we'll see how in a moment)
Events are appended to the logs of machines in **two phases**:

1st Phase: emitted events are appended to the local log of the emitting machine

2nd Phase: newly emitted events are shipped to other machines



a is shipped too!

Machines' syntax by example

Let's build a machine `InitialP` for coordinating a taxi service.


```
InitialP  =
```

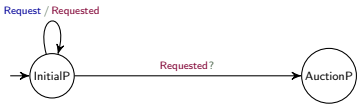
Machines' syntax by example

Let's build a machine `InitialP` for coordinating a taxi service.

$$\texttt{InitialP} \quad = \quad \text{Request} \mapsto \text{Requested} \cdot$$

Machines' syntax by example

Let's build a machine `InitialP` for coordinating a taxi service.

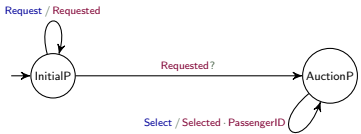


$$\texttt{InitialP} \quad = \quad \text{Request} \mapsto \text{Requested} \cdot [\text{Requested}?\,\underline{\texttt{AuctionP}}]$$

$$\texttt{AuctionP} \quad =$$

Machines' syntax by example

Let's build a machine `InitialP` for coordinating a taxi service.

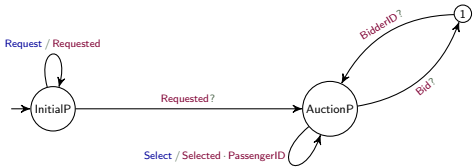


$$\texttt{InitialP} \quad = \quad \text{Request} \mapsto \text{Requested} \cdot \, [\text{Requested}?\, \underline{\texttt{AuctionP}}]$$

$$\texttt{AuctionP} \quad = \quad \text{Select} \mapsto \text{Selected} \cdot \text{PassengerId} \cdot$$

Machines' syntax by example

Let's build a machine `InitialP` for coordinating a taxi service.

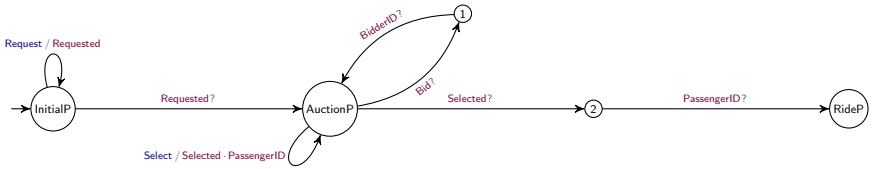


$$\texttt{InitialP} \quad = \quad \text{Request} \mapsto \text{Requested} \cdot \, [\text{Requested}?\, \underline{\texttt{AuctionP}}]$$

$$\texttt{AuctionP} \quad = \quad \text{Select} \mapsto \text{Selected} \cdot \text{PassengerId} \cdot \, [\\ \text{Bid}?\, \underline{\text{BidderId}?\, \texttt{AuctionP}}$$

Machines' syntax by example

Let's build a machine `InitialP` for coordinating a taxi service.



$$
\begin{aligned}
\texttt{InitialP} &= \mathsf{Request} \mapsto \mathsf{Requested} \cdot [\mathsf{Requested}?\,\underline{\texttt{AuctionP}}] \\
\texttt{AuctionP} &= \mathsf{Select} \mapsto \mathsf{Selected} \cdot \mathsf{PassengerId} \cdot [\\
&\qquad \mathsf{Bid}?\,\mathsf{BidderId}?\,\underline{\texttt{AuctionP}} \\
&\qquad \&\\
&\qquad \mathsf{Selected}?\,\mathsf{PassengerId}?\,\underline{\texttt{RideP}} \\
&\quad] \\
\texttt{RideP} &= \cdots
\end{aligned}
$$

Machines' semantics

So, think of $\mathtt{M} = \kappa \cdot [\mathtt{t}_1 ? \mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n ? \mathtt{M}_n]$ as an FSA emitting/consuming events according to its transitions:

- either self-loops (determined by the κ part)
- or event consumption (determined by the guards of the branches $\mathtt{t}_i$)

Machines' semantics

So, think of $\mathtt{M} = \kappa \cdot [\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n]$ as an FSA emitting/consuming events according to its transitions:

- either self-loops (determined by the κ part)
- or event consumption (determined by the guards of the branches $\mathtt{t}_i$)

State transition function:

$$\delta(\mathtt{M}, \epsilon) = \mathtt{M}$$

$$\delta(\mathtt{M}, e \cdot \ell) = \begin{cases} \delta(\mathtt{M}', \ell) & \text{if } \vdash e : \mathtt{t},\ \mathtt{M} \xrightarrow{\mathtt{t}?} \mathtt{M}' \\ \delta(\mathtt{M}, \ell) & \text{otherwise} \end{cases}$$

Machines' semantics

So, think of $\mathtt{M} = \kappa \cdot [\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n]$ as an FSA emitting/consuming events according to its transitions:

- either self-loops (determined by the κ part)
- or event consumption (determined by the guards of the branches $\mathtt{t}_i$)

State transition function:

$$\delta(\mathtt{M}, \epsilon) = \mathtt{M}$$

$$\delta(\mathtt{M}, e \cdot \ell) = \begin{cases} \delta(\mathtt{M}', \ell) & \text{if } \vdash e : \mathtt{t},\ \mathtt{M} \xrightarrow{\mathtt{t}?} \mathtt{M}' \\ \delta(\mathtt{M}, \ell) & \text{otherwise} \end{cases}$$

That is
$\mathtt{M}$ with local log ℓ is in the implicit state $\delta(\mathtt{M}, \ell)$ reached after processing each event in ℓ

Machines' semantics

So, think of $\mathtt{M} = \kappa \cdot [\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n]$ as an FSA emitting/consuming events according to its transitions:

- either self-loops (determined by the κ part)
- or event consumption (determined by the guards of the branches $\mathtt{t}_i$)

State transition function:

$$\delta(\mathtt{M}, \epsilon) = \mathtt{M}$$

$$\delta(\mathtt{M}, e \cdot \ell) = \begin{cases} \delta(\mathtt{M}', \ell) & \text{if } \vdash e : \mathtt{t},\ \mathtt{M} \xrightarrow{\mathtt{t}?} \mathtt{M}' \\ \delta(\mathtt{M}, \ell) & \text{otherwise} \end{cases}$$

That is
$\mathtt{M}$ with local log ℓ is in the implicit state $\delta(\mathtt{M}, \ell)$ reached after processing each event in ℓ

$$\frac{\delta(\mathtt{M}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \delta(\mathtt{M}, \ell) \qquad \ell' \text{ fresh} \qquad \vdash \ell' : \mathtt{1}}{(\mathtt{M}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} (\mathtt{M}, \ell \cdot \ell')}$$

Machines' semantics

So, think of $\mathtt{M} = \kappa \cdot [\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n]$ as an FSA emitting/consuming events according to its transitions:

- either self-loops (determined by the κ part)
- or event consumption (determined by the guards of the branches $\mathtt{t}_i$)

State transition function:

$$\delta(\mathtt{M}, \epsilon) = \mathtt{M}$$

$$\delta(\mathtt{M}, e \cdot \ell) = \begin{cases} \delta(\mathtt{M}', \ell) & \text{if } \vdash e : \mathtt{t},\ \mathtt{M} \xrightarrow{\mathtt{t}?} \mathtt{M}' \\ \delta(\mathtt{M}, \ell) & \text{otherwise} \end{cases}$$

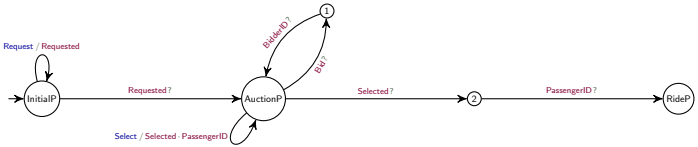
$$\frac{\delta(\mathtt{M}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \delta(\mathtt{M}, \ell) \qquad \ell' \text{ fresh} \qquad \vdash \ell' : \mathtt{1}}{(\mathtt{M}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} (\mathtt{M}, \ell \cdot \ell')}$$

That is
$\mathtt{M}$ with local log ℓ is in the implicit state $\delta(\mathtt{M}, \ell)$ reached after processing each event in ℓ

That is
after processing the events in ℓ, $\mathtt{M}$ reaches a state enabling $\mathtt{c}\,/\,\mathtt{1}$ then the command execution can emit ℓ' of type $\mathtt{1}$ and append it to the local log of $\mathtt{M}$

An example

Take the machine `InitialP` (slide 36) with a local log $\ell = \mathit{ignoreMe} \cdot \mathit{ignoreMeToo}$ where $\nvdash \mathit{ignoreMe} : \mathsf{Requested}$ and $\nvdash \mathit{ignoreMeToo} : \mathsf{Requested}$

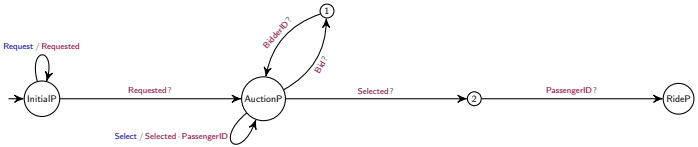


By definition of δ

- $\delta(\texttt{InitialP}, \ell) = \texttt{InitialP}$

An example

Take the machine `InitialP` (slide 36) with a local log $\ell = \mathit{ignoreMe} \cdot \mathit{ignoreMeToo}$ where $\nvdash \mathit{ignoreMe} : \mathrm{Requested}$ and $\nvdash \mathit{ignoreMeToo} : \mathrm{Requested}$

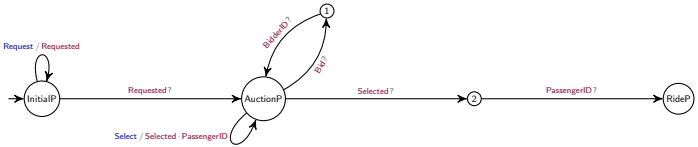


By definition of δ

- $\delta(\texttt{InitialP}, \ell) = \texttt{InitialP}$ hence
- $\delta(\texttt{InitialP}, \ell) \xrightarrow{\text{Request / Requested}} \delta(\texttt{InitialP}, \ell)$

An example

Take the machine `InitialP` (slide 36) with a local log $\ell = \mathit{ignoreMe} \cdot \mathit{ignoreMeToo}$ where $\nvdash \mathit{ignoreMe} : \mathsf{Requested}$ and $\nvdash \mathit{ignoreMeToo} : \mathsf{Requested}$

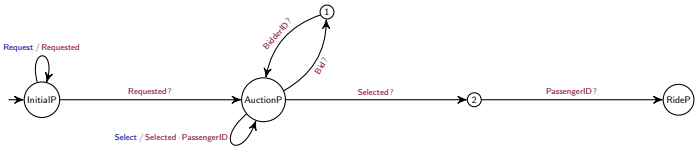


By definition of δ

- $\delta(\texttt{InitialP}, \ell) = \texttt{InitialP}$ hence
- $\delta(\texttt{InitialP}, \ell) \xrightarrow{\text{Request / Requested}} \delta(\texttt{InitialP}, \ell)$ hence
- $(\texttt{InitialP}, \ell) \xrightarrow{\text{Request / Requested}} (\texttt{InitialP}, \ell \cdot \mathit{Requested})$
  with $\vdash \mathit{Requested} : \mathsf{Request}$ and $\mathit{src}(\mathit{Requested}) = \mathtt{P}$

An example

Take the machine `InitialP` (slide 36) with a local log $\ell = \mathit{ignoreMe} \cdot \mathit{ignoreMeToo}$ where $\nvdash \mathit{ignoreMe} : \mathsf{Requested}$ and $\nvdash \mathit{ignoreMeToo} : \mathsf{Requested}$



By definition of δ

- $\delta(\texttt{InitialP}, \ell) = \texttt{InitialP}$ hence
- $\delta(\texttt{InitialP}, \ell) \xrightarrow{\text{Request / Requested}} \delta(\texttt{InitialP}, \ell)$ hence
- $(\texttt{InitialP}, \ell) \xrightarrow{\text{Request / Requested}} (\texttt{InitialP}, \ell \cdot \mathit{Requested})$
  with $\vdash \mathit{Requested} : \mathsf{Request}$ and $\mathit{src}(\mathit{Requested}) = \mathtt{P}$

Exercise

Calculate $\delta(\texttt{InitialP}, \ell \cdot \mathit{Requested})$

Some considerations

Commands are enabled only from the state reached after processing all the events in the local log of the machine

Some considerations

Commands are enabled only from the state reached after processing all the events in the local log of the machine

The environment triggers commands: swarms are inherently non-deterministic!

Some considerations

Commands are enabled only from the state reached after processing all the events in the local log of the machine

The environment triggers commands: swarms are inherently non-deterministic!

We have formalised the emission of events and their consumption
We now focus on the formalisation of log shipping

Swarms

A **swarm (of size n)** is a pair $(\mathtt{S}, \ell)$ where

- $\mathtt{S}$ maps each index $1 \leq i \leq n$ to a pair $(\mathtt{M}_i, \ell_i)$
- ℓ is the (global) log

Swarms

A **swarm (of size n)** is a pair $(\mathtt{S}, \ell)$ where

- $\mathtt{S}$ maps each index $1 \leq i \leq n$ to a pair $(\mathtt{M}_i, \ell_i)$
- ℓ is the (global) log

Notation

$\mathtt{M}_1 \boxed{\ell_1} \mid \ldots \mid \mathtt{M}_n \boxed{\ell_n} \mid \ell$

Swarms

A **swarm (of size n)** is a pair $(\mathtt{S}, \ell)$ where

- $\mathtt{S}$ maps each index $1 \leq i \leq n$ to a pair $(\mathtt{M}_i, \ell_i)$
- ℓ is the (global) log

Notation

$\mathtt{M}_1 \boxed{\ell_1} \mid \ldots \mid \mathtt{M}_n \boxed{\ell_n} \mid \ell$

Disclaimer

Seemingly, we've a contradiction: isn't the global log a centralisation point?

Well...no, it isn't: the global log is just a theoretical ploy!

- it abstracts away from low-level technical details for events' dispatching

Log shipping middlewares rely on timestamp mechanisms (Actyx uses Lamport's timestamps) and guarantee that events are in the same order in all the local logs

Swarms

A **swarm (of size n)** is a pair $(\mathtt{S}, \ell)$ where

- $\mathtt{S}$ maps each index $1 \leq i \leq n$ to a pair $(\mathtt{M}_i, \ell_i)$
- ℓ is the (global) log

Notation

$\mathtt{M}_1 \boxed{\ell_1} \mid \ldots \mid \mathtt{M}_n \boxed{\ell_n} \mid \ell$

Disclaimer

Seemingly, we've a contradiction: isn't the global log a centralisation point?

Well...no, it isn't: the global log is just a theoretical ploy!

- it abstracts away from low-level technical details for events' dispatching
- it elegantly (IOHO) models asynchrony

Swarms

A **swarm (of size n)** is a pair $(\mathtt{S}, \ell)$ where

- $\mathtt{S}$ maps each index $1 \leq i \leq n$ to a pair $(\mathtt{M}_i, \ell_i)$
- ℓ is the (global) log

Notation

$\mathtt{M}_1 \boxed{\ell_1} \mid \ldots \mid \mathtt{M}_n \boxed{\ell_n} \mid \ell$

Disclaimer

Seemingly, we've a contradiction: isn't the global log a centralisation point?

Well...no, it isn't: the global log is just a theoretical ploy!

- it abstracts away from low-level technical details for events' dispatching
- it elegantly (IOHO) models asynchrony
- it is not used in our algorithms and tools

Coherence

A swarm $\mathtt{M}_1\boxed{\ell_1}\mid \ldots \mid \mathtt{M}_n\boxed{\ell_n}\mid \ell$ is **coherent** if $\ell = \bigcup_{1\leq i\leq n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \leq i \leq n$

Coherence

A swarm $\mathtt{M}_1\boxed{\ell_1} \mid \ldots \mid \mathtt{M}_n\boxed{\ell_n} \mid \ell$ is **coherent** if $\ell = \bigcup_{1 \le i \le n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \le i \le n$

where $\ell_1 \sqsubseteq \ell_2$ is the **sublog** relation defined as

- $\ell_1 \subseteq \ell_2$ and $<_{\ell_1} \subseteq <_{\ell_2}$ and

That is
all events of ℓ_1 appear in the same order in ℓ_2

- $e <_{\ell_2} e'$, $src(e) = src(e')$ and $e' \in \ell_1 \implies e \in \ell_1$

That is
the per-source partitions of ℓ_1 are prefixes of the corresponding partitions of ℓ_2

Coherence

A swarm $\mathtt{M}_1\boxed{\ell_1}\,|\,\ldots\,|\,\mathtt{M}_n\boxed{\ell_n}\,|\,\ell$ is **coherent** if $\ell = \bigcup_{1\le i\le n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \le i \le n$

where $\ell_1 \sqsubseteq \ell_2$ is the **sublog** relation defined as

- $\ell_1 \subseteq \ell_2$ and $<_{\ell_1} \subseteq <_{\ell_2}$ and

That is
all events of ℓ_1 appear in the same order in ℓ_2

- $e <_{\ell_2} e'$, $src(e) = src(e')$ and $e' \in \ell_1 \implies e \in \ell_1$

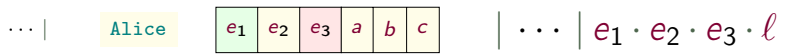
That is
the per-source partitions of ℓ_1 are prefixes of the corresponding partitions of ℓ_2

Hereafter, we assume coherence

Merging logs

Exercise

Recall slide 35 and consider a swarm



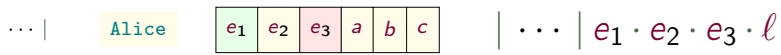
(1)

Under which condition is (1) coherent?

Merging logs

Exercise

Recall slide 35 and consider a swarm



(1)

Under which condition is (1) coherent?

The propagation of newly generated events happens by merging logs:
Log merging: $\ell_1 \bowtie \ell_2 = \{\ell \mid \ell \subseteq \ell_1 \cup \ell_2 \text{ and } \ell_1 \sqsubseteq \ell \text{ and } \ell_2 \sqsubseteq \ell\}$

Semantics of swarms

By rule [Local] below, a command's execution updates both local and global logs

$$\frac{\mathtt{S}(i) = \mathtt{M}\boxed{\ell_i} \qquad \mathtt{M}\boxed{\ell_i} \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathtt{M}\boxed{\ell_i'} \qquad \mathit{src}(\ell_i' \setminus \ell_i) = \{i\} \qquad \ell' \in \ell \bowtie \ell_i'}{(\mathtt{S}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} (\mathtt{S}[i \mapsto \mathtt{M}\boxed{\ell_i'}], \ell')}\,[\text{Local}]$$

Semantics of swarms

By rule [Local] below, a command's execution updates both local and global logs

$$\frac{\mathtt{S}(i) = \mathtt{M}\boxed{\ell_i} \qquad \mathtt{M}\boxed{\ell_i} \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathtt{M}\boxed{\ell_i'} \qquad \mathit{src}(\ell_i' \setminus \ell_i) = \{i\} \qquad \ell' \in \ell \bowtie \ell_i'}{(\mathtt{S}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} (\mathtt{S}[i \mapsto \mathtt{M}\boxed{\ell_i'}], \ell')}\,[\text{Local}]$$

$$\frac{\mathtt{S}(i) = \mathtt{M}\boxed{\ell_i} \qquad \ell_i \sqsubseteq \ell' \sqsubseteq \ell \qquad \ell_i \subset \ell'}{(\mathtt{S}, \ell) \xrightarrow{\tau} (\mathtt{S}[i \mapsto \mathtt{M}\boxed{\ell'}], \ell)}\,[\text{Prop}]$$

By rule [Prop] above, the propagation of events happens

- by shipping a **non-deterministically chosen** subset of events in the global log
- to a **non-deterministically chosen** machine

Semantics at work (I)

If $\mathtt{B}\,\boxed{b} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{B}\,\boxed{b \cdot d \cdot e}$ with $\vdash d \cdot e : \mathtt{1}$

Semantics at work (I)

If $\mathtt{B}\boxed{b} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{B}\boxed{b \cdot d \cdot e}$ with $\vdash d \cdot e : \mathtt{1}$

then, by [Local] $\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b} \mid \mathtt{C}\boxed{c} \mid b \cdot a \cdot c \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell$

Semantics at work (I)

If $\mathtt{B}\boxed{b} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{B}\boxed{b \cdot d \cdot e}$ with $\vdash d \cdot e : \mathtt{1}$

then, by [Local] $\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b} \mid \mathtt{C}\boxed{c} \mid b \cdot a \cdot c \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell$

for all $\ell \in (b \cdot a \cdot c) \bowtie (b \cdot d \cdot e)$

Semantics at work (I)

If $\mathtt{B}\boxed{b} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{B}\boxed{b \cdot d \cdot e}$ with $\vdash d \cdot e : \mathtt{1}$

then, by [Local] $\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b} \mid \mathtt{C}\boxed{c} \mid b \cdot a \cdot c \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell$

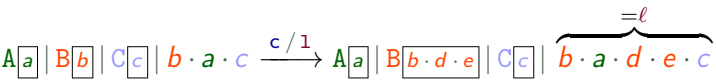
for all $\ell \in (b \cdot a \cdot c) \bowtie (b \cdot d \cdot e)$

Exercise

Compute $(b \cdot a \cdot c) \bowtie (b \cdot d \cdot e)$

Semantics at work (II)

With reference to slide 44, consider



Semantics at work (II)

With reference to slide 44, consider

$$\mathtt{A}\boxed{a}\,|\,\mathtt{B}\boxed{b}\,|\,\mathtt{C}\boxed{c}\,|\,b\cdot a\cdot c \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{A}\boxed{a}\,|\,\mathtt{B}\boxed{b\cdot d\cdot e}\,|\,\mathtt{C}\boxed{c}\,|\,\overbrace{b\cdot a\cdot d\cdot e\cdot c}^{=\ell}$$

By rule [Prop] we can propagate a non-deterministically chosen sublog of $b \cdot d \cdot e$

Semantics at work (II)

With reference to slide 44, consider

$$\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b} \mid \mathtt{C}\boxed{c} \mid b \cdot a \cdot c \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \overbrace{b \cdot a \cdot d \cdot e \cdot c}^{=\ell}$$

By rule [Prop] we can propagate a non-deterministically chosen sublog of $b \cdot d \cdot e$

Let's propagate $d \cdot e$

$$\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell \;\begin{cases} \xrightarrow{\tau} \mathtt{A}\boxed{b \cdot a \cdot d \cdot e} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell \\ \xrightarrow{\tau} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{b \cdot d \cdot e \cdot c} \mid \ell \end{cases}$$

Semantics at work (II)

With reference to slide 44, consider

$$\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b} \mid \mathtt{C}\boxed{c} \mid b \cdot a \cdot c \xrightarrow{\mathtt{c}\,/\,1} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \overbrace{b \cdot a \cdot d \cdot e \cdot c}^{=\ell}$$

By rule [Prop] we can propagate a non-deterministically chosen sublog of $b \cdot d \cdot e$

Let's propagate $d \cdot e$

$$\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell \;\begin{cases} \xrightarrow{\tau} \mathtt{A}\boxed{b \cdot a \cdot d \cdot e} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell \\ \xrightarrow{\tau} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{b \cdot d \cdot e \cdot c} \mid \ell \end{cases}$$

Exercise

In both cases b must be shipped too. Why?
And why is event a not shipped to $\mathtt{C}$ together with the events from $\mathtt{B}$?

Semantics at work (II)

With reference to slide 44, consider

$$\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b} \mid \mathtt{C}\boxed{c} \mid b \cdot a \cdot c \xrightarrow{\mathtt{c}\,/\,1} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \overbrace{b \cdot a \cdot d \cdot e \cdot c}^{=\ell}$$

By rule [Prop] we can propagate a non-deterministically chosen sublog of $b \cdot d \cdot e$

Let's propagate $d \cdot e$

$$\mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell \;\begin{cases} \xrightarrow{\tau} \mathtt{A}\boxed{b \cdot a \cdot d \cdot e} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{c} \mid \ell \\ \xrightarrow{\tau} \mathtt{A}\boxed{a} \mid \mathtt{B}\boxed{b \cdot d \cdot e} \mid \mathtt{C}\boxed{b \cdot d \cdot e \cdot c} \mid \ell \end{cases}$$

Exercise

In both cases b must be shipped too. Why?
And why is event a not shipped to $\mathtt{C}$ together with the events from $\mathtt{B}$?

Exercise

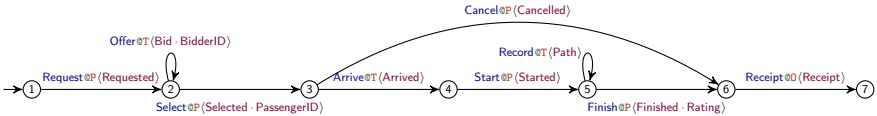
Can we propagate just event e?

Swarm protocols as FSA

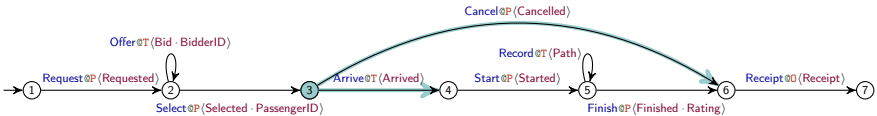
Like for machines, a swarm protocols $\mathsf{G} = \sum_{i \in I} \mathtt{c}_i @ \mathtt{R}_i \langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i$ has an associated FSA:

- the set of states consists of G plus the states in G_i for each $i \in \underline{n}$
- G is the initial state
- for each $i \in I$, G has a transition to state G_i labelled with $\mathtt{c}_i @ \mathtt{R}_i \langle \mathtt{l}_i \rangle$, written $\mathsf{G} \xrightarrow{\mathtt{c}_i \,/\, \mathtt{l}_i} \mathsf{G}_i$

An example


Cancel@P⟨Cancelled⟩
Offer@T⟨Bid · BidderID⟩
Record@T⟨Path⟩
Request@P⟨Requested⟩
Arrive@T⟨Arrived⟩
Start@P⟨Started⟩
Receipt@D⟨Receipt⟩
1
2
3
4
5
6
7
Select@P⟨Selected · PassengerID⟩
Finish@P⟨Finished · Rating⟩

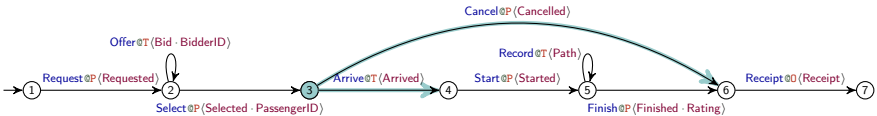
An example



There is a race in state 3!

- the driver of the selected taxi may invoke Arrive
- **while** P loses patience and invokes Cancel

An example



There is a race in state 3!

- the driver of the selected taxi may invoke Arrive
- **while** P loses patience and invokes Cancel

This protocol violates well-formedness conditions typically imposed on behavioural types due to the race in state 3 (because it has two selectors, which is also true of states 2 and 5)

Semantics of swarm protocols

One rule only!

$$\frac{}{(\mathsf{G}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} (\mathsf{G}, \ell \quad)}\ \text{[G-Cmd]}$$

Semantics of swarm protocols

One rule only!

$$\frac{\delta(\mathsf{G}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}'}{(\mathsf{G}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} (\mathsf{G}, \ell \quad)}\ \text{[G-Cmd]}$$

where

$$\delta(\mathsf{G}, \ell) = \begin{cases} \mathsf{G} & \text{if } \ell = \epsilon \\ \delta(\mathsf{G}', \ell'') & \text{if } \mathsf{G} \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}' \text{ and } \vdash \ell' : \mathtt{l} \text{ and } \ell = \ell' \cdot \ell'' \\ \bot & \text{otherwise} \end{cases}$$

Logs to be consumed "atomically", hence $\delta(\mathsf{G}, \ell)$ may be undefined

Semantics of swarm protocols

One rule only!

$$\frac{\delta(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathsf{G}' \qquad \vdash \ell' : \mathtt{1} \qquad \ell' \text{ log of fresh events}}{(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} (\mathsf{G},\ell\cdot\ell')}\text{[G-Cmd]}$$

where

$$\delta(\mathsf{G},\ell) = \begin{cases} \mathsf{G} & \text{if } \ell = \epsilon \\ \delta(\mathsf{G}',\ell'') & \text{if } \mathsf{G} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathsf{G}' \text{ and } \vdash \ell' : \mathtt{1} \text{ and } \ell = \ell'\cdot\ell'' \\ \bot & \text{otherwise} \end{cases}$$

Logs to be consumed "atomically", hence $\delta(\mathsf{G},\ell)$ may be undefined

Semantics of swarm protocols

One rule only!

$$\frac{\delta(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}' \qquad \vdash \ell' : \mathtt{l} \qquad \ell' \text{ log of fresh events}}{(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} (\mathsf{G},\ell\cdot\ell')}\,[\text{G-Cmd}]$$

where

$$\delta(\mathsf{G},\ell) = \begin{cases} \mathsf{G} & \text{if } \ell = \epsilon \\ \delta(\mathsf{G}',\ell'') & \text{if } \mathsf{G} \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}' \text{ and } \vdash \ell' : \mathtt{l} \text{ and } \ell = \ell'\cdot\ell'' \\ \bot & \text{otherwise} \end{cases}$$

Logs to be consumed "atomically", hence $\delta(\mathsf{G},\ell)$ may be undefined

We restrict ourselves to **deterministic** swarm protocols that is, on different transitions from a same state, we require that

- log types start differently — **log determinism**
- pairs (command,role) differ — **command determinism**

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

Each machine plays one role

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting G on each role

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting G on each role

First attempt

$$\left(\sum_{i \in I} \mathtt{c}_i @ \mathtt{R}_i \langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i\right) \downarrow_{\mathtt{R}} = \kappa \cdot [\&_{i \in I}\, \mathtt{l}_i ?\, \mathsf{G}_i \downarrow_{\mathtt{R}}]$$

where $\kappa = \{(\mathtt{c}_i \,/\, \mathtt{l}_i) \mid \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$

From swarm protocols to machines

Transitions of a swarm protocol $\mathtt{G}$ are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting $\mathtt{G}$ on each role

First attempt

$$\Big(\sum_{i\in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i\rangle \,.\, \mathtt{G}_i\Big)\downarrow_{\mathtt{R}} = \kappa \cdot [\&_{i\in I}\, \mathtt{l}_i?\, \mathtt{G}_i\downarrow_{\mathtt{R}}]$$

where $\kappa = \{(\mathtt{c}_i \,/\, \mathtt{l}_i) \;\big|\; \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$

simple, but

- projected machines are large in all but the most trivial cases
- processing **all** events is undesirable: security and efficiency

Another attempt

Let's subscribe to **subscriptions**: maps from roles to sets of event types

In pub-sub, processes subscribe to "topics"

Another attempt

Let's subscribe to **subscriptions**: maps from roles to sets of event types

In pub-sub, processes subscribe to "topics"

Given $\mathsf{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i$, the **projection of G on a role $\mathtt{R}$ with respect to subscription σ** is

$$\mathsf{G} \downarrow^{\sigma}_{\mathtt{R}} = \kappa \cdot [\&_{j \in J} \, \mathsf{filter}(\mathtt{l}_j, \sigma(\mathtt{R}))\, ?\, \mathsf{G}_j \downarrow^{\sigma}_{\mathtt{R}}] \qquad \text{where}$$

Another attempt

Let's subscribe to **subscriptions**: maps from roles to sets of event types

In pub-sub, processes subscribe to "topics"

Given $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$, the **projection of $\mathtt{G}$ on a role $\mathtt{R}$ with respect to subscription σ** is

$$\mathtt{G} \downarrow^{\sigma}_{\mathtt{R}} = \kappa \cdot [\&_{j \in J} \, \mathsf{filter}(\mathtt{l}_j, \sigma(\mathtt{R}))? \, \mathtt{G}_j \downarrow^{\sigma}_{\mathtt{R}}] \qquad \text{where}$$

$$\kappa = \{\mathtt{c}_i \,/\, \mathtt{l}_i \mid \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$$

$$J = \{i \in I \mid \mathsf{filter}(\mathtt{l}_i, \sigma(\mathtt{R})) \neq \epsilon\}$$

$$\mathsf{filter}(\mathtt{l}, E) = \begin{cases} \epsilon, & \text{if } \mathtt{t} = \epsilon \\ \mathtt{t} \cdot \mathsf{filter}(\mathtt{l}', E) & \text{if } \mathtt{t} \in E \text{ and } \mathtt{l} = \mathtt{t} \cdot \mathtt{l}' \\ \mathsf{filter}(\mathtt{l}, E) & \text{otherwise} \end{cases}$$

Well-formedness: sufficient conditions for well-behaviour

Transitory deviations are tolerated provided that consistency is eventually recovered

Well-formedness: sufficient conditions for well-behaviour

Transitory deviations are tolerated provided that consistency is eventually recovered

Example

`T` may bid after `P` has made their decision if the selection event `T` has not yet been received.

This inconsistency is temporary: when the selection event reaches `T` this inconsistency is recognised and resolved

Well-formedness

Trading consistency for availability has implications:

Well-formedness = Causality

Trading consistency for availability has implications:

Propagation of events is non-atomic (cf. rule [Prop])

$\Longrightarrow$ differences in how machines perceive the (state of the) computation

Causality

Fix a subscription σ. For each branch $i \in I$ of $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i @ \mathtt{R}_i \langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$

Explicit re-enabling $\quad \sigma(\mathtt{R}_i) \cap \mathtt{l}_i \neq \emptyset$

If $\mathtt{R}$ should have a command enabled after $\mathtt{c}_i$ then $\sigma(\mathtt{R})$ contains some event type emitted by $\mathtt{c}_i$

Command causality $\quad$ if $\mathtt{R}$ executes a command in $\mathtt{G}_i$ then $\sigma(\mathtt{R}) \cap \mathtt{l}_i \neq \emptyset$ and $\sigma(\mathtt{R}) \cap \mathtt{l}_i \supseteq \bigcup_{\mathtt{R}' \in_\sigma \mathtt{G}_i} \sigma(\mathtt{R}') \cap \mathtt{l}_i$

Well-formedness = Causality + Determinacy

Trading consistency for availability has implications:

Propagation of events is non-atomic (cf. rule [Prop])

$\Longrightarrow$ different roles may take inconsistent decisions

Causality & Determinacy

Fix a subscription σ. For each branch $i \in I$ of $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$

Explicit re-enabling $\quad \sigma(\mathtt{R}_i) \cap \mathtt{l}_i \neq \emptyset$

Command causality $\quad$ if $\mathtt{R}$ executes a command in $\mathtt{G}_i$
then $\sigma(\mathtt{R}) \cap \mathtt{l}_i \neq \emptyset$ and $\sigma(\mathtt{R}) \cap \mathtt{l}_i \supseteq \bigcup_{\mathtt{R}' \in_\sigma \mathtt{G}_i} \sigma(\mathtt{R}') \cap \mathtt{l}_i$

Determinacy $\quad \mathtt{R} \in_\sigma \mathtt{G}_i \implies \mathtt{l}_i[0] \in \sigma(\mathtt{R})$

Well-formedness = Causality + Determinacy - Confusion

Trading consistency for availability has implications:

Propagation of events is non-atomic (cf. rule [Prop])

$\Longrightarrow$ branches unambiguously identified and events emitted on eventually discharged branches ignored

Causality & Determinacy & Confusion freeness

Fix a subscription σ. For each branch $i \in I$ of $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$

Explicit re-enabling	$\sigma(\mathtt{R}_i) \cap \mathtt{l}_i \neq \emptyset$
Command causality	if $\mathtt{R}$ executes a command in $\mathtt{G}_i$ then $\sigma(\mathtt{R}) \cap \mathtt{l}_i \neq \emptyset$ and $\sigma(\mathtt{R}) \cap \mathtt{l}_i \supseteq \bigcup_{\mathtt{R}' \in_\sigma \mathtt{G}_i} \sigma(\mathtt{R}') \cap \mathtt{l}_i$
Determinacy	$\mathtt{R} \in_\sigma \mathtt{G}_i \implies \mathtt{l}_i[0] \in \sigma(\mathtt{R})$
Confusion freeness	there is a unique subtree $\mathtt{G}'$ of $\mathtt{G}$ emitting $\mathtt{t}$ for each $\mathtt{t}$ starting a log emitted by a command in $\mathtt{G}$

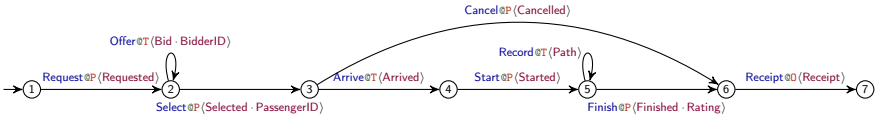
– Behavioural Types for Swarms –

Swarm protocols as FSA

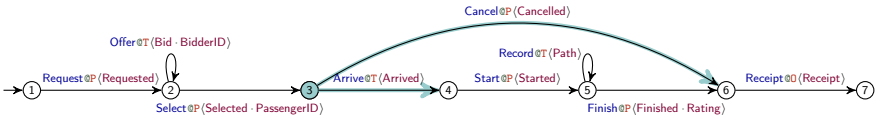
Like for machines, a swarm protocols $\mathsf{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i$ has an associated FSA:

- the set of states consists of G plus the states in G_i for each $i \in \underline{n}$
- G is the initial state
- for each $i \in I$, G has a transition to state G_i labelled with $\mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle$, written $\mathsf{G} \xrightarrow{\mathtt{c}_i \,/\, \mathtt{l}_i} \mathsf{G}_i$

An example


Cancel@P⟨Cancelled⟩
Offer@T⟨Bid · BidderID⟩
Record@T⟨Path⟩
Request@P⟨Requested⟩
Arrive@T⟨Arrived⟩
Start@P⟨Started⟩
Receipt@D⟨Receipt⟩
1
2
3
4
5
6
7
Select@P⟨Selected · PassengerID⟩
Finish@P⟨Finished · Rating⟩

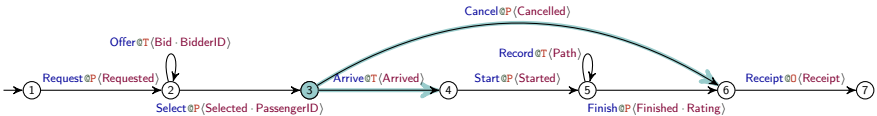
An example



There is a race in state 3!

- the driver of the selected taxi may invoke Arrive
- **while** P loses patience and invokes Cancel

An example



There is a race in state 3!

- the driver of the selected taxi may invoke Arrive
- **while** P loses patience and invokes Cancel

This protocol violates well-formedness conditions typically imposed on behavioural types due to the race in state 3 (because it has two selectors, which is also true of states 2 and 5)

Semantics of swarm protocols

One rule only!

$$\frac{}{(G, \ell) \xrightarrow{\texttt{c} \,/\, \texttt{1}} (G, \ell \quad)} \text{[G-Cmd]}$$

Semantics of swarm protocols

One rule only!

$$\frac{\delta(\mathsf{G}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}'}{(\mathsf{G}, \ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} (\mathsf{G}, \ell \quad)} \text{[G-Cmd]}$$

where

$$\delta(\mathsf{G}, \ell) = \begin{cases} \mathsf{G} & \text{if } \ell = \epsilon \\ \delta(\mathsf{G}', \ell'') & \text{if } \mathsf{G} \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}' \text{ and } \vdash \ell' : \mathtt{l} \text{ and } \ell = \ell' \cdot \ell'' \\ \bot & \text{otherwise} \end{cases}$$

Logs to be consumed "atomically", hence $\delta(\mathsf{G}, \ell)$ may be undefined

Semantics of swarm protocols

One rule only!

$$\frac{\delta(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathsf{G}' \qquad \vdash \ell' : \mathtt{1} \qquad \ell' \text{ log of fresh events}}{(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} (\mathsf{G},\ell\cdot\ell')}\ \text{[G-Cmd]}$$

where

$$\delta(\mathsf{G},\ell) = \begin{cases} \mathsf{G} & \text{if } \ell = \epsilon \\ \delta(\mathsf{G}',\ell'') & \text{if } \mathsf{G} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathsf{G}' \text{ and } \vdash \ell' : \mathtt{1} \text{ and } \ell = \ell'\cdot\ell'' \\ \bot & \text{otherwise} \end{cases}$$

Logs to be consumed "atomically", hence $\delta(\mathsf{G},\ell)$ may be undefined

Semantics of swarm protocols

One rule only!

$$\frac{\delta(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathsf{G}' \qquad \vdash \ell' : \mathtt{1} \qquad \ell' \text{ log of fresh events}}{(\mathsf{G},\ell) \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} (\mathsf{G},\ell\cdot\ell')}\,[\text{G-Cmd}]$$

where

$$\delta(\mathsf{G},\ell) = \begin{cases} \mathsf{G} & \text{if } \ell = \epsilon \\ \delta(\mathsf{G}',\ell'') & \text{if } \mathsf{G} \xrightarrow{\mathtt{c}\,/\,\mathtt{1}} \mathsf{G}' \text{ and } \vdash \ell' : \mathtt{1} \text{ and } \ell = \ell'\cdot\ell'' \\ \bot & \text{otherwise} \end{cases}$$

Logs to be consumed "atomically", hence $\delta(\mathsf{G},\ell)$ may be undefined

We restrict ourselves to **deterministic** swarm protocols that is, on different transitions from a same state, we require that

- log types start differently — **log determinism**
- pairs (command,role) differ — **command determinism**

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

Each machine plays one role

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting G on each role

From swarm protocols to machines

Transitions of a swarm protocol G are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting G on each role

First attempt

$$\Big(\sum_{i \in I} \mathsf{c}_i @ \mathtt{R}_i \langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i\Big) \downarrow_{\mathtt{R}} = \kappa \cdot [\&_{i \in I}\, \mathtt{l}_i ?\, \mathsf{G}_i \downarrow_{\mathtt{R}}]$$

where $\kappa = \{(\mathsf{c}_i \,/\, \mathtt{l}_i) \;\big|\; \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$

From swarm protocols to machines

Transitions of a swarm protocol $\mathtt{G}$ are labelled with a role that may invoke the command

Each machine plays one role

 Obtain machines by projecting $\mathtt{G}$ on each role

First attempt

$$\Big(\sum_{i\in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i\rangle \,.\, \mathtt{G}_i\Big)\downarrow_{\mathtt{R}} = \kappa \cdot [\&_{i\in I}\, \mathtt{l}_i?\, \mathtt{G}_i\downarrow_{\mathtt{R}}]$$

where $\kappa = \{(\mathtt{c}_i \,/\, \mathtt{l}_i) \;\big|\; \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$

simple, but

- projected machines are large in all but the most trivial cases
- processing **all** events is undesirable: security and efficiency

Another attempt

Let's subscribe to **subscriptions**: maps from roles to sets of event types

In pub-sub, processes subscribe to "topics"

Another attempt

Let's subscribe to **subscriptions**: maps from roles to sets of event types

In pub-sub, processes subscribe to "topics"

Given $\mathsf{G} = \sum_{i \in I} \mathtt{c}_i\mathtt{@R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i$, the **projection of G on a role $\mathtt{R}$ with respect to subscription σ** is

$$\mathsf{G} \downarrow^{\sigma}_{\mathtt{R}} = \kappa \cdot [\&_{j \in J}\, \mathsf{filter}(\mathtt{l}_j, \sigma(\mathtt{R}))\,?\,\mathsf{G}_j \downarrow^{\sigma}_{\mathtt{R}}] \qquad \text{where}$$

Another attempt

Let's subscribe to **subscriptions**: maps from roles to sets of event types

In pub-sub, processes subscribe to "topics"

Given $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$, the **projection of $\mathtt{G}$ on a role $\mathtt{R}$ with respect to subscription σ** is

$$\mathtt{G} \downarrow^{\sigma}_{\mathtt{R}} = \kappa \cdot [\&_{j \in J} \, \mathsf{filter}(\mathtt{l}_j, \sigma(\mathtt{R}))? \, \mathtt{G}_j \downarrow^{\sigma}_{\mathtt{R}}] \qquad \text{where}$$

$$\kappa = \{\mathtt{c}_i \,/\, \mathtt{l}_i \mid \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$$

$$J = \{i \in I \mid \mathsf{filter}(\mathtt{l}_i, \sigma(\mathtt{R})) \neq \epsilon\}$$

$$\mathsf{filter}(\mathtt{l}, E) = \begin{cases} \epsilon, & \text{if } \mathtt{t} = \epsilon \\ \mathtt{t} \cdot \mathsf{filter}(\mathtt{l}', E) & \text{if } \mathtt{t} \in E \text{ and } \mathtt{l} = \mathtt{t} \cdot \mathtt{l}' \\ \mathsf{filter}(\mathtt{l}, E) & \text{otherwise} \end{cases}$$

Well-formedness: sufficient conditions for well-behaviour

Transitory deviations are tolerated provided that consistency is eventually recovered

Well-formedness: sufficient conditions for well-behaviour

Transitory deviations are tolerated provided that consistency is eventually recovered

Example

`T` may bid after `P` has made their decision if the selection event `T` has not yet been received.

This inconsistency is temporary: when the selection event reaches `T` this inconsistency is recognised and resolved

Well-formedness

Trading consistency for availability has implications:

Well-formedness = Causality

Trading consistency for availability has implications:

Propagation of events is non-atomic (cf. rule [Prop])

$\Longrightarrow$ differences in how machines perceive the (state of the) computation

Causality

Fix a subscription σ. For each branch $i \in I$ of $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i @ \mathtt{R}_i \langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$

Explicit re-enabling $\quad \sigma(\mathtt{R}_i) \cap \mathtt{l}_i \neq \emptyset$

If $\mathtt{R}$ should have a command enabled after $\mathtt{c}_i$ then $\sigma(\mathtt{R})$ contains some event type emitted by $\mathtt{c}_i$

Command causality $\quad$ if $\mathtt{R}$ executes a command in $\mathtt{G}_i$ then $\sigma(\mathtt{R}) \cap \mathtt{l}_i \neq \emptyset$ and $\sigma(\mathtt{R}) \cap \mathtt{l}_i \supseteq \bigcup_{\mathtt{R}' \in_\sigma \mathtt{G}_i} \sigma(\mathtt{R}') \cap \mathtt{l}_i$

Well-formedness = Causality + Determinacy

Trading consistency for availability has implications:

Propagation of events is non-atomic (cf. rule [Prop])

$\Longrightarrow$ different roles may take inconsistent decisions

Causality & Determinacy

Fix a subscription σ. For each branch $i \in I$ of $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$

Explicit re-enabling $\quad \sigma(\mathtt{R}_i) \cap \mathtt{l}_i \neq \emptyset$

Command causality $\quad$ if $\mathtt{R}$ executes a command in $\mathtt{G}_i$
then $\sigma(\mathtt{R}) \cap \mathtt{l}_i \neq \emptyset$ and $\sigma(\mathtt{R}) \cap \mathtt{l}_i \supseteq \bigcup_{\mathtt{R}' \in_\sigma \mathtt{G}_i} \sigma(\mathtt{R}') \cap \mathtt{l}_i$

Determinacy $\quad \mathtt{R} \in_\sigma \mathtt{G}_i \implies \mathtt{l}_i[0] \in \sigma(\mathtt{R})$

Well-formedness = Causality + Determinacy - Confusion

Trading consistency for availability has implications:

Propagation of events is non-atomic (cf. rule [Prop])

$\Longrightarrow$ branches unambiguously identified and events emitted on eventually discharged branches ignored

Causality & Determinacy & Confusion freeness

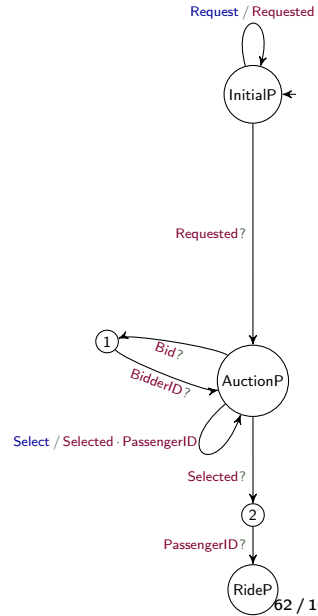
Fix a subscription σ. For each branch $i \in I$ of $\mathtt{G} = \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathtt{G}_i$

Explicit re-enabling	$\sigma(\mathtt{R}_i) \cap \mathtt{l}_i \neq \emptyset$
Command causality	if $\mathtt{R}$ executes a command in $\mathtt{G}_i$ then $\sigma(\mathtt{R}) \cap \mathtt{l}_i \neq \emptyset$ and $\sigma(\mathtt{R}) \cap \mathtt{l}_i \supseteq \bigcup_{\mathtt{R}' \in_\sigma \mathtt{G}_i} \sigma(\mathtt{R}') \cap \mathtt{l}_i$
Determinacy	$\mathtt{R} \in_\sigma \mathtt{G}_i \implies \mathtt{l}_i[0] \in \sigma(\mathtt{R})$
Confusion freeness	there is a unique subtree $\mathtt{G}'$ of $\mathtt{G}$ emitting $\mathtt{t}$ for each $\mathtt{t}$ starting a log emitted by a command in $\mathtt{G}$

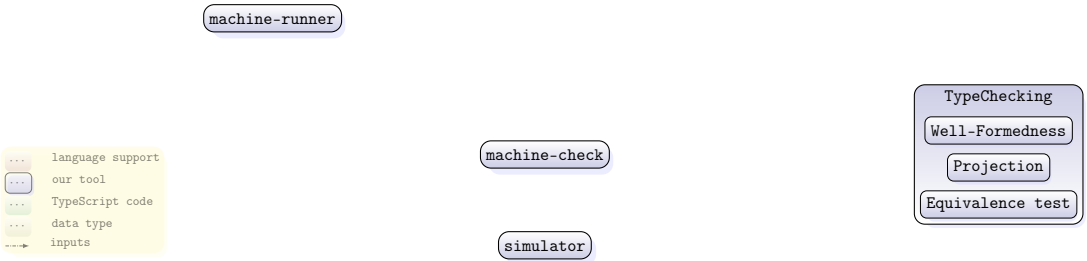
– Tooling –

```ts
// analogous for other events; "type" property matches type name (checked by tool)
type Requested = { type: 'Requested'; pickup: string; dest: string }
type Events = Requested | Bid | BidderID | Selected | ...

/** Initial state for role P */
@proto('taxiRide') // decorator injects inferred protocol into runtime
export class InitialP extends State<Events> {
  constructor(public id: string) { super() }
  execRequest(pickup: string, dest: string) {
    return this.events({ type: 'Requested', pickup, dest })
  }
  onRequested(ev: Requested) {
    return new AuctionP(this.id, ev.pickup, ev.dest, [])
  }
}
@proto('taxiRide')
export class AuctionP extends State<Events> {
  constructor(public id: string, public pickup: string, public dest: string,
    public bids: BidData[]) { super() }
  onBid(ev1: Bid, ev2: BidderID) {
    const [ price, time ] = ev1
    this.bids.push({ price, time, bidderID: ev2.id })
    return this
  }
  execSelect(taxiId: string) {
    return this.events({ type: 'Selected', taxiID },
                       { type: 'PassengerID', id: this.id })
  }
  onSelected(ev: Selected, id: PassengerID) {
    return new RideP(this.id, ev.taxiID)
  }
}
@proto('taxiRide')
export class RideP extends State<Events> { ... }
```

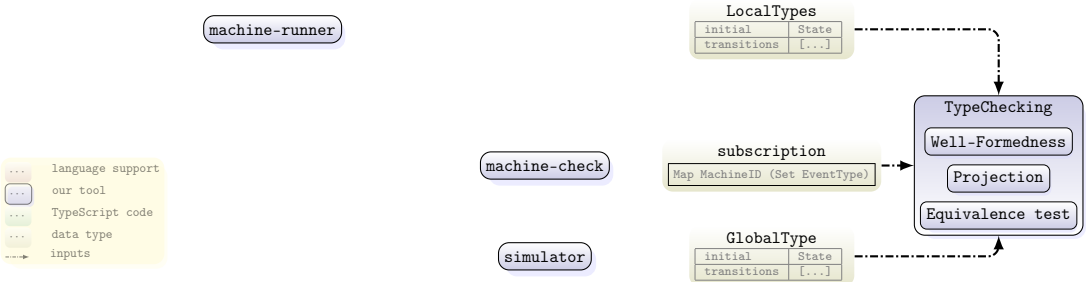


Architecture [10]



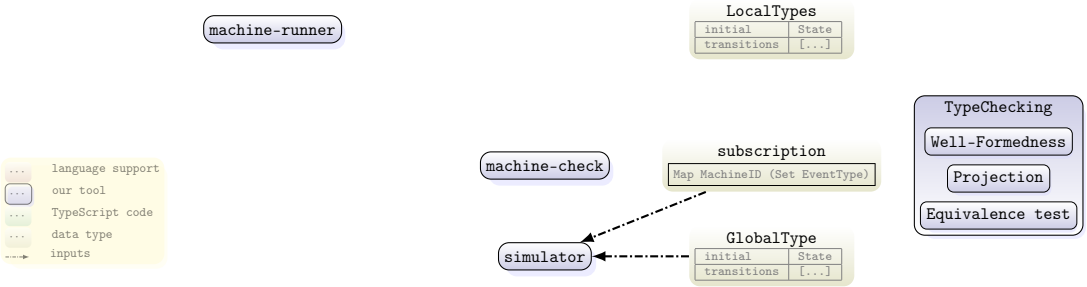
- `TypeChecking` implements the functionalities of our typing discipline
- `simulator` simulates the semantics of swarm realisations
- `machine-check` and `machine-runner` integrate our framework in the Actyx platform

Architecture [10]



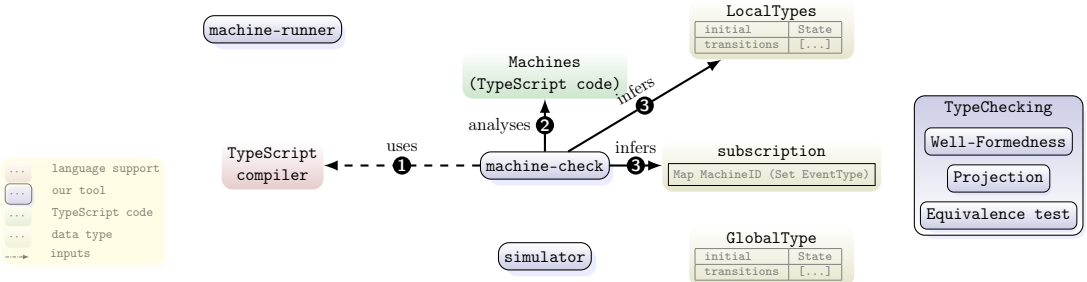
- `TypeChecking` implements the functionalities of our typing discipline
- `simulator` simulates the semantics of swarm realisations
- `machine-check` and `machine-runner` integrate our framework in the Actyx platform

Architecture [10]



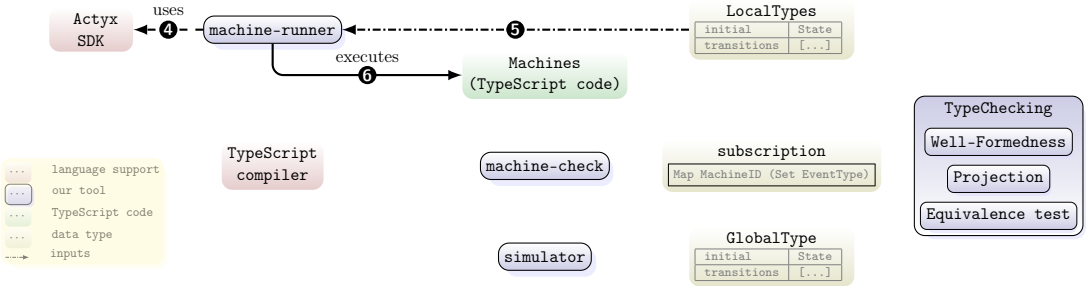
- `TypeChecking` implements the functionalities of our typing discipline
- `simulator` simulates the semantics of swarm realisations
- `machine-check` and `machine-runner` integrate our framework in the Actyx platform

Architecture [10]



- `TypeChecking` implements the functionalities of our typing discipline
- `simulator` simulates the semantics of swarm realisations
- `machine-check` and `machine-runner` integrate our framework in the Actyx platform

Architecture [10]



- `TypeChecking` implements the functionalities of our typing discipline
- `simulator` simulates the semantics of swarm realisations
- `machine-check` and `machine-runner` integrate our framework in the Actyx platform

– Programming Actors –

Erlang's actor model

Erlang's actor model

- Processes execute function (and run on a pre-emptive VM, not OS)

Erlang's actor model

- Processes execute function (and run on a pre-emptive VM, not OS)
- Processes are veeeeery light!

Erlang's actor model

- Processes execute function (and run on a pre-emptive VM, not OS)
- Processes are veeeeery light!
- Context commutation is veeeeery fast!

Erlang's actor model

- Processes execute function (and run on a pre-emptive VM, not OS)
- Processes are veeeeery light!
- Context commutation is veeeeery fast!
- Hence VM can handle maaaaany processes!

Erlang's actor model

- Processes execute function (and run on a pre-emptive VM, not OS)
- Processes are veeeeery light!
- Context commutation is veeeeery fast!
- Hence VM can handle maaaaany processes!
- Program code is shared: no assignment!

Erlang's basics

Processes

- have unique IDs: e.g., Pid = self()
- can be spawned: e.g., Pid = spawn(module, function, arguments)
- Pids to send messages:

  Output: $Pid\,!\{1,2,3\}$

  Input:

```
receive
  {X} when X = 0 -> X+X;
  {X,Y} when X = Y -> Y;
  {X,Y,Z} when X < Z andalso X = Y * Z -> ...
end
```

  clauses process in order, the first enable is executed. If none enable, then wait.

An example

```
-module(TempConverter).
-export([convert/1]).
convert(Helper) ->
  receive
    {Pid, cs, C} ->
      if
        overloaded() -> Helper ! {Pid, cs, C} ;
        true ->  Pid ! {self(), ft, (1.8 * C) + 32};
      end,
      convert()
    ;
    {Pid, ft, F} ->
      if
        overloaded() -> Helper ! {Pid, ft, F} ;
        true -> Pid ! {self(), cs, (F - 32) / 1.8} ;
      end,
      convert();
      stop -> true
    ;
      -> convert()
  end.
```

Erlang programming

Write a client and the helper function.

"The reflexive chemistry" [4, § 3]

$$
\begin{array}{rcl}
P & \triangleq & x\langle \tilde{v} \rangle \\
 & \mid & \textbf{def } D \textbf{ in } P \\
 & \mid & P \mid P
\end{array}
\qquad
\begin{array}{rcl}
J & \triangleq & x\langle \tilde{v} \rangle \\
 & \mid & J \mid J
\end{array}
\qquad
\begin{array}{rcl}
D & \triangleq & J \triangleright P \\
 & \mid & D \wedge D
\end{array}
$$

$\mathcal{R}$eactions $\vdash$ $\mathcal{M}$olecules

$$
\begin{array}{rcl}
\mathcal{R} \vdash P \mid Q & \rightleftharpoons & \mathcal{R} \vdash P, Q \\
D \wedge E \vdash P & \rightleftharpoons & D, E \vdash P \\
D \vdash \textbf{def } E \textbf{ in } P & \rightleftharpoons & D, E\sigma_E \vdash P\sigma_E \\
J \triangleright P \vdash J\sigma_J & \rightarrow & J \triangleright P \vdash P\sigma_J
\end{array}
$$

- σ_E renames vars defined in E with fresh names
- σ_J substitutes the receive vars in J with the values in the corresponding molecules

An "simple" example [4, § 3]

Let's program a memory cells to store some values:

$$
\begin{array}{l}
\textbf{def}\ \mathrm{mkcell}\langle v_0, k\rangle \triangleright \\
\quad \textbf{def} \\
\qquad s\langle v\rangle \mid \mathrm{get}\langle k\rangle \quad \triangleright \quad s\langle v\rangle \mid k\langle v\rangle \\
\qquad s\langle v\rangle \mid \mathrm{set}\langle u, k\rangle \quad \triangleright \quad s\langle u\rangle \mid k\langle\rangle \\
\quad \textbf{in} \\
\qquad s\langle v_0\rangle \mid k\langle \mathrm{get}, \mathrm{set}\rangle \\
\textbf{in} \\
\quad \mathrm{mkcell}\langle 3, c\rangle
\end{array}
$$

An "simple" example [4, § 3]

Let's program a memory cells to store some values:

$$
\begin{array}{l}
\textbf{def}\ \mathrm{mkcell}\langle v_0, k\rangle \triangleright \\
\quad \textbf{def} \\
\qquad s\langle v\rangle \mid \mathrm{get}\langle k\rangle \quad \triangleright \quad s\langle v\rangle \mid k\langle v\rangle \\
\qquad s\langle v\rangle \mid \mathrm{set}\langle u, k\rangle \quad \triangleright \quad s\langle u\rangle \mid k\langle\rangle \\
\quad \textbf{in} \\
\qquad s\langle 3\rangle \mid c\langle \mathrm{get}, \mathrm{set}\rangle \\
\textbf{in} \\
\quad \mathrm{mkcell}\langle 3, c\rangle
\end{array}
$$

An "simple" example [4, § 3]

Let's program a memory cells to store some values:

$$
\begin{array}{l}
\textbf{def}\ \mathrm{mkcell}\langle v_0, k\rangle\triangleright \\
\quad \textbf{def} \\
\qquad s\langle v\rangle \mid \mathrm{get}\langle k\rangle \quad \triangleright \quad s\langle v\rangle \mid k\langle v\rangle \\
\qquad s\langle v\rangle \mid \mathrm{set}\langle u, k\rangle \quad \triangleright \quad s\langle u\rangle \mid k\langle\rangle \\
\quad \textbf{in} \\
\qquad s\langle v_0\rangle \mid k\langle \mathrm{get}, \mathrm{set}\rangle \\
\textbf{in} \\
\quad \mathrm{mkcell}\langle 3, c\rangle
\end{array}
$$

Exercise

Extend the program above so that after the invocation to mkcell the cell is set to 0.

"I have this terrible feeling of déjà vu..."

Doesn't

$$\textbf{def } x_1\langle \tilde{v}_1 \rangle \mid x_2\langle \tilde{v} \rangle_2 \mid \cdots \textbf{ in } P$$

remind you something?

"I have this terrible feeling of déjà vu..."

Doesn't

$$\textbf{def } x_1\langle \tilde{v}_1 \rangle \mid x_2\langle \tilde{v} \rangle_2 \mid \cdots \textbf{ in } P$$

remind you something? e.g.,

$$\textbf{let } x_1 = E_1 \textbf{ in let } x_2 = E_2 \cdots \textbf{ in } E$$

$\lambda \to$ Join?

Recall: $M \triangleq x \mid \lambda x.M \mid MM$

Call-by-name

Leftmost-order reduction strategy & no reduction under λ:

$$
\begin{array}{rcl}
[\![x]\!]_v & \triangleq & v\langle x\rangle \\
[\![\lambda x.M]\!]_v & \triangleq & \textbf{def } k\langle x, m\rangle \triangleright [\![M]\!]_m \textbf{ in } v\langle k\rangle \\
[\![MN]\!]_v & \triangleq & \textbf{def } x\langle n\rangle \triangleright [\![N]\!]_n \textbf{ in} \\
 & & \quad \textbf{def } m\langle k\rangle \triangleright k\langle x, v\rangle \textbf{ in } [\![M]\!]_m
\end{array}
$$

Intuition: a λ-expression is a join process that sends the result of its computation

$\lambda \to$ Join?

Recall: $M \triangleq x \mid \lambda x.M \mid MM$

Call-by-name

Leftmost-order reduction strategy & no reduction under λ:

$$
\begin{array}{rcl}
[\![x]\!]_v & \triangleq & v\langle x\rangle \\
[\![\lambda x.M]\!]_v & \triangleq & \textbf{def } k\langle x, m\rangle \triangleright [\![M]\!]_m \textbf{ in } v\langle k\rangle \\
[\![MN]\!]_v & \triangleq & \textbf{def } x\langle n\rangle \triangleright [\![N]\!]_n \textbf{ in} \\
 & & \quad \textbf{def } m\langle k\rangle \triangleright k\langle x, v\rangle \textbf{ in } [\![M]\!]_m
\end{array}
$$

Intuition: a λ-expression is a join process that sends the result of its computation
Some magic ☺

$$
[\![MN]\!]_v \triangleq \textbf{def } m\langle k\rangle \mid n\langle k'\rangle \triangleright k\langle k', v\rangle \textbf{ in } [\![M]\!]_m \mid [\![N]\!]_n
$$

A problem in concurrency [12]

Problem Definition

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may **never** get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

A Solution

The solution that has worked best over the years, and also appears to be the simplest, is written using C statements and pseudo-code. (Constants are also used in case the number of reindeer were to change, or if the group size of "solution-seeking" elves is modified.) Basically, the reindeer arrive, update the count of how many have arrived, and the last one wakes up Santa. An elf, upon discovering a problem, attempts to modify the count for the number of elves with a problem and either: waits outside Santa's shop if he/she is the first or second such elf; knocks on the door and wakes up Santa if that elf is the third one; or waits in the elves' shop until the elves currently with Santa start coming back. (The code for this solution can be found in the Appendix.)

```
1  receive
2      {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3}
3        and {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6}
4        and {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
5          io:format("Ho, ho, ho! Let's deliver presents!~n"),
6          [Pid1, Pid2, Pid3, Pid4, Pid5, Pid6, Pid7, Pid8, Pid9];
7      {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
8          io:format("Ho, ho, ho! Let's discuss R&D possibilities!~n"),
9          [Pid1, Pid2, Pid3]
10 end
```

The solution with semaphores takes about 2 pages of C code [12]!

Where're join patterns? [Dagstuhl 24051, Jan-Feb 2024]

Where're join patterns? [Dagstuhl 24051, Jan-Feb 2024]

Where're join patterns? [Dagstuhl 24051, Jan-Feb 2024]

Where're join patterns? [Dagstuhl 24051, Jan-Feb 2024]

Where're join patterns? [Dagstuhl 24051, Jan-Feb 2024]

The remaining slides of this lecture are courtesy of Ayman Hussein.

– Formalising market models –

Today's menu

- Market model design is tricky
  - CLOB issues
  - FBA solution
  - FBA issues

Today's menu

- Market model design is tricky
  - CLOB issues
  - FBA solution
  - FBA issues
- **C**oncurrent e**X**change (**CX**)
  - A new concurrent model
    - concurrency: what can be executed independently?

Today's menu

- Market model design is tricky
  - CLOB issues
  - FBA solution
  - FBA issues
- **C**oncurrent e**X**change (**CX**)
  - A new concurrent model
    - concurrency: what can be executed independently?
    - parallelism: what can be executed at the same time?

Today's menu

- Market model design is tricky
  - CLOB issues
  - FBA solution
  - FBA issues
- **C**oncurrent e**X**change (**CX**)
  - A new concurrent model
    - concurrency: what can be executed independently?
    - parallelism: what can be executed at the same time?

    The latter requires the former, not viceversa!
  - A formal definition of **CX** ... with a lot of hand-weaving ☺
  - Supported by a prototype tool for simulations

Take-away message

CX: defined in a model of concurrency

Take-away message

CX: defined in a model of concurrency

- is more "natural" than CLOB (& FBA)

Take-away message

CX: defined in a model of concurrency

- is more "natural" than CLOB (& FBA)
- is white-box: rules precisely defined and applied

Take-away message

CX: defined in a model of concurrency

- is more "natural" than CLOB (& FBA)
- is white-box: rules precisely defined and applied
- can certify desirable properties (or show absence of undesirable ones)

Take-away message

CX: defined in a model of concurrency

- is more "natural" than CLOB (& FBA)
- is white-box: rules precisely defined and applied
- can certify desirable properties (or show absence of undesirable ones)

There is a prototype ... which can also be shown to be correct

Is our approach worthwhile?

Is our approach worthwhile?

JEL Codes: D47, G10, G12, G14

"The market is rigged." —Michael Lewis, *Flash Boys* (Lewis 2014)

"Widespread latency arbitrage is a myth." —Bill Harts, CEO of the Modern Markets Initiative, a high-frequency trading (HFT) lobbyist (Michaels 2016)

Trading floor and open outcry

CBOT "The Pit" in 1908

The New York stock exchange trading floor in September 1963, before the introduction of electronic readouts and computer screens

Open outcry "pit" at the Chicago Board of Trade (CBOT) in 1993

- Human-mediated market interaction (open outcry)
- . . . humans performed order-matching and agreed on trades

Towards electronic systems

- 1971: NASDAQ introduced elements of electronic infrastructure
- 1977: Toronto Stock Exchange
  - First operational electronic limit order book system in the world
  - Initially used only for less liquid stocks, not the entire main market
- 1986: Paris Bourse: starts using CLOB as primary mechanism
- 1990s–2000s: NYSE, NASDAQ, and CME transitioned to electronic matching
- Today: market matching is automated, but fundamentally sequential (CLOB)

1992–1994: Borsa Italiana completed the transition from open outcry to electronic trading (`https://www.borsaitaliana.it/borsaitaliana/storia/storia/telematizzazione-scambi.en.htm`)

The big question

CME trading pits, Chicago

(2025)

Open outcry, Japan (circa 1960)

- *What is the real nature of the "computation" taking place on a trading venue?*

The big question

CME trading pits, Chicago

(2025)

Open outcry, Japan (circa 1960)

- *What is the real nature of the "computation" taking place on a trading venue?*
- Concurrency / Parallelism?
- What is the "right" model?

The big question

CME trading pits, Chicago

(2025)

Open outcry, Japan (circa 1960)

- *What is the real nature of the "computation" taking place on a trading venue?*
- Concurrency / Parallelism?
- What is the "right" model?
- We must start from a blank paper

Overview: where are we?

- **Fundamental** market design

- When humans were removed, markets converged to the CLOB model
- Is that solution optimal?
- *Is it optimally designed as a computational model at the fundamental level?*
- Not quite ...

Overview: where are we?

- **Fundamental** market design

- When humans were removed, markets converged to the CLOB model
- Is that solution optimal?
- *Is it optimally designed as a computational model at the fundamental level?*
- Not quite ...

- Motivation for Frequent Batch Auctions (Budish et al, 2015.)

Volume 130, Issue 4
November 2015

JOURNAL ARTICLE EDITOR'S CHOICE

The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response *

Eric Budish, Peter Cramton, John Shim

The Quarterly Journal of Economics, Volume 130, Issue 4, November 2015, Pages 1547–1621, https://doi.org/10.1093/qje/qjv027

Published: 23 July 2015

The issue with CLOB

The problem arises from the **interaction** of two **fundamental** design choices:

1. Treating time as continuous
2. Sequential processing of orders

The issue with CLOB

The problem arises from the **interaction** of two **fundamental** design choices:

1. Treating time as continuous
2. Sequential processing of orders

Both at the level of fundamental design

Manifest later at different levels

FBA's solution

Key idea: address both in the following way:

1. "Put time into discrete units"
   (discretize time into small uniform intervals)
2. "Process incoming orders in *batches* using auctions"
   (match orders simultaneously at the end of each interval; single market clearing price)

Frequent Batch Auctions

Budish, Cramton, Shim. *The high-frequency trading arms race: frequent batch auctions as a market design response.* QJE, 2015.

Limitation of FBA

Remark: FBAs work aimed to reveal the structural issues of markets, rather than a quest for definitive computational model(s).

FBA removes continuous-time priority ✓

However, regarding order matching: (?)

- Concurrency is not explicit in the model
- Parallelism is not native to the computation

Moreover, at the structural level: (?)

- Incoming orders are not mandated to always interact first with the resident market[1]

[1] In CLOB, incoming orders always interact with the resident market first. By necessity.

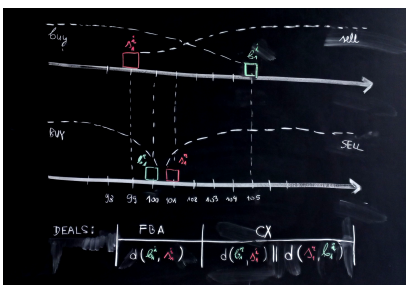
Our solution **CX**: beyond FBA

1. Put time into discrete units ✓ (as in FBA)
2. Enable natural concurrency and parallelism in order matching ✓ ✓
   (while preserving desirable computational and economic properties)

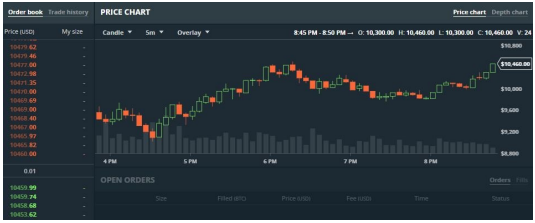
- Incoming order flow is mandated to always interact first with the resident market ✓

CX:

- enabling native concurrency and parallelism in electronic markets' computation
- retaining the structural consistency with CLOB, rathen that departing from it



Clarification: main views of the market



Price chart (more often used)

Depth chart (more holistic view)

From CLOB to FBA to **CX**

Key **starting insight** that **did not exist** and had to be recognized:

- incoming set of orders is a **market in its own right**[2], and therefore
- incoming and resident are two interacting markets

CLOB FBA **CX**

From **sequential**, to **batch**, to **parallel and concurrent processing**

[2] Pseudomarket, to be more precise.

From CLOB to FBA to **CX**

Key **starting insight** that **did not exist** and had to be recognized:

- incoming set of orders is a **market in its own right**[3], and therefore
- incoming and resident are two interacting markets

CLOB FBA **CX**

From **sequential**, to **batch**, to **parallel and concurrent processing**

[3] Pseudomarket, to be more precise.

From CLOB to FBA to **CX**

Key **starting insight** that **did not exist** and had to be recognized:

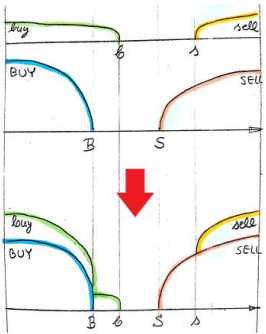
- incoming set of orders is a **market in its own right**[4], and therefore
- incoming and resident are two interacting markets

CLOB FBA **CX**

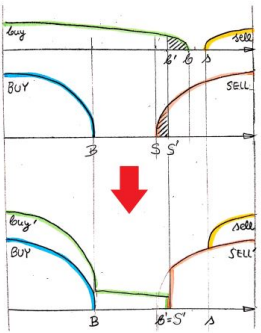
From **sequential**, to **batch**, to **parallel and concurrent processing**

[4] Pseudomarket, to be more precise.

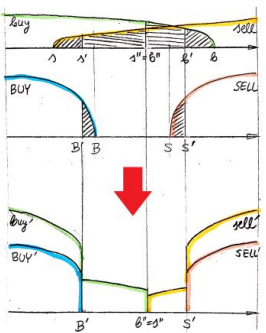
A closer look into **CX** computation



no successful order-matching

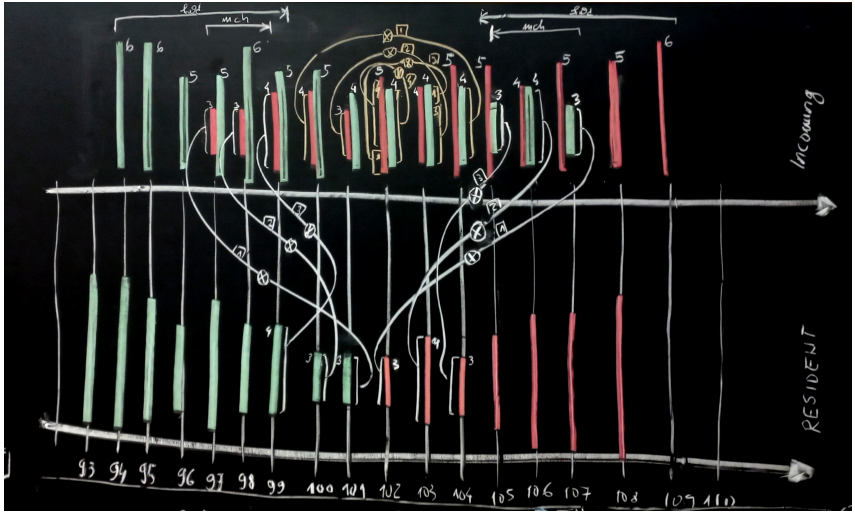


matching on one side



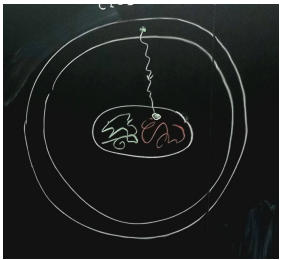
matching in all 3 segments

Illustrating an example computation

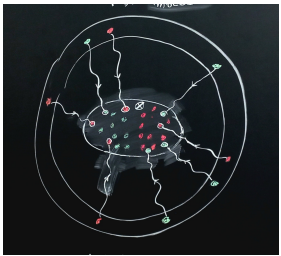


Another view on the 3 market models

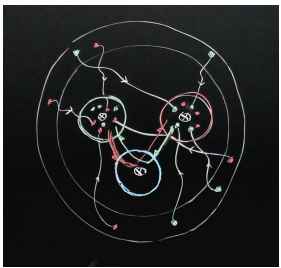
A higher level of abstraction: we emphasize the flow of orders, bringing the model closer to the **Reaction Systems** view of interaction.



CLOB

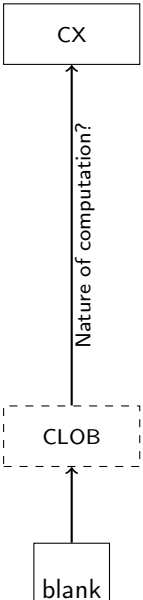


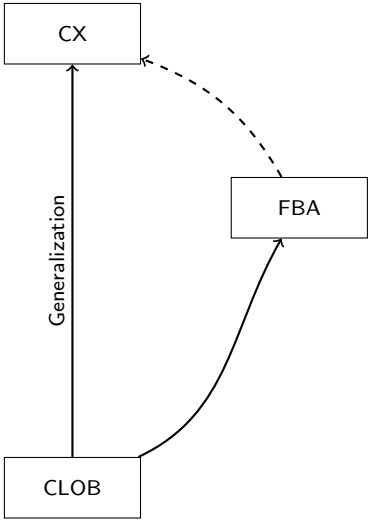
FBA



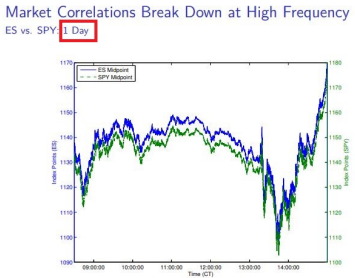
CX

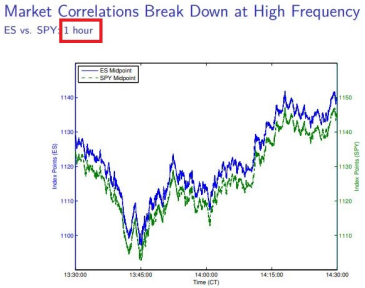
The big picture

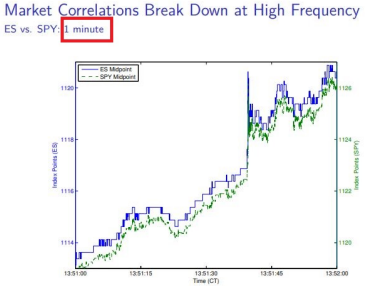


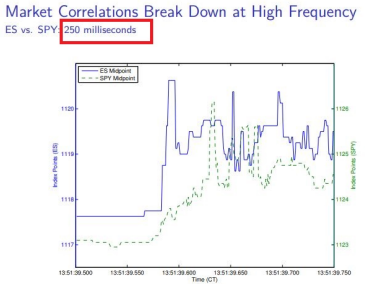


The HFT issue: latency arbitrage in correlated assets, an example









Primarily related to speed; rooted in continuous time

The sniping issue (related to front-running)

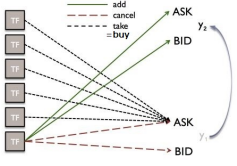
"Sniping"

Fundamental value and bid-ask spread

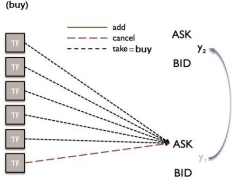
Fundamental value jumps

TFs providing liquidity send messages to cancel old quotes and add new quotes

TFs providing liquidity send messages to cancel old quotes and add new quotes



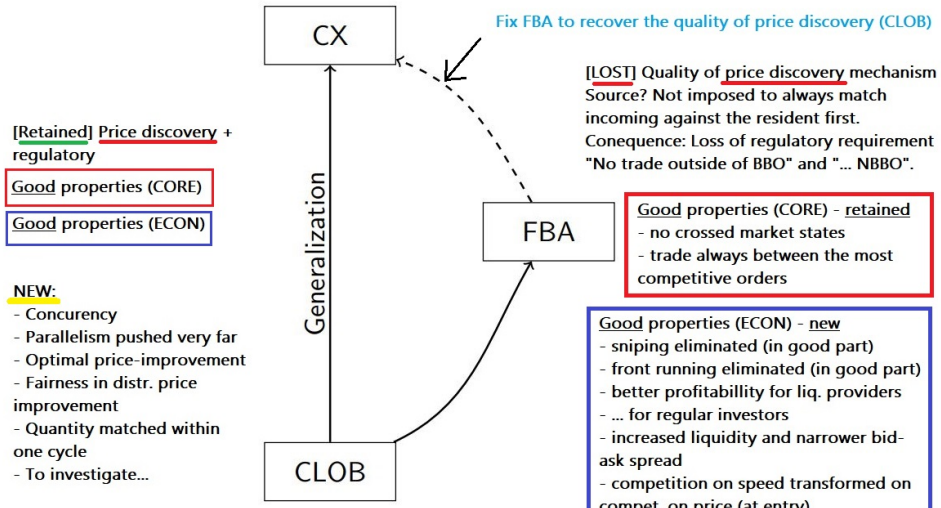
At same time, *other* TFs send messages to "snipe" the stale quotes



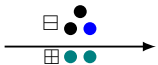
Because the market design processes messages in *serial*, liquidity providers get sniped with probability $\frac{N-1}{N}$ ... even though the information was public and all TFs have the exact same technology

- Even assuming the same speed; primarily rooted in sequential order processing.

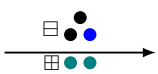
The big picture: properties

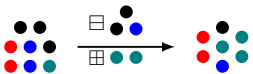


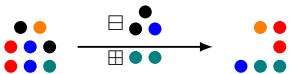
Reaction systems in a nutshell



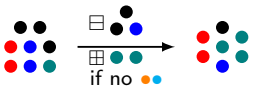
Reaction systems in a nutshell

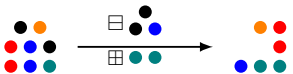




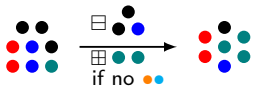


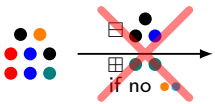
Reaction systems in a nutshell





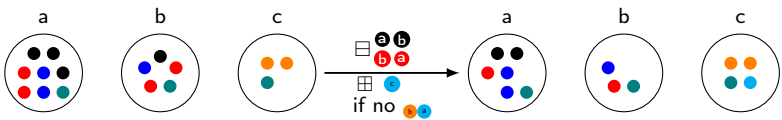
Reaction systems in a nutshell





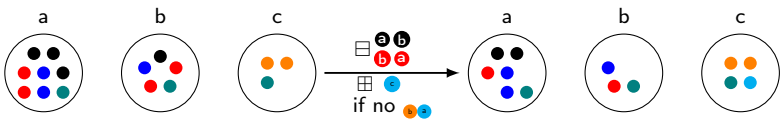
Reaction systems in a nutshell

Locations:

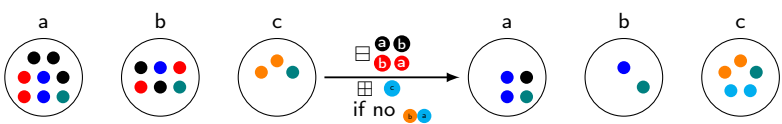


Reaction systems in a nutshell

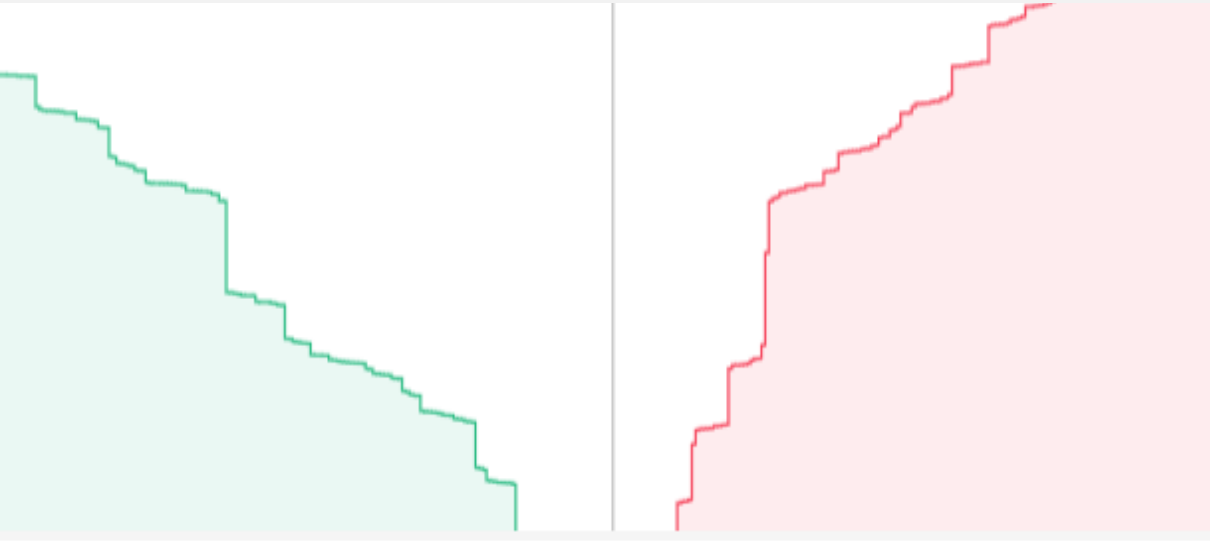
Locations:



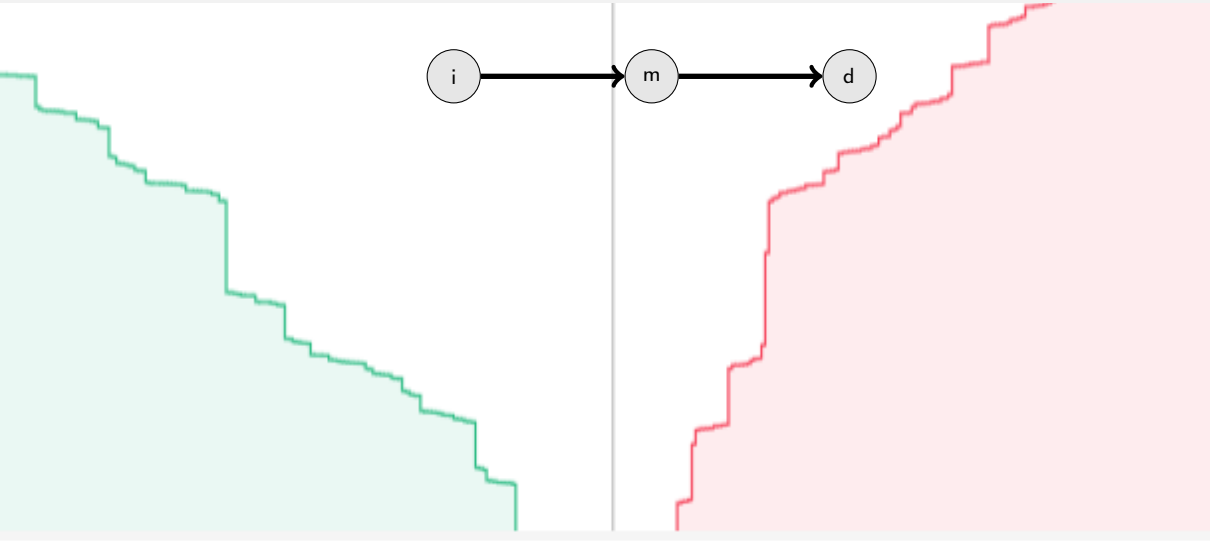
Concurrency:



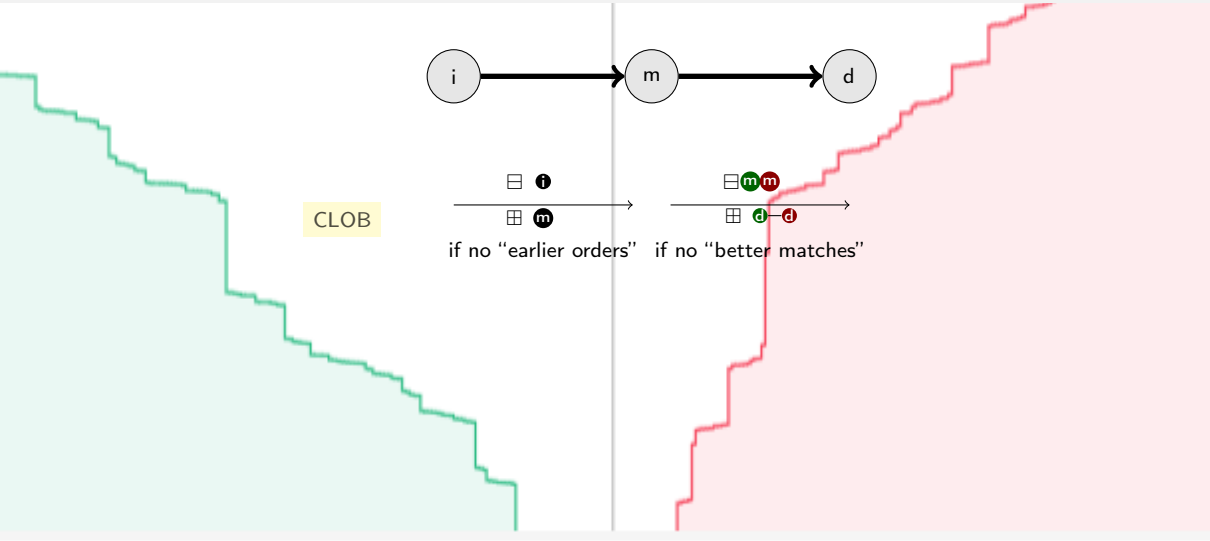
Formalising markets' with reaction systems



Formalising markets' with reaction systems

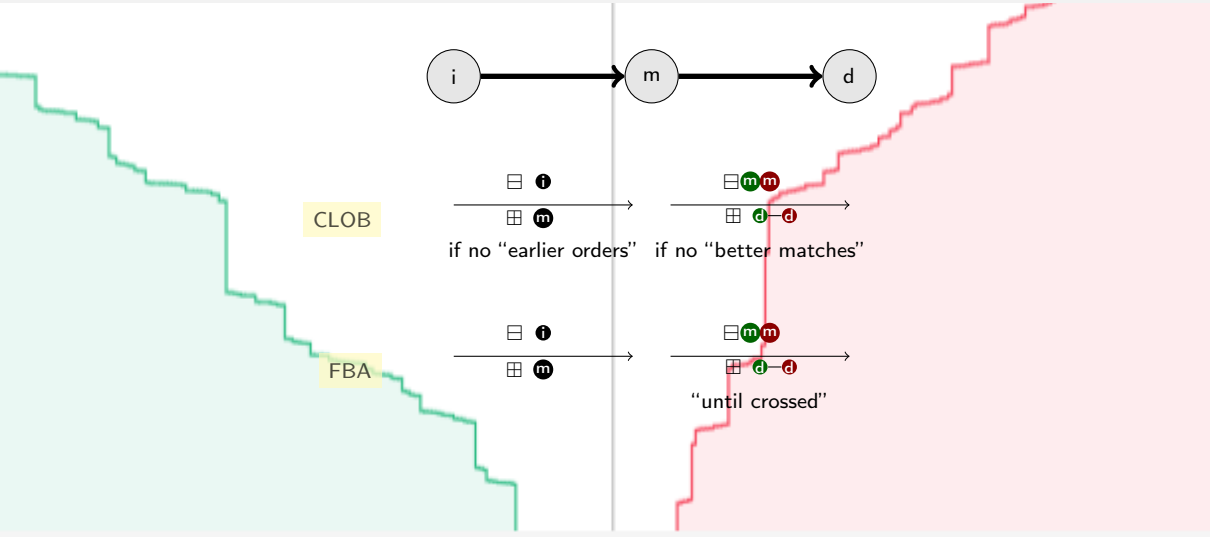


Formalising markets' with reaction systems



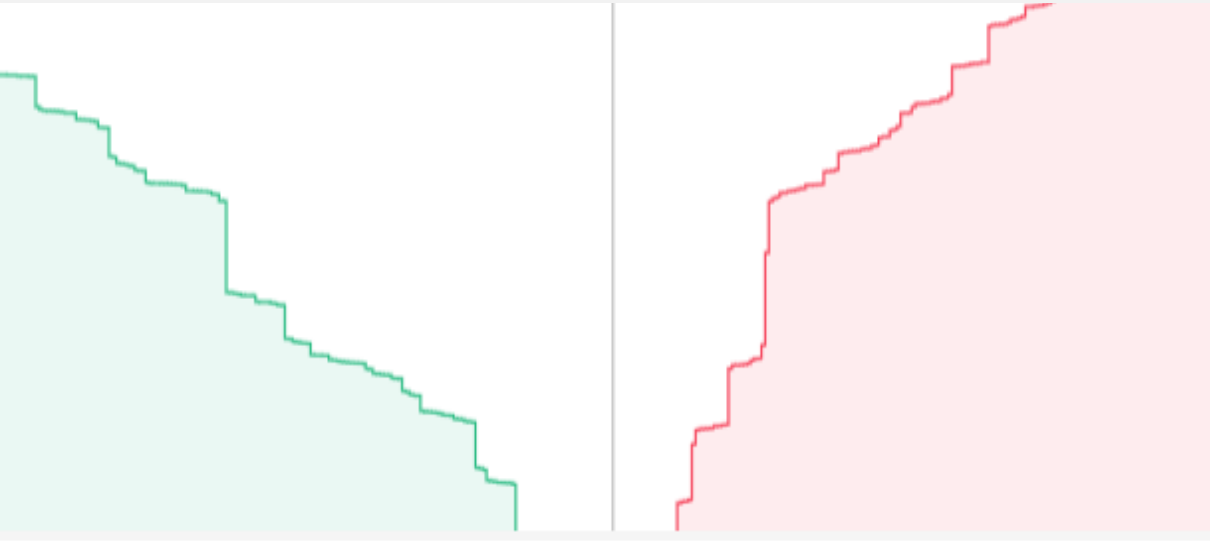
NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

Formalising markets' with reaction systems

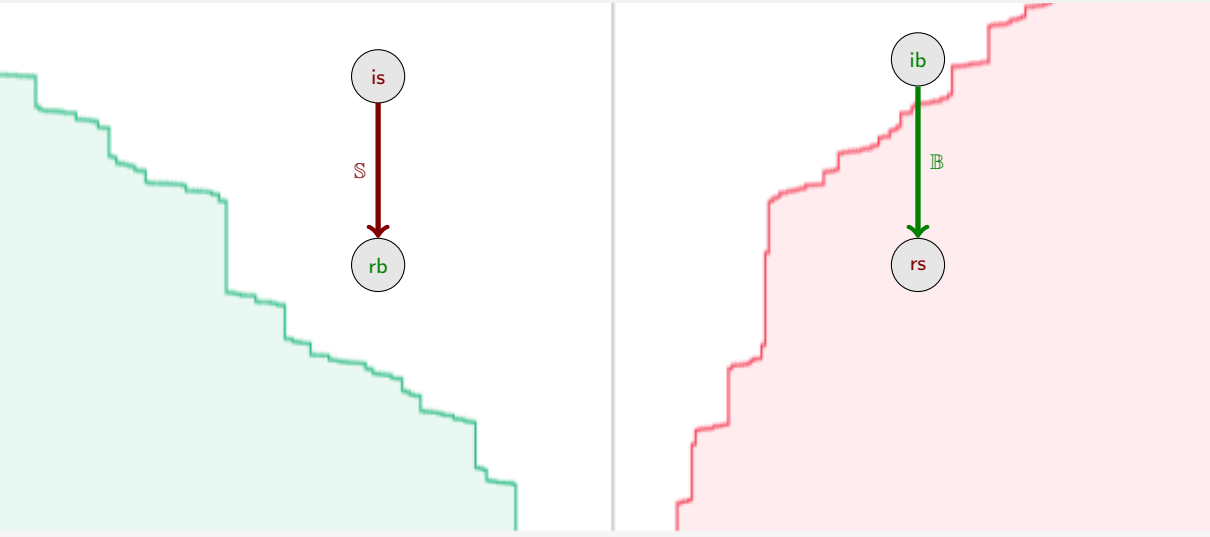


NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

CX: A new market model

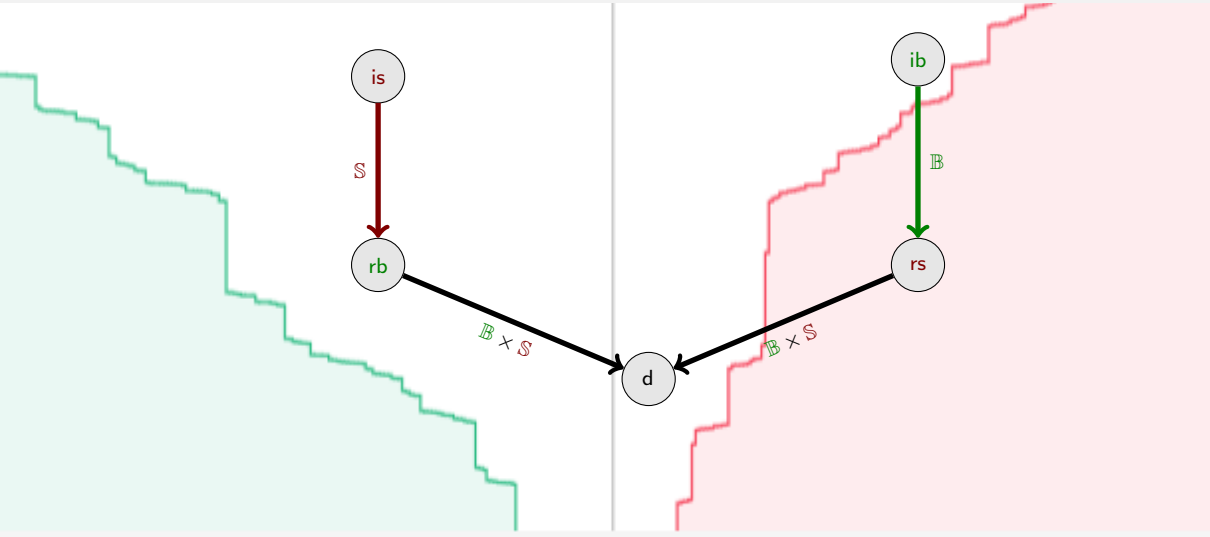


CX: A new market model



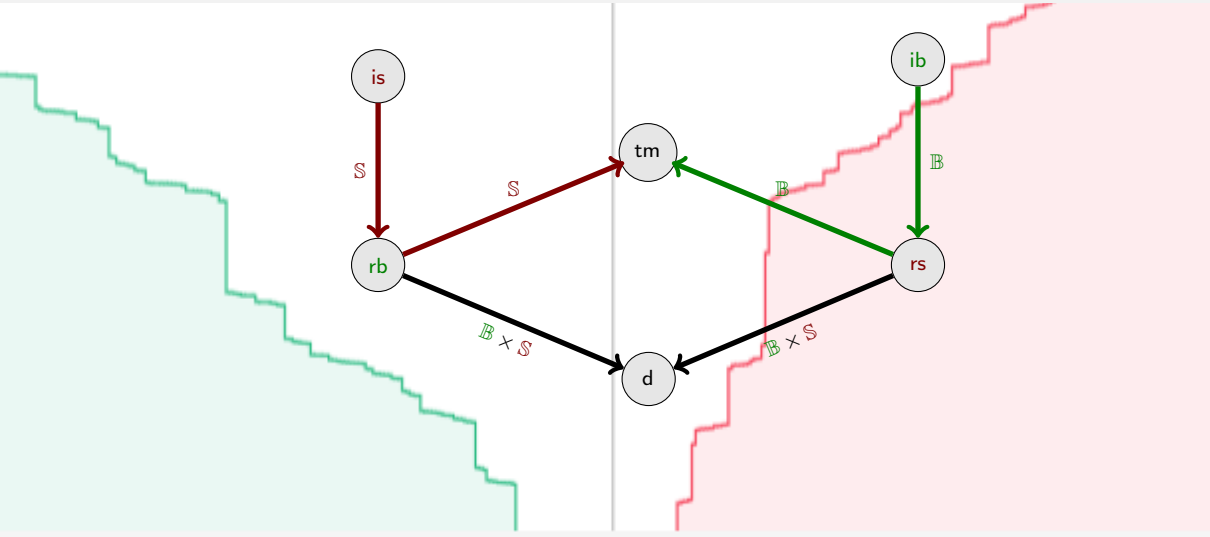
NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

CX: A new market model



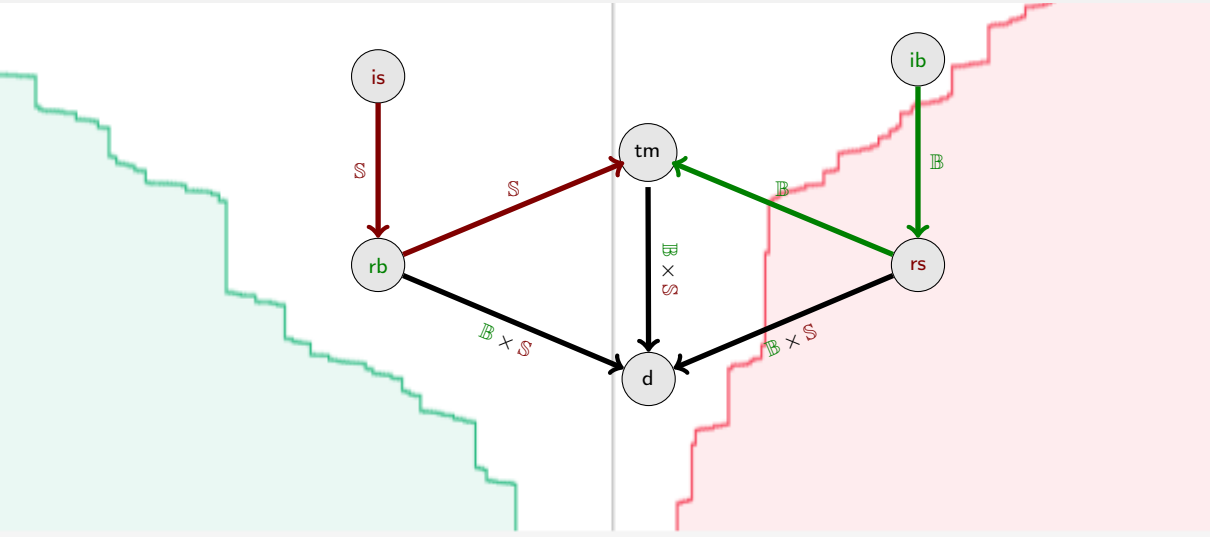
NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

CX: A new market model



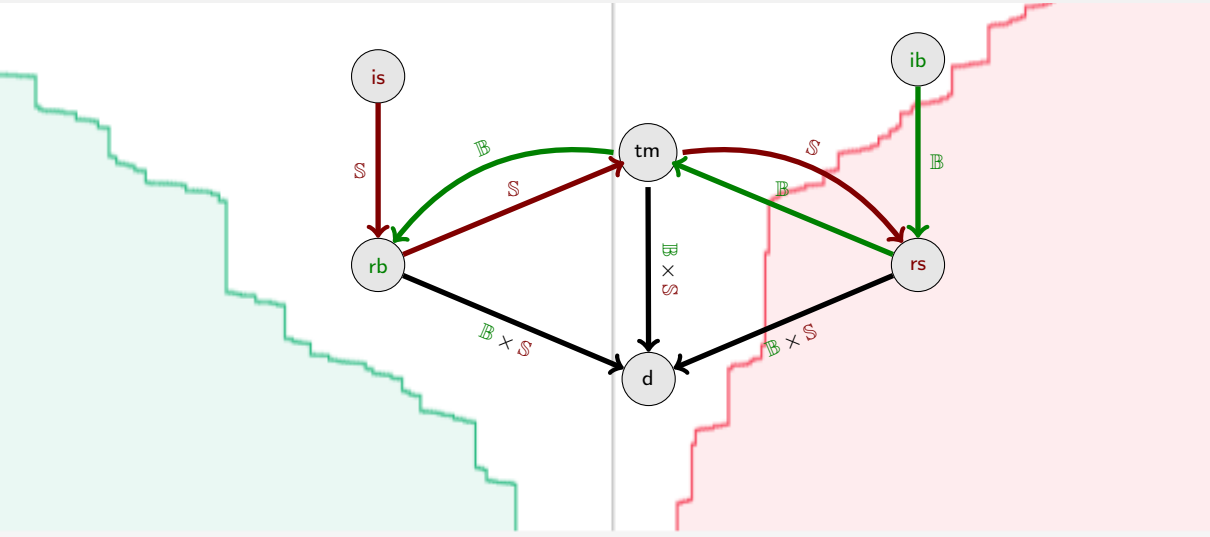
NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

CX: A new market model



NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

CX: A new market model



NIT/USDT +87.82% 0.1326 PAXG/USDT -0.46% 5,014.27 AGLD/USDT -0.86% 0.231 SCR

A glimpse of CX's formalisation

$$
\begin{aligned}
r_{\mathsf{deal}} &= \ell[(\mathsf{tok}(\ell), \mathsf{b}^{@i}, \mathsf{s}^{@j} \ ; \ \mathbb{B}^{>i} \cup \mathbb{S}^{<j} \ ; \ \mathsf{tok}(\ell), \mathsf{d}_{(\mathsf{b}^{@i}, \mathsf{s}^{@j})})] \\
r_{\mathsf{fwd}} &= \ell[(\mathsf{tok}(\ell), e \ ; \ \overline{e} \ ; \ \mathsf{tok}(\ell), \ell'[e])] \\
r_{\mathsf{ctl}} &= \ell[(\mathsf{tok}(\ell) \ ; \ I \ ; \ \ell'[\mathsf{tok}(\ell')])] \qquad \text{with} \qquad \ell \neq \mathsf{tm} \text{ and } \ell \xrightarrow{I} \ell' \text{ in } \mathcal{G} \\
r_{\mathsf{tms}} &= \mathsf{tm}_{(\beta,\sigma \ ; \ \emptyset \ ; \ \tau)} \\
r_{\mathsf{close}} &= \mathsf{tm}_{(\tau \ ; \ \mathbb{B} \cup \mathbb{S} \ ; \ \mathsf{ib}[\beta], \mathsf{is}[\sigma])}
\end{aligned}
$$

Added values

We can prove several properties:

- Market never crossed
- No trade outside current bid or ask
- Order priority respected

Added values

We can prove several properties:

- Market never crossed
- No trade outside current bid or ask
- Order priority respected

We can compare models and simulate them

Added values

We can prove several properties:

- Market never crossed
- No trade outside current bid or ask
- Order priority respected

We can compare models and simulate them

We can specify agents' strategies

- (not covered in this talk)

Added values

We can prove several properties:

- Market never crossed
- No trade outside current bid or ask
- Order priority respected

We can compare models and simulate them

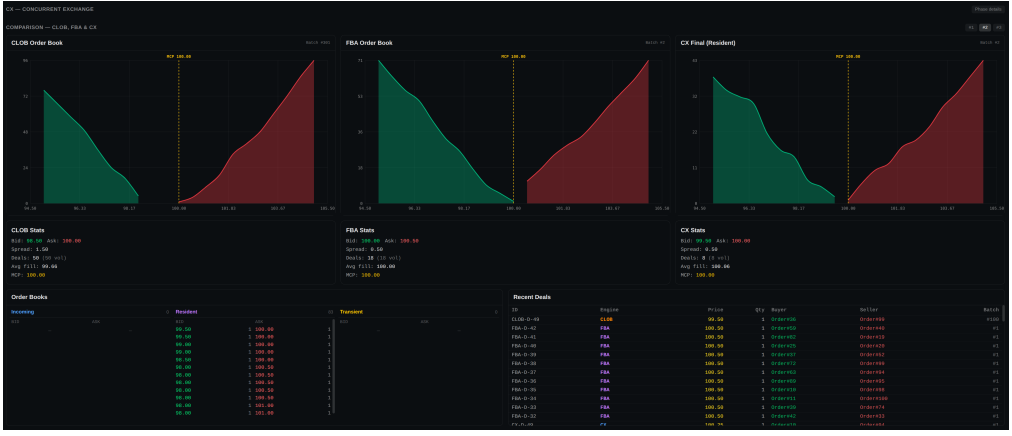
We can specify agents' strategies

- (not covered in this talk)

We conjecture several economic properties, e.g.,:

- Mitigation of front-running regular investors (non-HFT participants)
- Mitigation of sniping (i.e., the predatory removal of liquidity provider quotes)
- Better execution prices and increased profitability for regular investors
- Greater liquidity and narrower bid-ask spreads
- ...

Our prototype tool



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] Cedric Fournet and George Gonthier. The reflexive CHAM and the join-calculus.
In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996.

[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.

References II

[9] Roland Kuhn, Hernán C. Melgratti, and Emilio Tuosto. Behavioural types for local-first software. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[10] Roland Kuhn, Hernán C. Melgratti, and Emilio Tuosto. Behavioural types for local-first software (artifact). *Dagstuhl Artifacts Ser.*, 9(2):14:1–14:5, 2023.

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

[12] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, September 1994.

[13] 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`.