

# Behavioural Types for Local-First Software

Emilio Tuosto @ GSSI

joint work with

Roland Kuhn @ Actyx



and

Hernán Melgratti @ UBA



It-Matters  
Lucca 11-12 July, 2023

– Prelude –

## Take-away message

An approach to

trade consistency for availability in systems of **asymmetric replicated peers**

## Take-away message

An approach to  
trade consistency for availability in systems of **asymmetric replicated peers**  
using **local-first**'s principles to establish **eventual consensus**

## Take-away message

An approach to  
trade consistency for availability in systems of **asymmetric replicated peers**  
using **local-first**'s principles to establish **eventual consensus**  
formally supported by behavioural types

# Take-away message

An approach to

trade consistency for availability in systems of **asymmetric replicated peers**

using **local-first**'s principles to establish **eventual consensus**

formally supported by behavioural types



- **swarm** = (**machines** + **local logs**) \* **imaginary global log**
- **swarm protocols**: systems from an abstract **global** viewpoint
- enforce **good behaviour** via behavioural typing

# Take-away message

An approach to

trade consistency for availability in systems of **asymmetric replicated peers**

using **local-first**'s principles to establish **eventual consensus**

formally supported by behavioural types



- **swarm** = (**machines** + **local logs**) \* **imaginary global log**
- **swarm protocols**: systems from an abstract **global** viewpoint
- enforce **good behaviour** via behavioural typing



See our recent ECOOP 2023 paper

(<https://drops.dagstuhl.de/opus/volltexte/2023/18254/>; extended version available at <https://arxiv.org/abs/2305.04848>)

# Distributed coordination

## An “old” problem

Distributed agreement

Distributed sharing

Security

Computer-assisted collaborative work

...

## With some “solutions”

Centralisation points

Consensus protocols

Commutative replicated data types

...



# Distributed coordination

## An “old” problem

Distributed agreement

Distributed sharing

Security

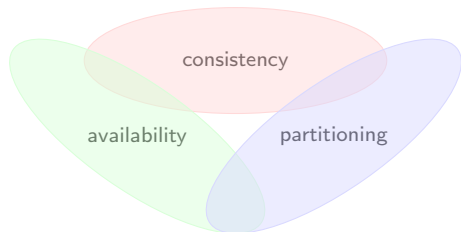
Computer-assisted collaborative work

...

## Availability = Money

Kohavi et al. KDD'14

- Amazon sales down 1% if 100ms delay
- Google searches down 0.2% - 0.6% if 100-400ms delay
- Bing's revenue down ~1.5% if 250ms delay



## With some “solutions”

Centralisation points

Consensus protocols

Commutative replicated data types

...

## A new (?) solution

### What about using local-first principles?

Thou shall be autonomous

Thou shall collaborate

Thou shall recognise conflicts

Thou shall resolve conflicts

Thou shall be consistent

# Plan of the talk

Some motivations

Our formalisation

Our typing discipline

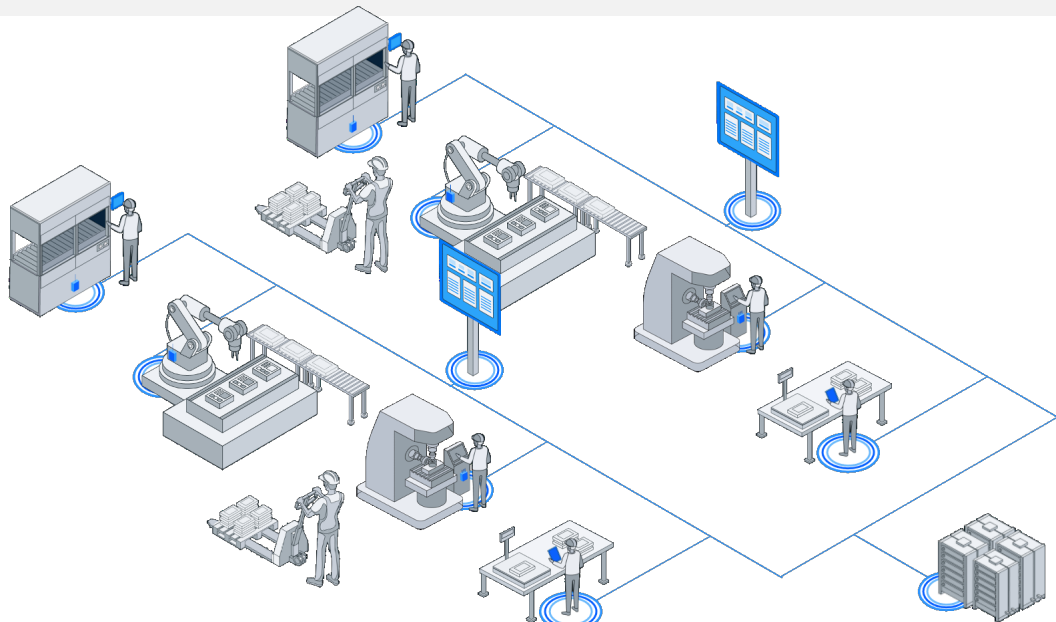
Tool support

Open issues

– Motivations –

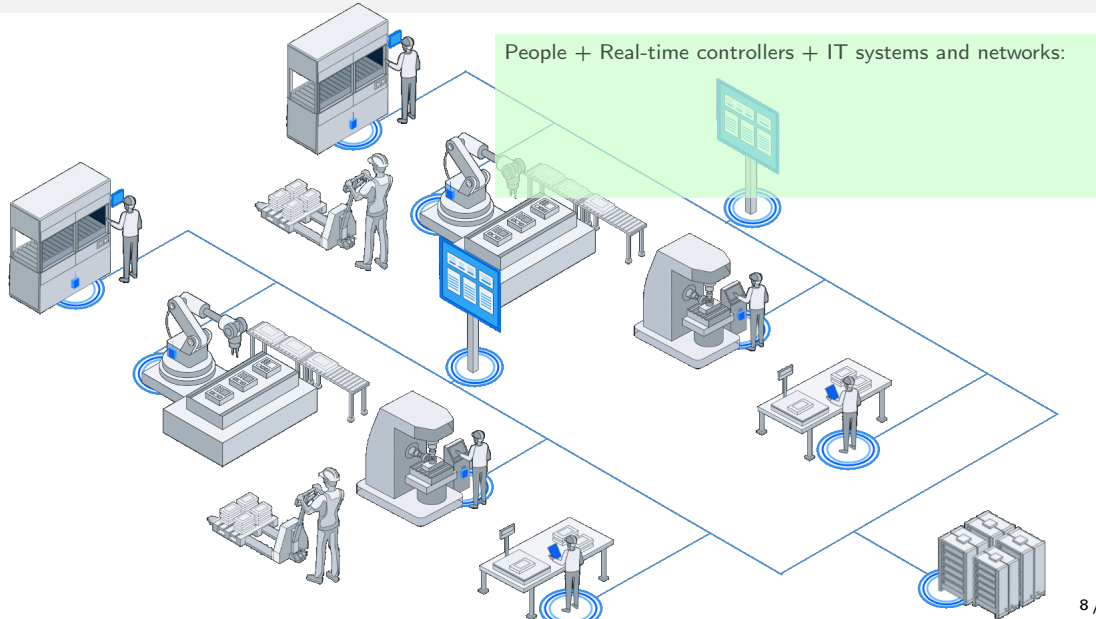
# A collaborative environment and its execution model

(the pictures are courtesy of Actyx AG)



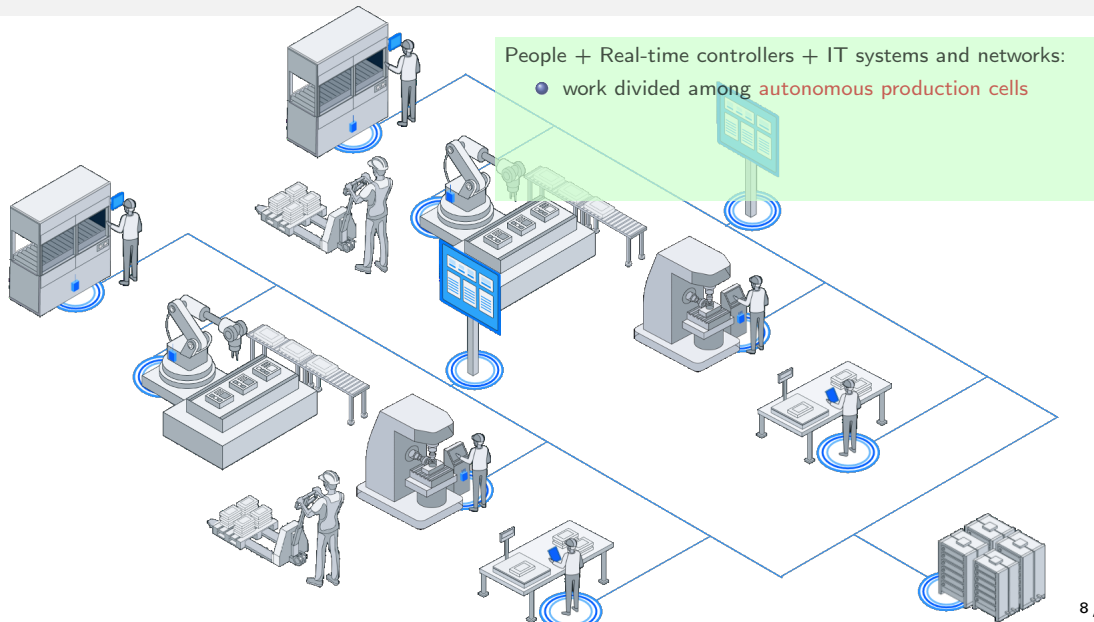
# A collaborative environment and its execution model

(the pictures are courtesy of Actyx AG)



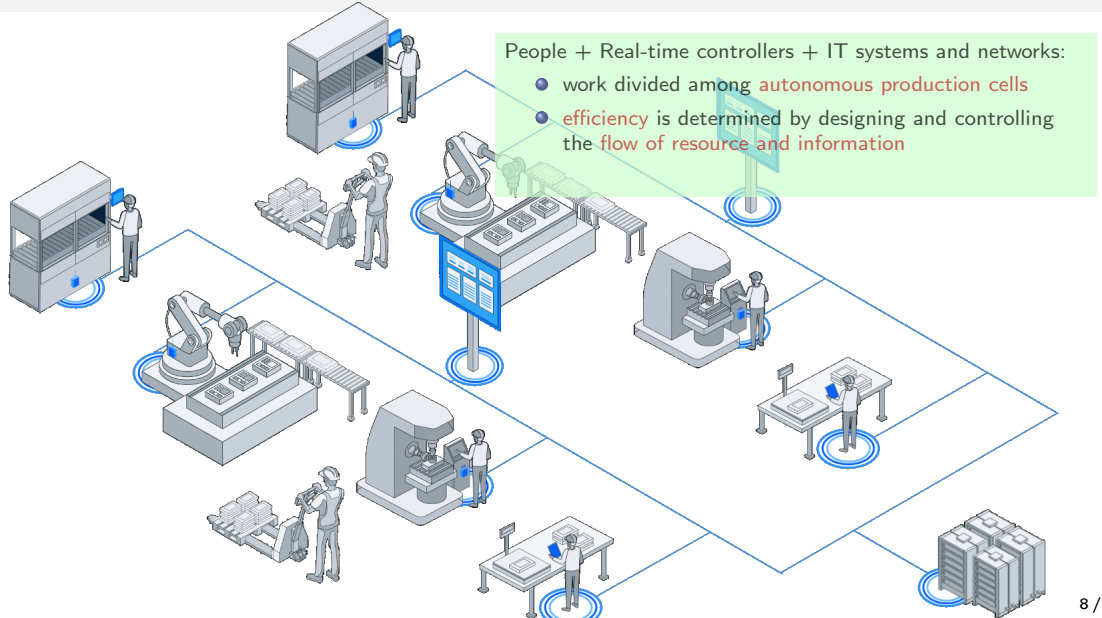
# A collaborative environment and its execution model

(the pictures are courtesy of Actyx AG)



# A collaborative environment and its execution model

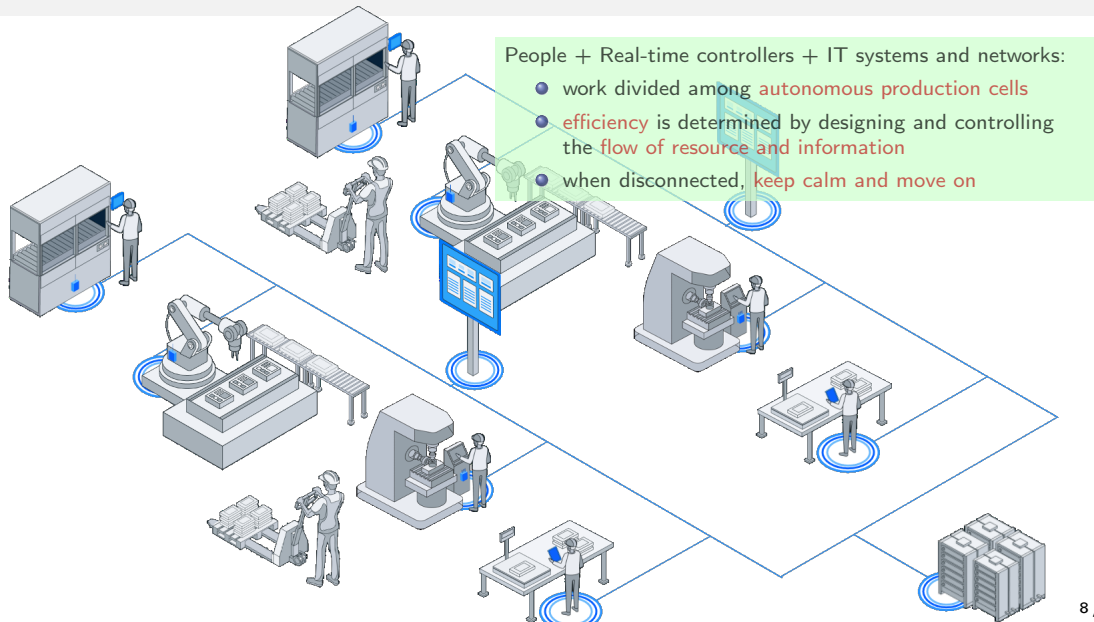
(the pictures are courtesy of Actyx AG)





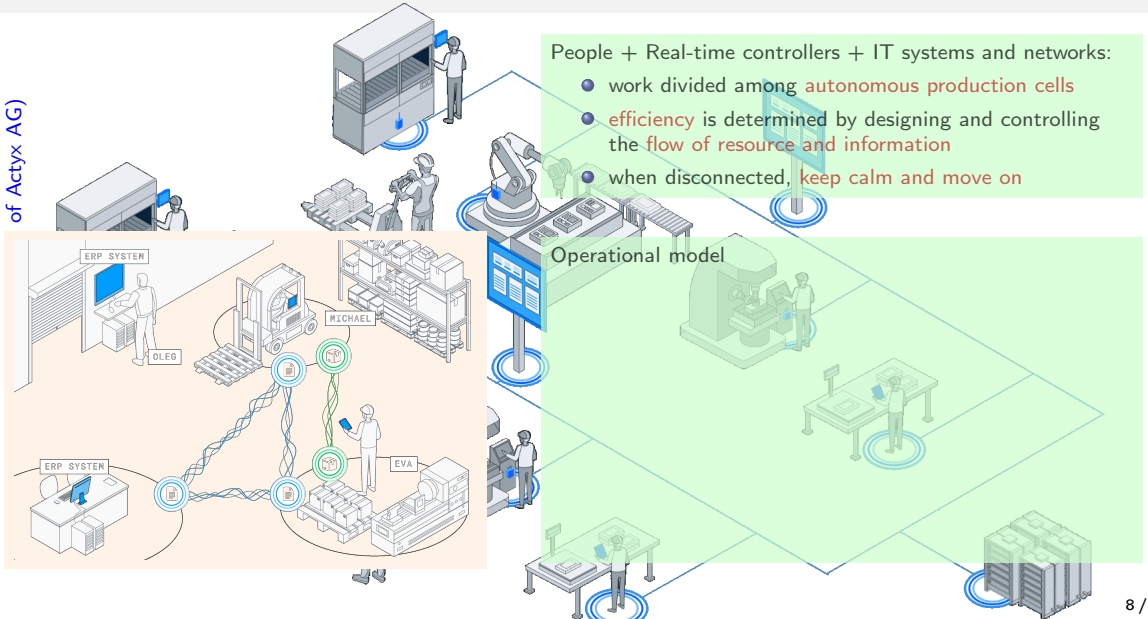
# A collaborative environment and its execution model

(the pictures are courtesy of Actyx AG)



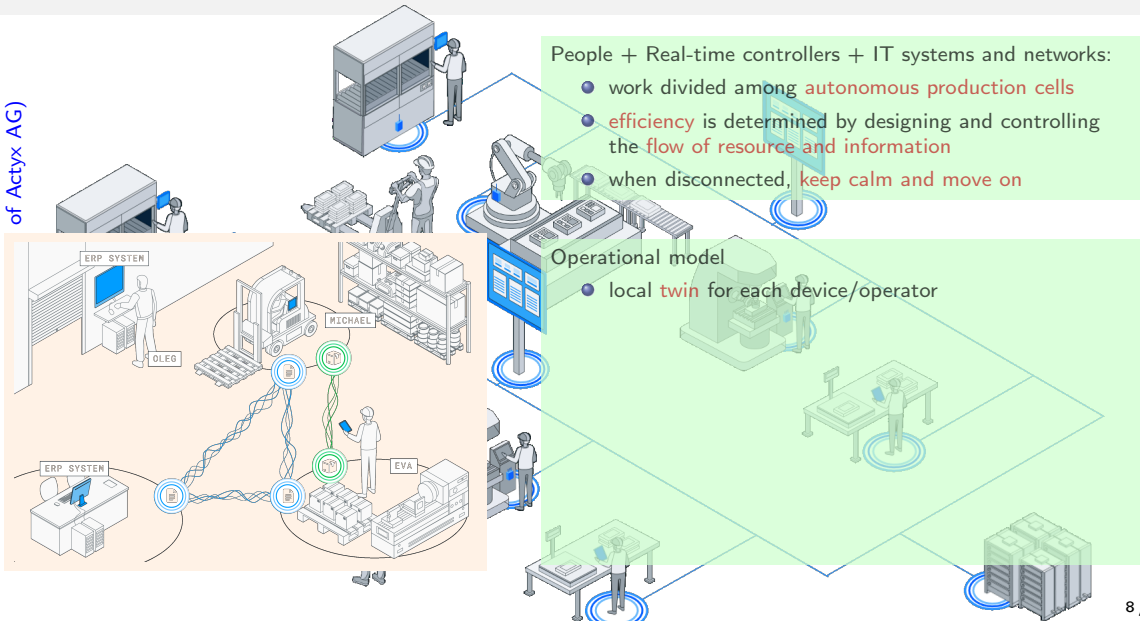
# A collaborative environment and its execution model

of Actyx AG)



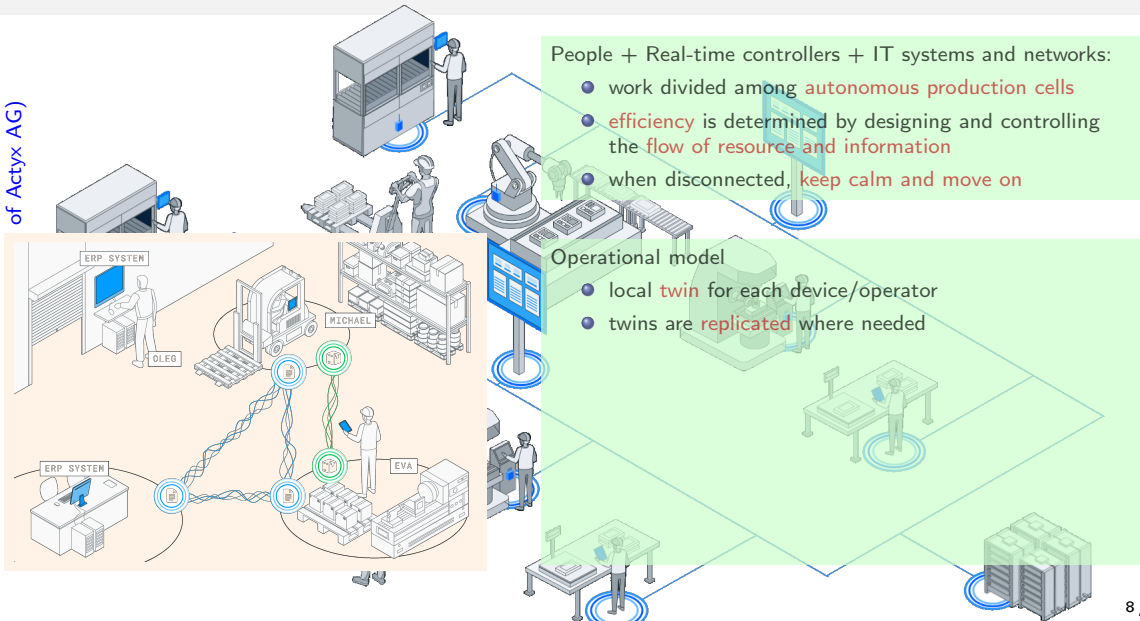
# A collaborative environment and its execution model

of Actyx AG)

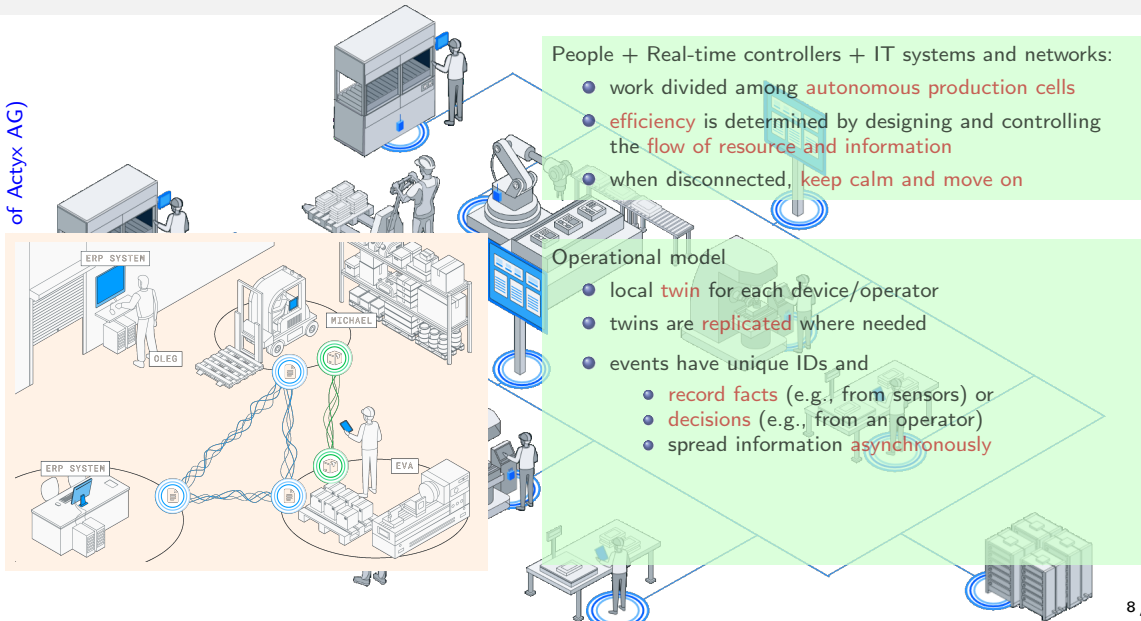


# A collaborative environment and its execution model

of Actyx AG)

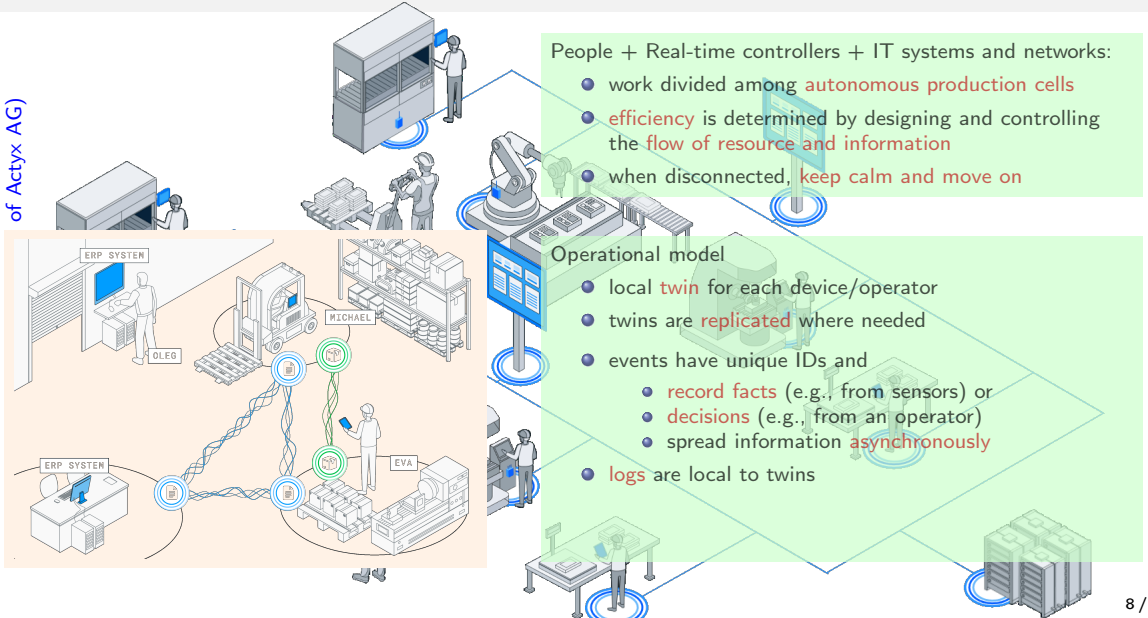


## of Actyx AG)



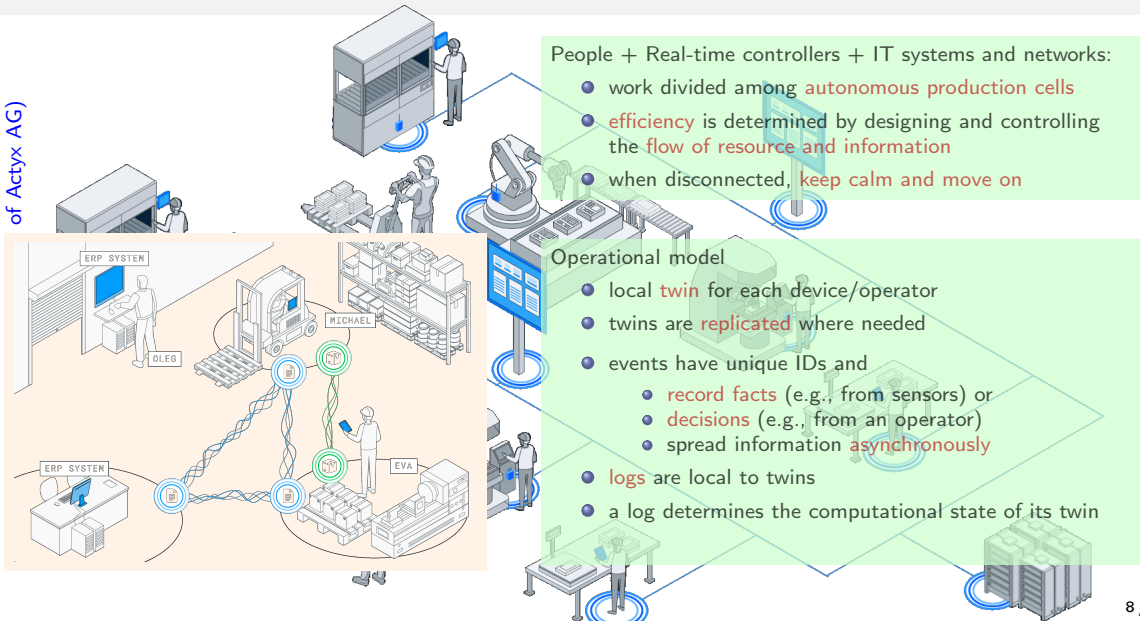
# A collaborative environment and its execution model

of Actyx AG)



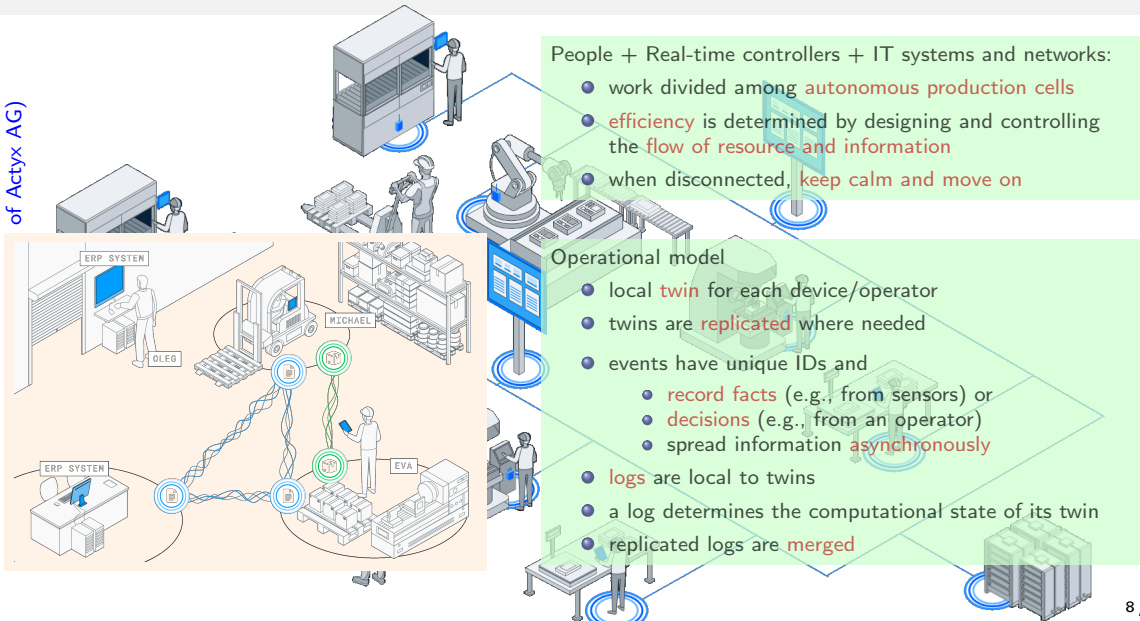
# A collaborative environment and its execution model

of Actyx AG)



# A collaborative environment and its execution model

of Actyx AG)





```
while true:
```

```
    execute ;
```

```
    propagate ;
```

```
    merge
```

## Other application domains / motivations

### More applications

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

Collaborative applications (<https://automerge.org/>)

Home automation

## Other application domains / motivations

### IoT...really?

Why your fridge and mobile **should go in the cloud** to talk to each other?

## Other application domains / motivations

“Anytime, anywhere...” really?

like 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>

## Other application domains / motivations

Also, taking decisions locally

can reduce downtime

shifts data ownership

gets rid of any centralization point...for real

# Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work

– A formal model –

Events

$e$

Logs

$e_1 \cdot e_2 \dots$



Events

$\vdash e : t$

$src(e)$

Logs

$\vdash e_1 \cdot e_2 \dots : t_1 \cdot t_2 \dots$

Events  $\vdash e : t$   
 $src(e)$

Logs  $\vdash e_1 \cdot e_2 \dots : t_1 \cdot t_2 \dots$

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

## Ingredients (II): log shipping

Machine **Alice** **emits** logs upon **execution** of commands (we'll see how in a moment)

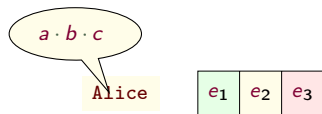
## Ingredients (II): log shipping

Machine **Alice** **emits** logs upon **execution** of commands (we'll see how in a moment)  
Such events are **appended** to the logs of machines in **two phases**:

## Ingredients (II): log shipping

Machine **Alice** **emits** logs upon **execution** of commands (we'll see how in a moment)  
Such events are **appended** to the logs of machines in **two phases**:

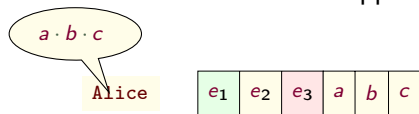
**Phase I**: emitted events are appended to the local log of the emitting machine



## Ingredients (II): log shipping

Machine **Alice** **emits** logs upon **execution** of commands (we'll see how in a moment)  
Such events are **appended** to the logs of machines in **two phases**:

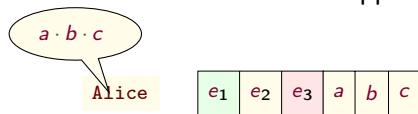
**Phase I**: emitted events are appended to the local log of the emitting machine



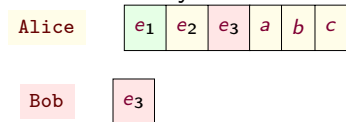
## Ingredients (II): log shipping

Machine **Alice** **emits** logs upon **execution** of commands (we'll see how in a moment)  
Such events are **appended** to the logs of machines in **two phases**:

**Phase I:** emitted events are appended to the local log of the emitting machine



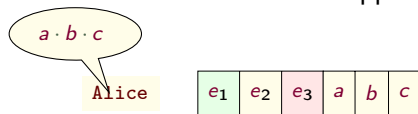
**Phase II:** newly emitted events are shipped to other machines



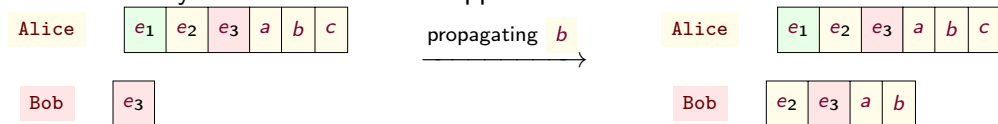
## Ingredients (II): log shipping

Machine **Alice** **emits** logs upon **execution** of commands (we'll see how in a moment)  
Such events are **appended** to the logs of machines in **two phases**:

**Phase I:** emitted events are appended to the local log of the emitting machine



**Phase II:** newly emitted events are shipped to other machines



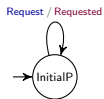


## Machines by example



`InitialP` =

# Machines by example



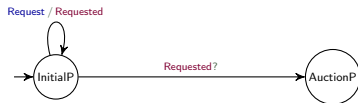
`InitialP` = `Request`  $\mapsto$  `Requested`.

# Machines by example



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

# Machines by example



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

$\text{AuctionP} =$

# Machines by example



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

$\text{AuctionP} = [ \text{Bid? BidderId? } \underline{\text{AuctionP}} ]$

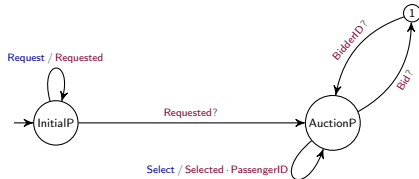
# Machines by example



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

$\text{AuctionP} = [ \text{Bid?} \underline{\text{BidderId? AuctionP}} ]$

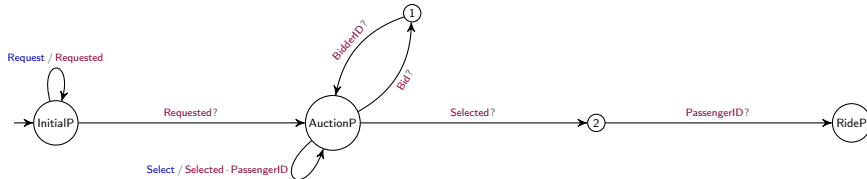
# Machines by example



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

$\text{AuctionP} = \text{Select} \mapsto \text{Selected} \cdot \text{PassengerID} \cdot [\text{Bid? BidderID?} \underline{\text{AuctionP}}]$

# Machines by example



InitialP = Request  $\mapsto$  Requested · [Requested? AuctionP]

AuctionP = Select  $\mapsto$  Selected · PassengerId · [  
 Bid? BidderId? AuctionP  
 &  
 Selected? PassengerId? RideP  
 ]

RideP = ...



## Machines, formally

Fix a set of commands ranged over by  $c$

Let  $\kappa$  range over finite maps from commands to non-empty log types

# Machines, formally

Fix a set of commands ranged over by  $c$

Let  $\kappa$  range over finite maps from commands to non-empty log types

Machine: **deterministic regular term** of  $M \stackrel{\text{co}}{::=} \kappa \cdot [t_1?M_1 \& \dots \& t_n?M_n]$

# Machines, formally

Fix a set of commands ranged over by  $c$

Let  $\kappa$  range over finite maps from commands to non-empty log types

Machine: **deterministic regular term** of  $M \stackrel{\text{co}}{::=} \kappa \cdot [t_1?M_1 \& \dots \& t_n?M_n]$

Think of machines as emitters/consumers of events with a semantics given in terms of state transition function :

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

That is

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

# Machines, formally

Fix a set of commands ranged over by  $c$

Let  $\kappa$  range over finite maps from commands to non-empty log types

**Machine:** **deterministic regular term** of  $M ::=^{\text{co}} \kappa \cdot [t_1?M_1 \& \dots \& t_n?M_n]$

Think of machines as emitters/consumers of events with a semantics given in terms of state transition function :

$$\delta(M, \epsilon) = M$$

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

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

That is

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

That is

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

# Swarms

Swarms:  $M_1[\ell_1] \mid \dots \mid M_n[\ell_n] \mid \ell$  s.t.  $\ell = \bigcup_{1 \leq i \leq n} \ell_i$  and  $\ell_i \sqsubseteq \ell$  for  $1 \leq i \leq n$

# Swarms

Swarms:  $M_1[\ell_1] \mid \dots \mid M_n[\ell_n] \mid \ell$  s.t.  $\ell = \bigcup_{1 \leq i \leq n} \ell_i$  and  $\ell_i \sqsubseteq \ell$  for  $1 \leq i \leq n$

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

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

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

That is

all events of  $\ell_1$  appear in the same order in  $\ell_2$

That is

the per-source partitions of  $\ell_1$  are prefixes of the corresponding partitions of  $\ell_2$

# Swarms

Swarms:  $M_1[l_1] \mid \dots \mid M_n[l_n] \mid \ell$  s.t.  $\ell = \bigcup_{1 \leq i \leq n} \ell_i$  and  $\ell_i \sqsubseteq \ell$  for  $1 \leq i \leq n$

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  $\ell_1$  appear in the same order in  $\ell_2$

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

That is

the per-source partitions of  $\ell_1$  are prefixes of the corresponding partitions of  $\ell_2$

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

$$\frac{\begin{array}{l} S(i) = M[\ell_i] \quad M[\ell_i] \xrightarrow{c/1} M[\ell'_i] \quad \text{src}(\ell'_i \setminus \ell_i) = \{i\} \quad \ell' \in \ell \bowtie \ell'_i \end{array}}{(S, \ell) \xrightarrow{c/1} (S[i \mapsto M[\ell'_i]], \ell')} \text{[Local]}$$



## Semantics of swarms

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

$$\frac{S(i) = M[\ell_i] \quad M[\ell_i] \xrightarrow{c/1} M[\ell'_i] \quad \text{src}(\ell'_i \setminus \ell_i) = \{i\} \quad \ell' \in \ell \bowtie \ell'_i}{(S, \ell) \xrightarrow{c/1} (S[i \mapsto M[\ell'_i]], \ell')} \text{[Local]}$$

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

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

# Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work

– Behavioural types for swarms –

# Inspired by choreographies

## Quoting W3C:

*"[...] a **contract** [...] of the common **ordering conditions and constraints** under which **messages** are exchanged [...] from a **global viewpoint** [...]  
Each **party** can then use the global definition to **build and test solutions** [...]  
global specification is in turn **realised by combination of** the resulting **local systems**"*

# Inspired by choreographies

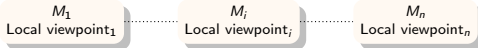
## Quoting W3C:

*"[...] a **contract** [...] of the common **ordering conditions and constraints** under which **messages** are exchanged [...] from a **global viewpoint** [...]  
Each **party** can then use the global definition to **build and test solutions** [...]  
global specification is in turn **realised by combination of** the resulting **local systems**"*

### Synchrony

Choreography G  
global viewpoint

### Asynchrony



# Inspired by choreographies

## Quoting W3C:

“[...] a *contract* [...] of the common *ordering conditions and constraints* under which *messages* are exchanged [...] from a *global viewpoint* [...]  
Each *party* can then use the global definition to *build and test solutions* [...]  
global specification is in turn *realised by combination of* the resulting *local systems*”

### Synchrony

Choreography G  
global viewpoint

### Asynchrony

$M_1$   
Local viewpoint<sub>1</sub>

$M_i$   
Local viewpoint<sub>i</sub>

$M_n$   
Local viewpoint<sub>n</sub>

spec, no code

# Inspired by choreographies

## Quoting W3C:

"[...] a *contract* [...] of the common *ordering conditions and constraints* under which *messages* are exchanged [...] from a *global viewpoint* [...]  
*Each party* can then use the global definition to *build and test solutions* [...]  
global specification is in turn *realised by combination of* the resulting *local systems*"

### Synchrony

Choreography G  
global viewpoint

Well-formedness

### Asynchrony

$M_1$   
Local viewpoint<sub>1</sub>

$M_i$   
Local viewpoint<sub>i</sub>

$M_n$   
Local viewpoint<sub>n</sub>

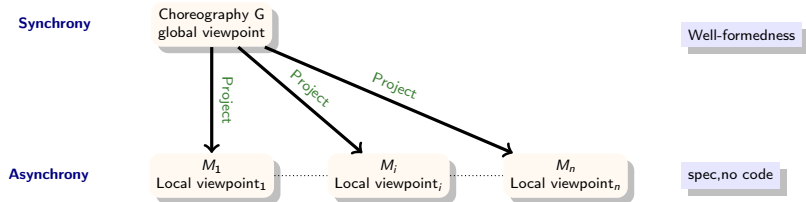
spec, no code

# Inspired by choreographies

## Quoting W3C:

"[...] a *contract* [...] of the common *ordering conditions and constraints* under which *messages* are exchanged [...] from a *global viewpoint* [...]"

*Each party* can then use the global definition to *build and test solutions* [...] global specification is in turn *realised by combination of* the resulting *local systems*"



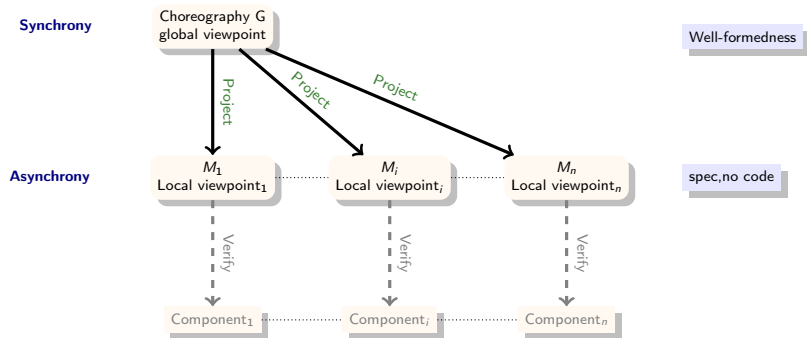


# Inspired by choreographies

## Quoting W3C:

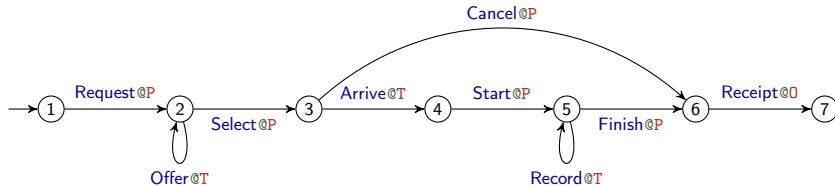
"[...] a **contract** [...] of the common **ordering conditions and constraints** under which **messages** are exchanged [...] from a **global viewpoint** [...]"

**Each party** can then use the global definition to **build and test solutions** [...] global specification is in turn **realised by combination of** the resulting **local systems**"



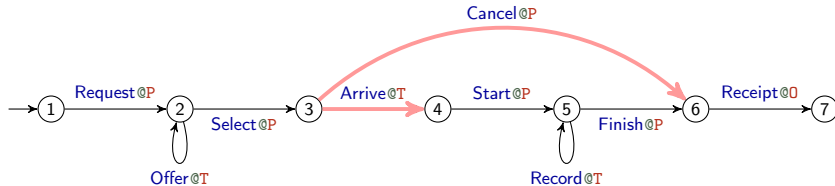
## Swarm protocols by example

An intuitive auction protocol for a passenger **P** to get a taxi **T**:



## Swarm protocols by example

An intuitive auction protocol for a passenger  $P$  to get a taxi  $T$ :



# Swarm protocols: global type for local-first applications

An **idealised** specification relying on **synchronous communication**

Fix a set of roles ranged over by  $\mathbf{R}$  (e.g.,  $\mathbf{P}$ ,  $\mathbf{T}$ , and  $\mathbf{O}$  on slide 31)

The syntax of swarm protocols is again given co-inductively:

$$\mathbf{G} ::=^{\text{co}} \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i \quad | \quad 0 \quad \text{where } I \text{ is a finite set (of indexes)}$$

## An example

A swarm protocol for the taxi scenario on slide 31:

$$G = \text{Request@P}\langle \text{Requested} \rangle . G_{\text{auction}}$$

$$\begin{aligned} G_{\text{auction}} &= \text{Offer@T}\langle \text{Bid} \cdot \text{BidderID} \rangle . G_{\text{auction}} \\ &+ \text{Select@P}\langle \text{Selected} \cdot \text{PassengerID} \rangle . G_{\text{choose}} \end{aligned}$$

$$\begin{aligned} G_{\text{choose}} &= \text{Arrive@T}\langle \text{Arrived} \rangle . \text{Start@P}\langle \text{Started} \rangle . G_{\text{ride}} \\ &+ \text{Cancel@P}\langle \text{Cancelled} \rangle . \text{Receipt@O}\langle \text{Receipt} \rangle . 0 \end{aligned}$$

$$\begin{aligned} G_{\text{ride}} &= \text{Record@T}\langle \text{Path} \rangle . G_{\text{ride}} \\ &+ \text{Finish@P}\langle \text{Finished} \cdot \text{Rating} \rangle . \text{Receipt@O}\langle \text{Receipt} \rangle . 0 \end{aligned}$$

## An example

A swarm protocol for the taxi scenario on slide 31:

$$G = \text{Request@P}\langle \text{Requested} \rangle . G_{\text{auction}}$$

$$\begin{aligned} G_{\text{auction}} &= \text{Offer@T}\langle \text{Bid} \cdot \text{BidderID} \rangle . G_{\text{auction}} \\ &+ \text{Select@P}\langle \text{Selected} \cdot \text{PassengerID} \rangle . G_{\text{choose}} \end{aligned}$$

*Note the log types  
in each prefixes*

$$\begin{aligned} G_{\text{choose}} &= \text{Arrive@T}\langle \text{Arrived} \rangle . \text{Start@P}\langle \text{Started} \rangle . G_{\text{ride}} \\ &+ \text{Cancel@P}\langle \text{Cancelled} \rangle . \text{Receipt@O}\langle \text{Receipt} \rangle . 0 \end{aligned}$$

$$\begin{aligned} G_{\text{ride}} &= \text{Record@T}\langle \text{Path} \rangle . G_{\text{ride}} \\ &+ \text{Finish@P}\langle \text{Finished} \cdot \text{Rating} \rangle . \text{Receipt@O}\langle \text{Receipt} \rangle . 0 \end{aligned}$$

## Swarm protocols as FSA

Like for machines, a swarm protocols  $G = \sum_{i \in I} c_i @ R_i \langle 1_i \rangle$ .  $G_i$  has an associated FSA:

- the set of states consists of  $G$  plus the states in  $G_i$  for each  $i \in \{1 \dots, n\}$
- $G$  is the initial state
- for each  $i \in I$ ,  $G$  has a transition to state  $G_i$  labelled with  $c_i @ R_i \langle 1_i \rangle$ , written
$$G \xrightarrow{c_i / 1_i} G_i$$

# Semantics of swarm protocols

One rule only!

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



# Semantics of swarm protocols

One rule only!

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

where

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

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

# Semantics of swarm protocols

One rule only!

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

where

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

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

# Semantics of swarm protocols

One rule only!

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

where

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

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

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

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

log determinism

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

$$\left( \sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i \right) \downarrow_R = \kappa \cdot [\&_{i \in I} l_i ? G_i \downarrow_R]$$

where  $\kappa = \{ (c_i / l_i) \mid R_i = R \text{ and } i \in I \}$

# 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

$$\left( \sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i \right) \downarrow_R = \kappa \cdot [\&_{i \in I} l_i? G_i \downarrow_R]$$

where  $\kappa = \{(c_i / l_i) \mid R_i = R \text{ and } i \in I\}$

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  $G = \sum_{i \in I} c_i @ R_i \langle 1_i \rangle . G_i$ , the  
projection of  $G$  on a role  $R$  with respect to subscription  $\sigma$  is

$$G \downarrow_R^\sigma = \kappa \cdot [\&_{j \in J} \text{filter}(1_j, \sigma(R)) ? G_j \downarrow_R^\sigma]$$

where

## Another attempt



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

*In pub-sub,  
processes subscribe  
to "topics"*

Given  $G = \sum_{i \in I} c_i @ R_i \langle 1_i \rangle . G_i$ , the  
projection of  $G$  on a role  $R$  with respect to subscription  $\sigma$  is

$$G \downarrow_R^\sigma = \kappa \cdot [\&_{j \in J} \text{filter}(1_j, \sigma(R)) ? G_j \downarrow_R^\sigma] \quad \text{where}$$

$$\begin{aligned} \kappa &= \{c_i / 1_i \mid R_i = R \text{ and } i \in I\} \\ J &= \{i \in I \mid \text{filter}(1_i, \sigma(R)) \neq \epsilon\} \\ \text{filter}(1, E) &= \begin{cases} \epsilon, & \text{if } t = \epsilon \\ t \cdot \text{filter}(1', E) & \text{if } t \in E \text{ and } 1 = t \cdot 1' \\ \text{filter}(1, E) & \text{otherwise} \end{cases} \end{aligned}$$

## 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])

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

## Causality

Fix a subscription  $\sigma$ . For each branch  $i \in I$  of  $G = \sum_{i \in I} c_i @ R_i \langle 1_i \rangle . G_i$

Explicit re-enabling  $\sigma(R_i) \cap 1_i \neq \emptyset$

*If  $R$  should have  $c$  enabled after  $c'$  then  $\sigma(R)$  contains some event type emitted by  $c'$*

Command causality if  $R$  executes a command in  $G_i$   
then  $\sigma(R) \cap 1_i \neq \emptyset$  and  $\sigma(R) \cap 1_i \supseteq \bigcup_{R' \in \sigma G_i} \sigma(R') \cap 1_i$

## Well-formedness = Causality + Determinacy

Trading consistency for availability has implications:

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

$\implies$  different roles may take inconsistent decisions

### Causality & Determinacy

Fix a subscription  $\sigma$ . For each branch  $i \in I$  of  $G = \sum_{i \in I} c_i @ R_i \langle 1_i \rangle . G_i$

Explicit re-enabling  $\sigma(R_i) \cap 1_i \neq \emptyset$

Command causality if  $R$  executes a command in  $G_i$   
then  $\sigma(R) \cap 1_i \neq \emptyset$  and  $\sigma(R) \cap 1_i \supseteq \bigcup_{R' \in_\sigma G_i} \sigma(R') \cap 1_i$

Determinacy  $R \in_\sigma G_i \implies 1_i[0] \in \sigma(R)$

# Well-formedness = Causality + Determinacy - Confusion

Trading consistency for availability has implications:

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

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

## Causality & Determinacy & Confusion freeness

Fix a subscription  $\sigma$ . For each branch  $i \in I$  of  $G = \sum_{i \in I} c_i @ R_i \langle 1_i \rangle . G_i$

Explicit re-enabling  $\sigma(R_i) \cap 1_i \neq \emptyset$

Command causality if  $R$  executes a command in  $G_i$   
then  $\sigma(R) \cap 1_i \neq \emptyset$  and  $\sigma(R) \cap 1_i \supseteq \bigcup_{R' \in_\sigma G_i} \sigma(R') \cap 1_i$

Determinacy  $R \in_\sigma G_i \Rightarrow 1_i[0] \in \sigma(R)$

Confusion freeness for each  $t$  starting a log emitted by a command in  $G$   
there is a unique state  $G'$  reachable from  $G$  which emits  $t$

# Implementations

Write  $\ell \equiv_{G,\sigma} \ell'$  when  $\ell$  and  $\ell'$  have the same effective type wrt  $G$  and  $\sigma$

A swarm  $(S, \epsilon)$  is eventually faithful to  $G$  and  $\sigma$  if  $(S, \epsilon) \Longrightarrow (S, \ell)$  then there is  $(G, \epsilon) \Longrightarrow (G, \ell')$  with  $\ell \equiv_{G,\sigma} \ell'$



# Implementations

Write  $\ell \equiv_{G,\sigma} \ell'$  when  $\ell$  and  $\ell'$  have the same effective type wrt  $G$  and  $\sigma$

A swarm  $(S, \epsilon)$  is eventually faithful to  $G$  and  $\sigma$  if  $(S, \epsilon) \Longrightarrow (S, \ell)$  then there is  $(G, \epsilon) \Longrightarrow (G, \ell')$  with  $\ell \equiv_{G,\sigma} \ell'$

A  $(\sigma, G)$ -realisation is a swarm  $(S, \epsilon)$  such that, for each  $i \in \text{dom } S$ , there exists a role  $R \in \text{roles}(G, \sigma)$  such that  $S(i) = G \downarrow_R^\sigma$   $\square$

# Implementations & projections

Write  $\ell \equiv_{G,\sigma} \ell'$  when  $\ell$  and  $\ell'$  have the same effective type wrt  $G$  and  $\sigma$

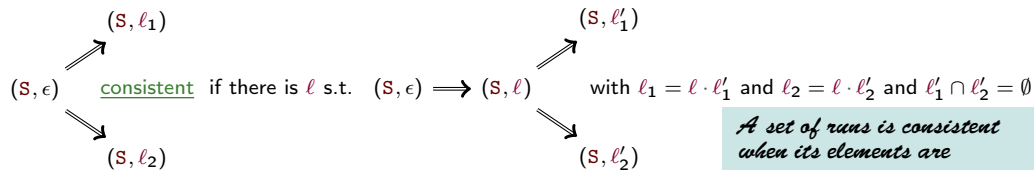
A swarm  $(S, \epsilon)$  is eventually faithful to  $G$  and  $\sigma$  if  $(S, \epsilon) \Longrightarrow (S, \ell)$  then there is  $(G, \epsilon) \Longrightarrow (G, \ell')$  with  $\ell \equiv_{G,\sigma} \ell'$

A  $(\sigma, G)$ -realisation is a swarm  $(S, \epsilon)$  such that, for each  $i \in \text{dom } S$ , there exists a role  $R \in \text{roles}(G, \sigma)$  such that  $S(i) = G \downarrow_R^\sigma$

Lemma (Projections of well-formed protocols are eventually faithful)

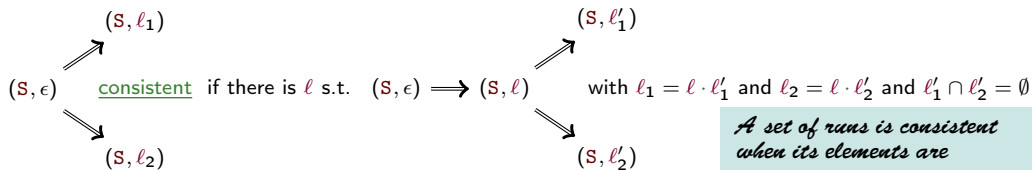
If  $G$  is a  $\sigma$ -WF protocol and  $(\delta(G \downarrow_R^\sigma, \ell)) \downarrow_{c/1}$  then there exists  $\ell' \equiv_{G,\sigma} \ell$  such that  $(G, \epsilon) \Longrightarrow (G, \ell')$  and  $\delta(G, \ell') \xrightarrow{c/1} G'$

## On correct realisations



*A set of runs is consistent  
when its elements are  
pair-wise consistent*

# On correct realisations

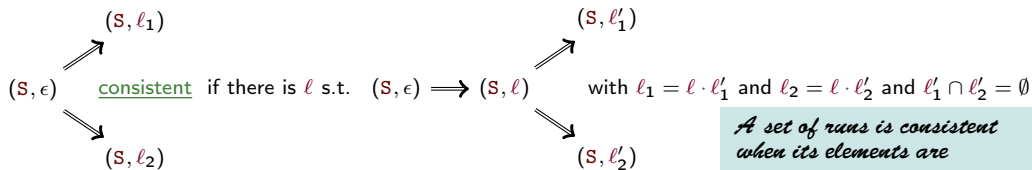


*A set of runs is consistent when its elements are pair-wise consistent*

## Notation

For  $(G, \epsilon) \xrightarrow{c_1 / l_1} (G, \ell_1) \xrightarrow{c_2 / l_2} \dots \xrightarrow{c_n / l_n} (G, \overbrace{\ell_1 \cdot \ell_2 \cdot \dots \cdot \ell_n}^{=\ell})$   
 let  $\ell^{(j)} = \ell_1 \cdot \dots \cdot \ell_j$

# On correct realisations



*A set of runs is consistent when its elements are pair-wise consistent*

## Notation

For  $(G, \epsilon) \xrightarrow{c_1 / l_1} (G, \ell_1) \xrightarrow{c_2 / l_2} \dots \xrightarrow{c_n / l_n} (G, \overbrace{\ell_1 \cdot \ell_2 \cdot \dots \cdot \ell_n}^{=\ell})$   
 let  $\ell^{(j)} = \ell_1 \cdot \dots \cdot \ell_j$

## Admissibility

A log  $\ell$  is admissible for a  $\sigma$ -WF protocol  $G$  if there are consistent runs  $\{(G, \epsilon) \implies (G, \ell_i)\}_{1 \leq i \leq k}$  and a log  $\ell' \in (\boxtimes_{1 \leq i \leq k} \ell_i)$  such that

$$\ell = \bigcup_{1 \leq i \leq k} \ell_i, \quad \ell' \equiv_{G, \sigma} \ell, \quad \text{and} \quad \ell_i^{(j)} \sqsubseteq \ell \text{ for all } 1 \leq i \leq k$$

# Results

Let  $G$  be well-formed; a realisation is a swarm whose components are projections of  $G$

**Lemma (Well-formedness generates any admissible log)**

*If  $\ell$  is admissible for  $G$  then there is a log  $\ell'$  such that  $(G, \epsilon) \implies (G, \ell')$  and  $\ell \equiv_{G, \sigma} \ell'$*

**Theorem (Well-formed protocols generate only admissible logs)**

*If  $(S, \epsilon) \implies (S', \ell)$  for  $(S, \epsilon)$  realisation of  $G$  then  $\ell$  is admissible for  $G$*

**Corollary**

*Every realisation of  $G$  is eventually faithful wrt  $G$  and  $\sigma$*

**Theorem (Full realisations are complete)**

*If  $S$  is a full realisation of  $G$  and  $(G, \epsilon) \implies (G, \ell')$  then there is  $S'$  s.t.  $(S, \epsilon) \implies (S', \ell)$*

# Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work

– Tooling –



```
// 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
```

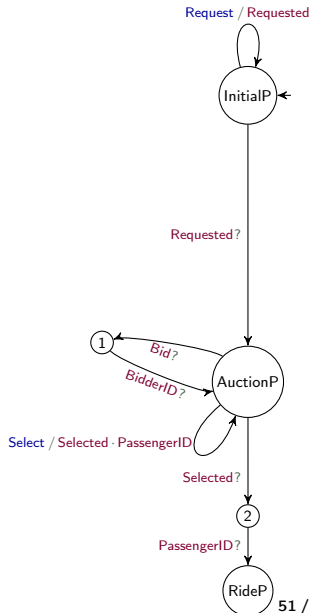
```
export class InitialP extends State<Events> {
  constructor(public id: string) { super() }
  execRequest(pickup: string, dest: string) {
    return this.events({ type: 'Requested', pickup, dest })
  }
  onRequest(ev: Requested) {
    return new AuctionP(this.id, ev.pickup, ev.dest, [])
  }
}
```

```
@proto('taxiRide')
```

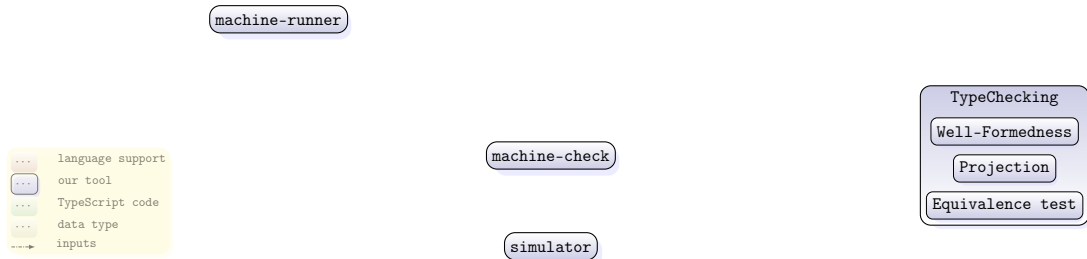
```
export class AuctionP extends State<Events> {
  constructor(public id: string, public pickup: string, public dest: string,
    public bids: BidData[]) { super() }
  onBid(ev1: Bid, ev2: BidderID) {
    const [ price, time ] = ev1
    this.bids.push({ price, time, bidderID: ev2.id })
    return this
  }
  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> { ... }
```

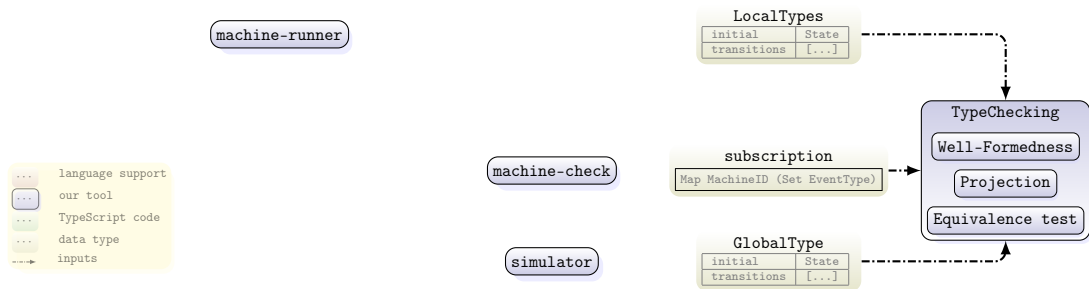


# Architecture



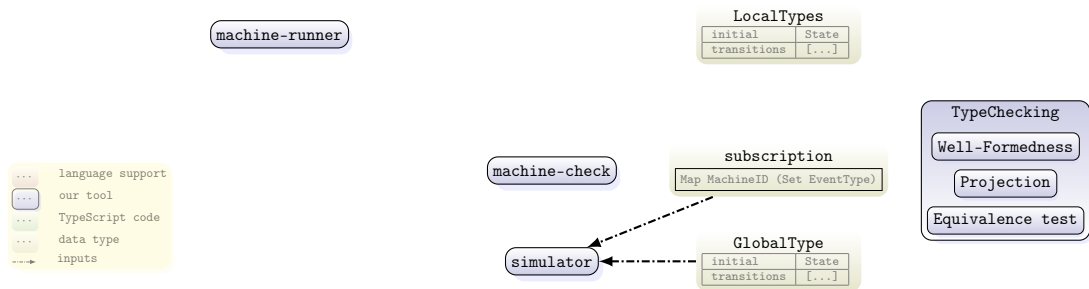
- TypeChecking implements the functionalities of our typing discipline
- simulator simulates the semantics of swarm realisations
- machine-check and machine-runner integrate our framework in the Actyx platform

# Architecture



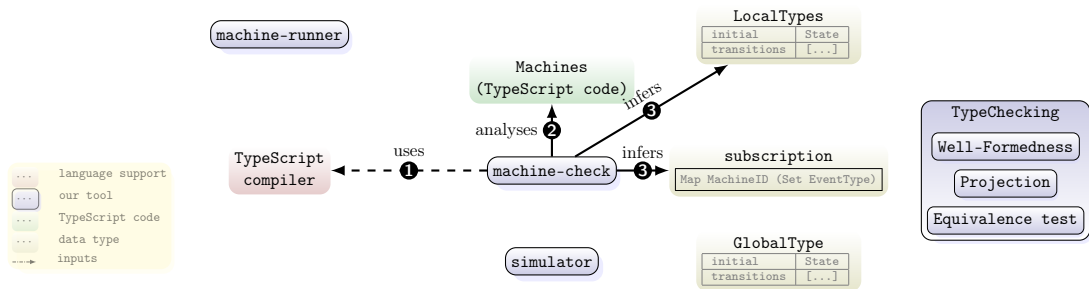
- TypeChecking implements the functionalities of our typing discipline
- simulator simulates the semantics of swarm realisations
- machine-check and machine-runner integrate our framework in the Actyx platform

# Architecture



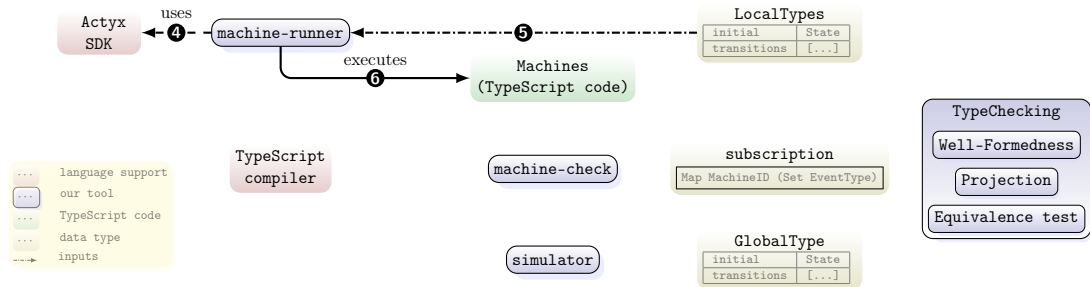
- TypeChecking implements the functionalities of our typing discipline
- simulator simulates the semantics of swarm realisations
- machine-check and machine-runner integrate our framework in the Actyx platform

# Architecture



- 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



- 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

# If you want to play with our prototype?

Have a look at

- our ECOOP artifact paper (not online yet; extended version at <https://arxiv.org/abs/2305.04848>)
- code at <https://doi.org/10.5281/zenodo.7737188>
- An ISSTA tool paper from Actyx (<https://arxiv.org/abs/2306.09068>)

# Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work



– Epilogue –

To be continued....

There are a number of future directions to explore:

## To be continued....

There are a number of future directions to explore:

Identify weaker conditions for well-formedness

## To be continued....

There are a number of future directions to explore:

Identify weaker conditions for well-formedness

“Efficiency”

## To be continued....

There are a number of future directions to explore:

- Identify weaker conditions for well-formedness

- “Efficiency”

- Subscriptions are hard to determine

## To be continued....

There are a number of future directions to explore:

- Identify weaker conditions for well-formedness

- “Efficiency”

- Subscriptions are hard to determine

- Relax some of our assumptions

## To be continued....

There are a number of future directions to explore:

- Identify weaker conditions for well-formedness

- “Efficiency”

- Subscriptions are hard to determine

- Relax some of our assumptions

  - Compensations

  - Unreliable propagation

## To be continued....

There are a number of future directions to explore:

- Identify weaker conditions for well-formedness

- “Efficiency”

- Subscriptions are hard to determine

- Relax some of our assumptions

  - Compensations

  - Unreliable propagation

  - Adversarial contexts



## To be continued....

There are a number of future directions to explore:

- Identify weaker conditions for well-formedness

- “Efficiency”

- Subscriptions are hard to determine

- Relax some of our assumptions

  - Compensations

  - Unreliable propagation

  - Adversarial contexts

- .....

## Summary

An interesting paradigm grounded on principles for local-first principles: temporary inconsistency are tolerated provided that they can be (and are) resolved at some point

## Summary

An interesting paradigm grounded on principles for local-first principles: temporary inconsistency are tolerated provided that they can be (and are) resolved at some point

A formal semantics that faithfully captures Actyx's platform

An interesting paradigm grounded on principles for local-first principles: temporary inconsistency are tolerated provided that they can be (and are) resolved at some point

A formal semantics that faithfully captures Actyx's platform

and behavioural types to specify and verify eventual consensus

*Thank you!*