

Formal Methods for Communication Protocols

Harnessing distributed software design with behavioural contracts

Emilio Tuosto @ GSSI

– Lecture 3 –

3 - 12 March, 2026 - Novi Sad



Finanziato
dall'Unione europea
NextGenerationEU



Ministero
dell'Università
e della Ricerca



Italiadomani
PIANO NAZIONALE
DI SVILUPPO E INNOVAZIONE



Developing
Shared
Knowledge

design (well)

+

project

+

run

execute

+

propagate

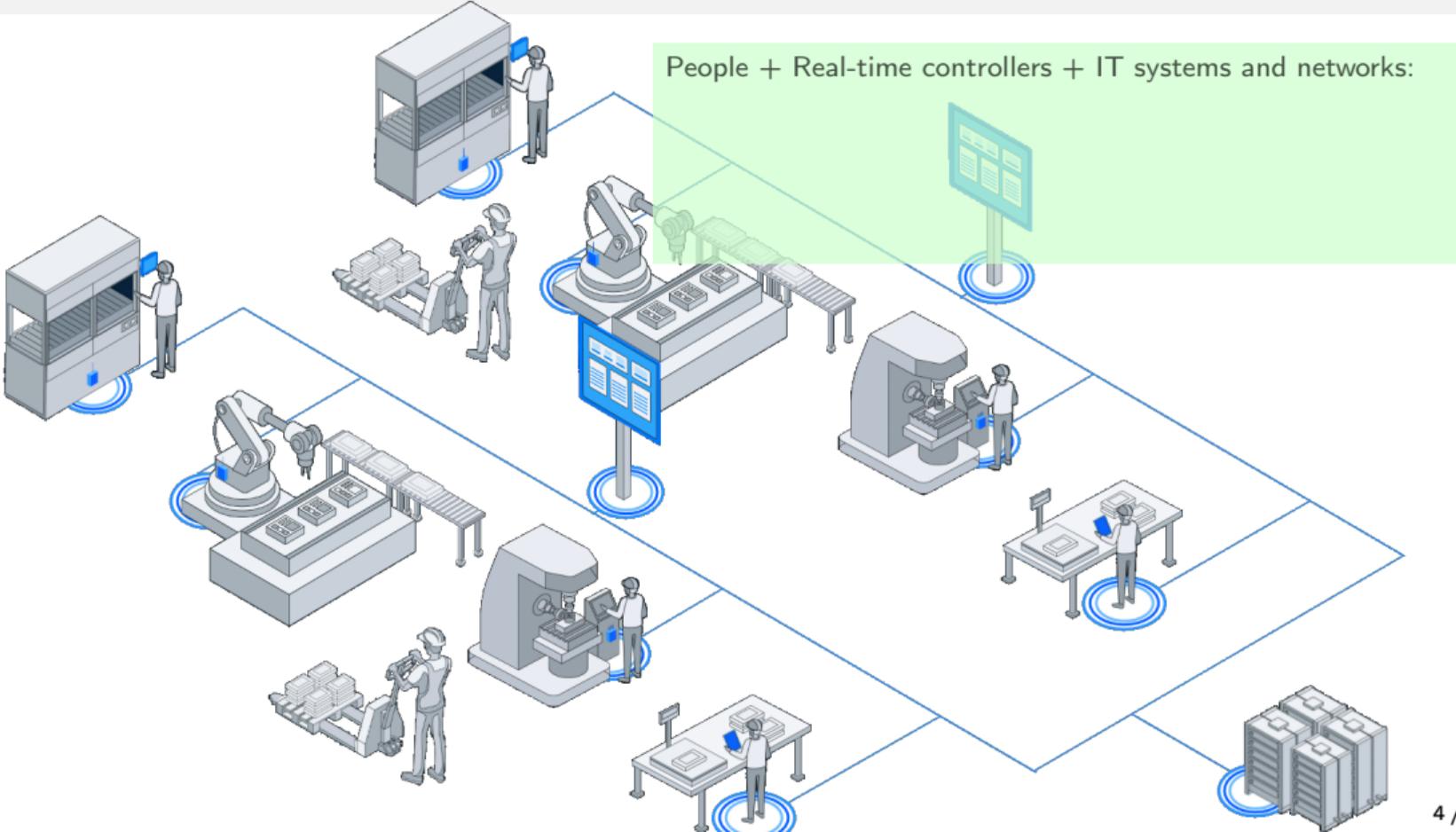
+

merge

– Swarms and Swarm Protocols –

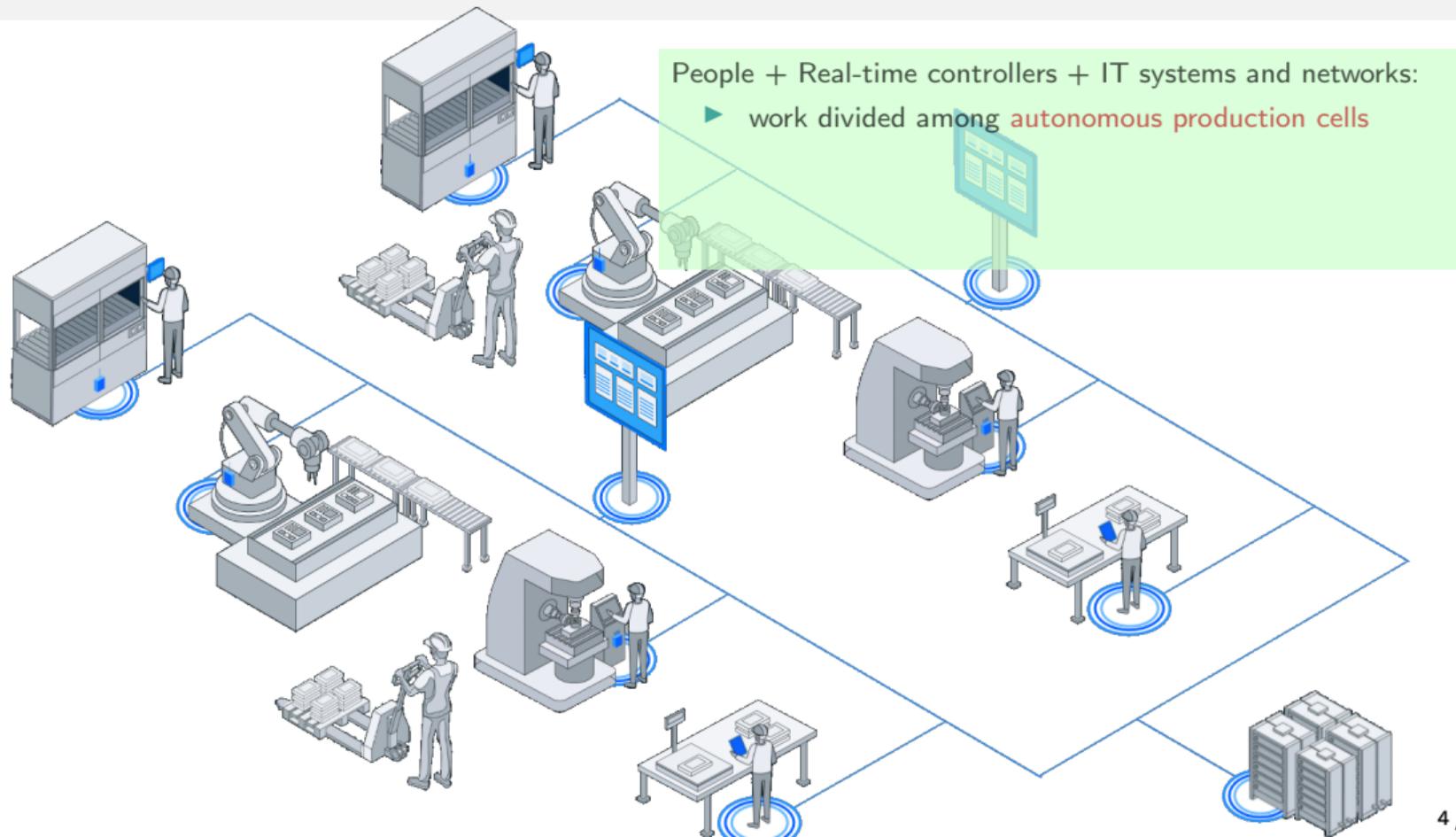
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



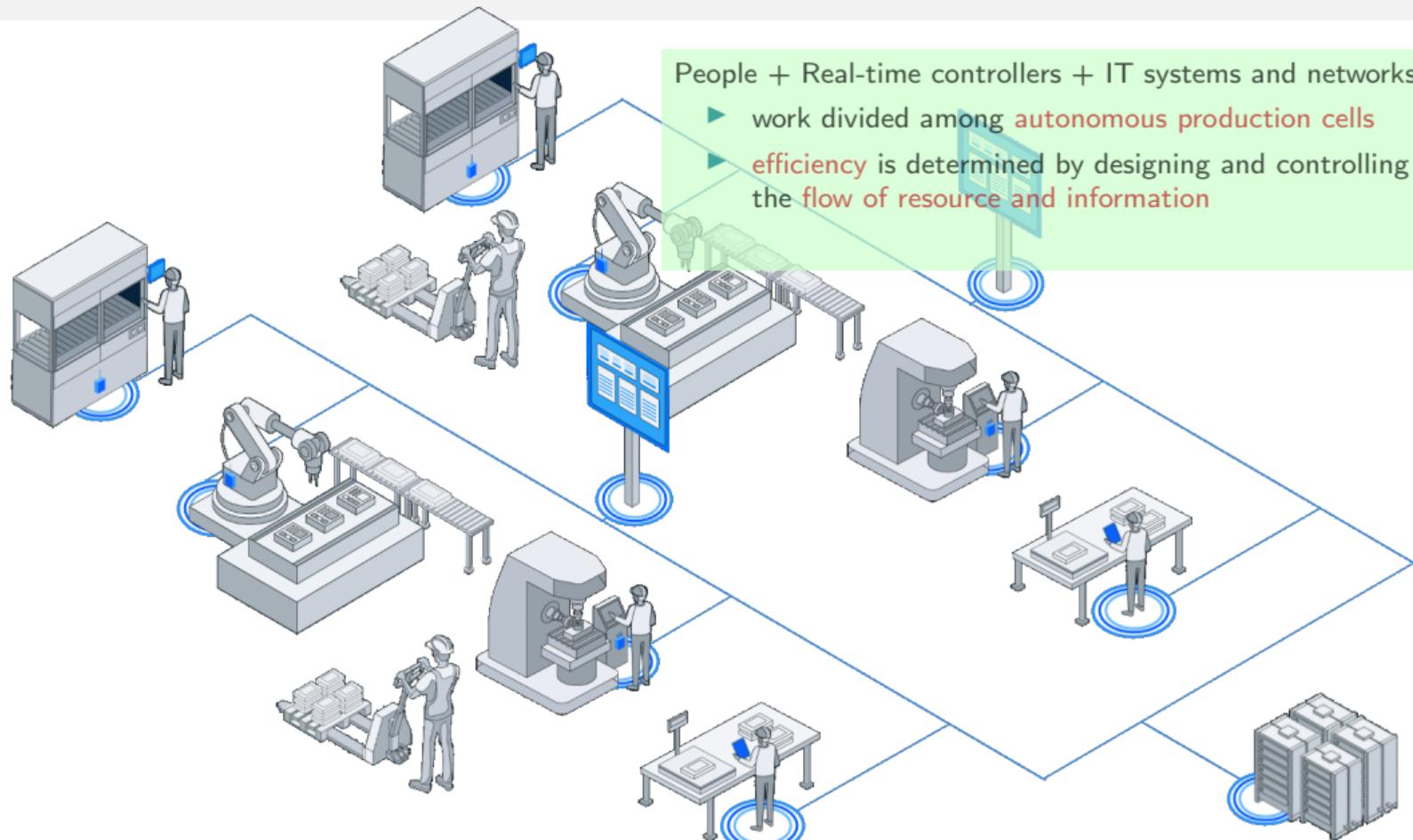
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



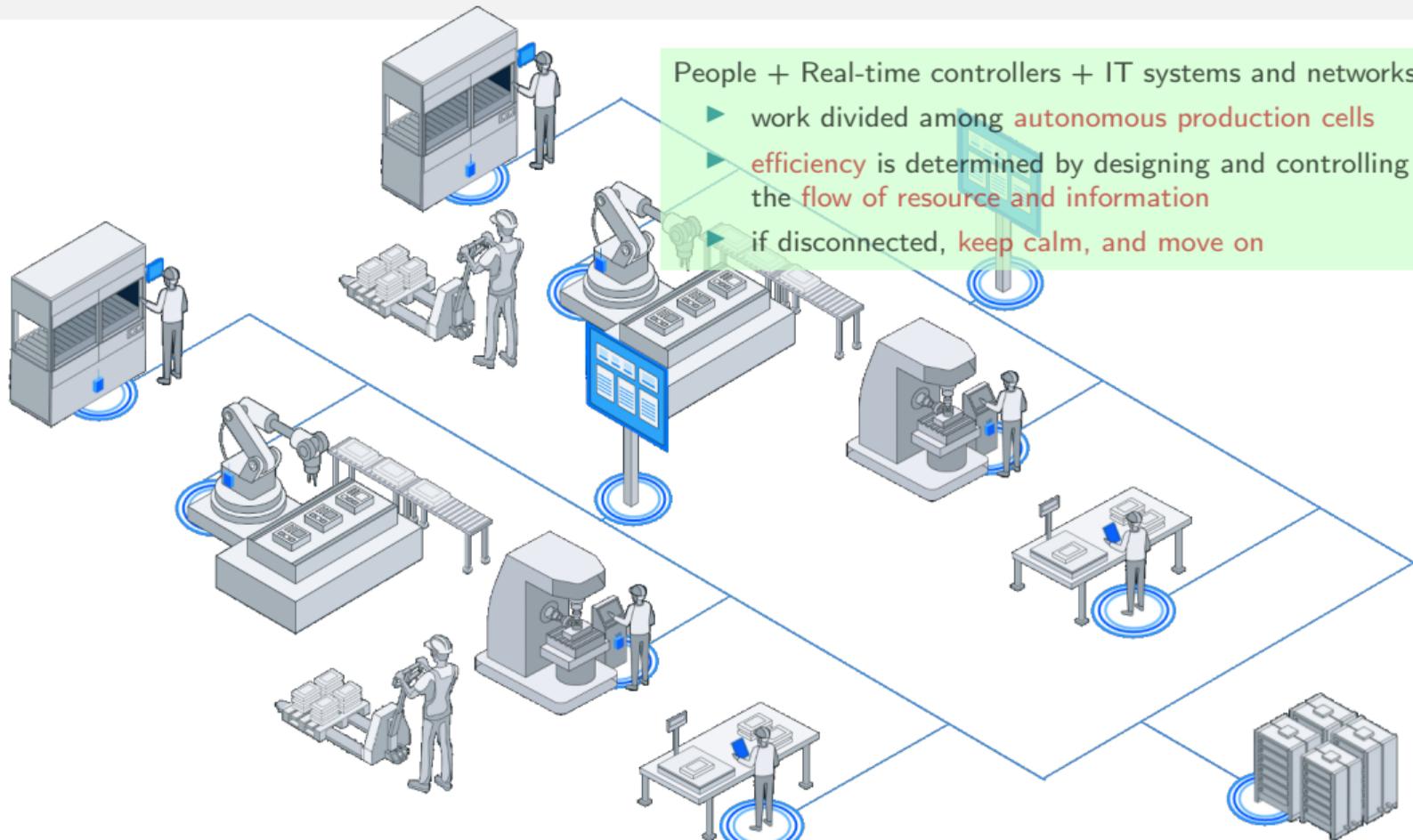
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



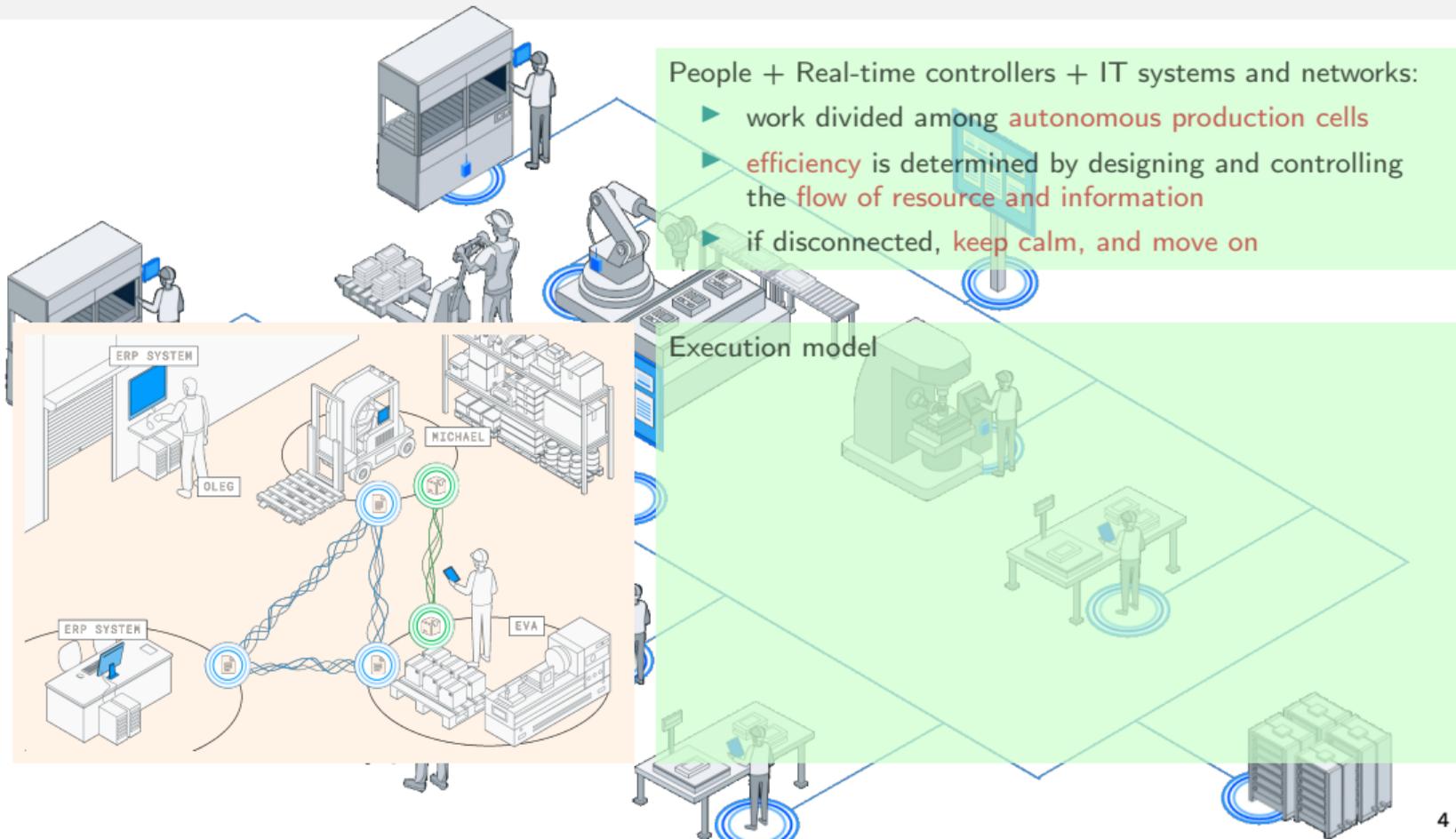
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



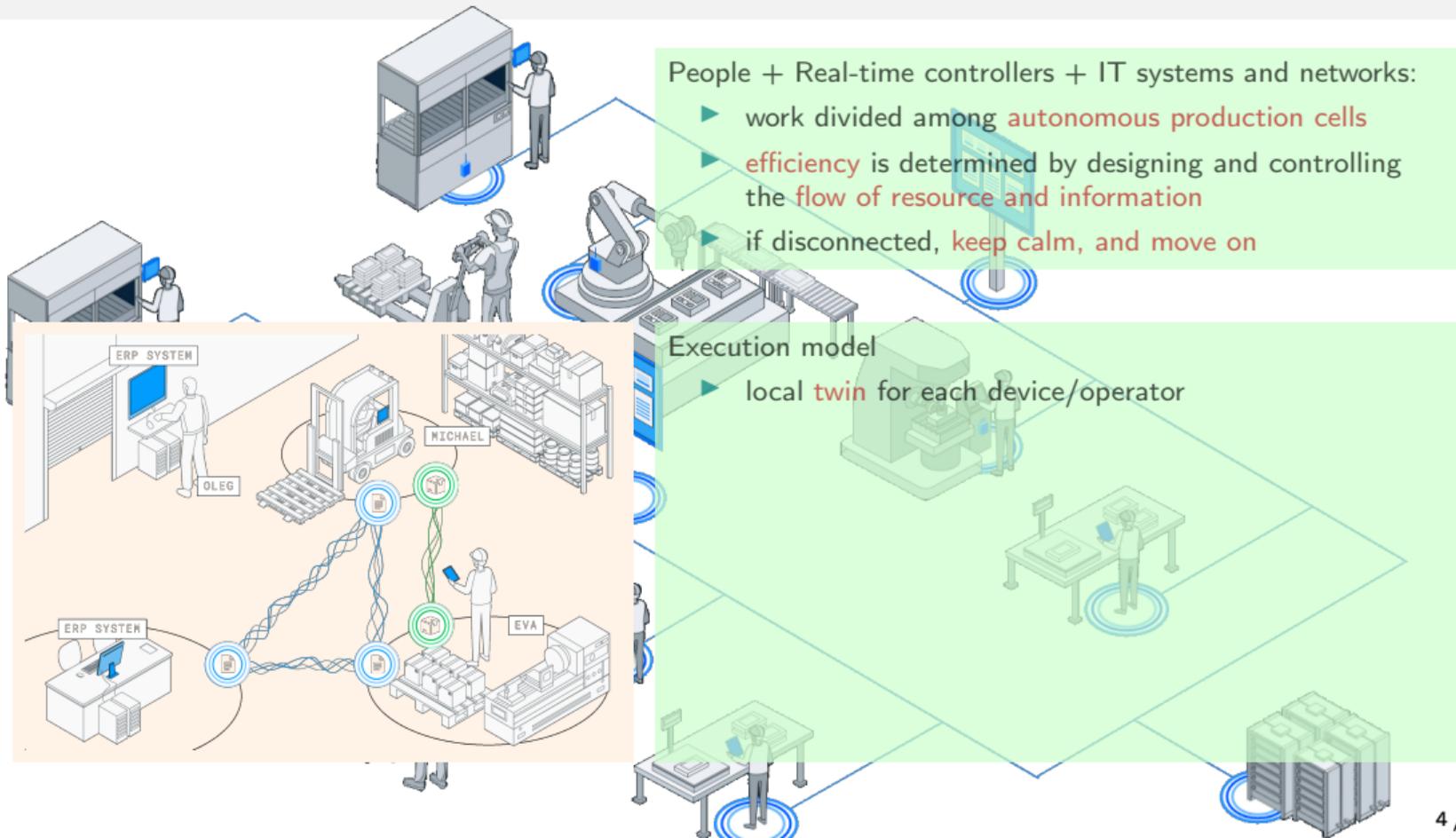
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



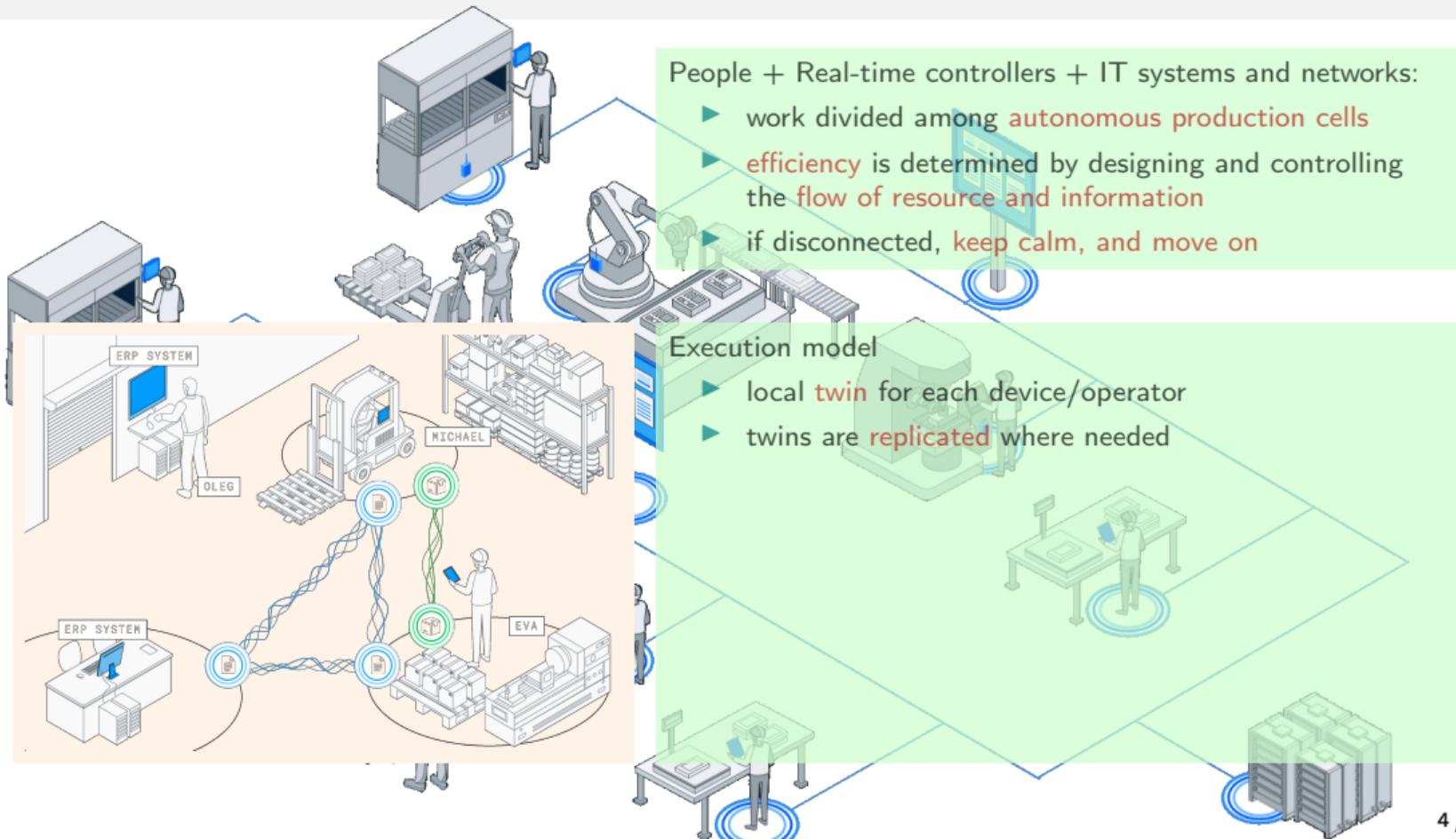
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



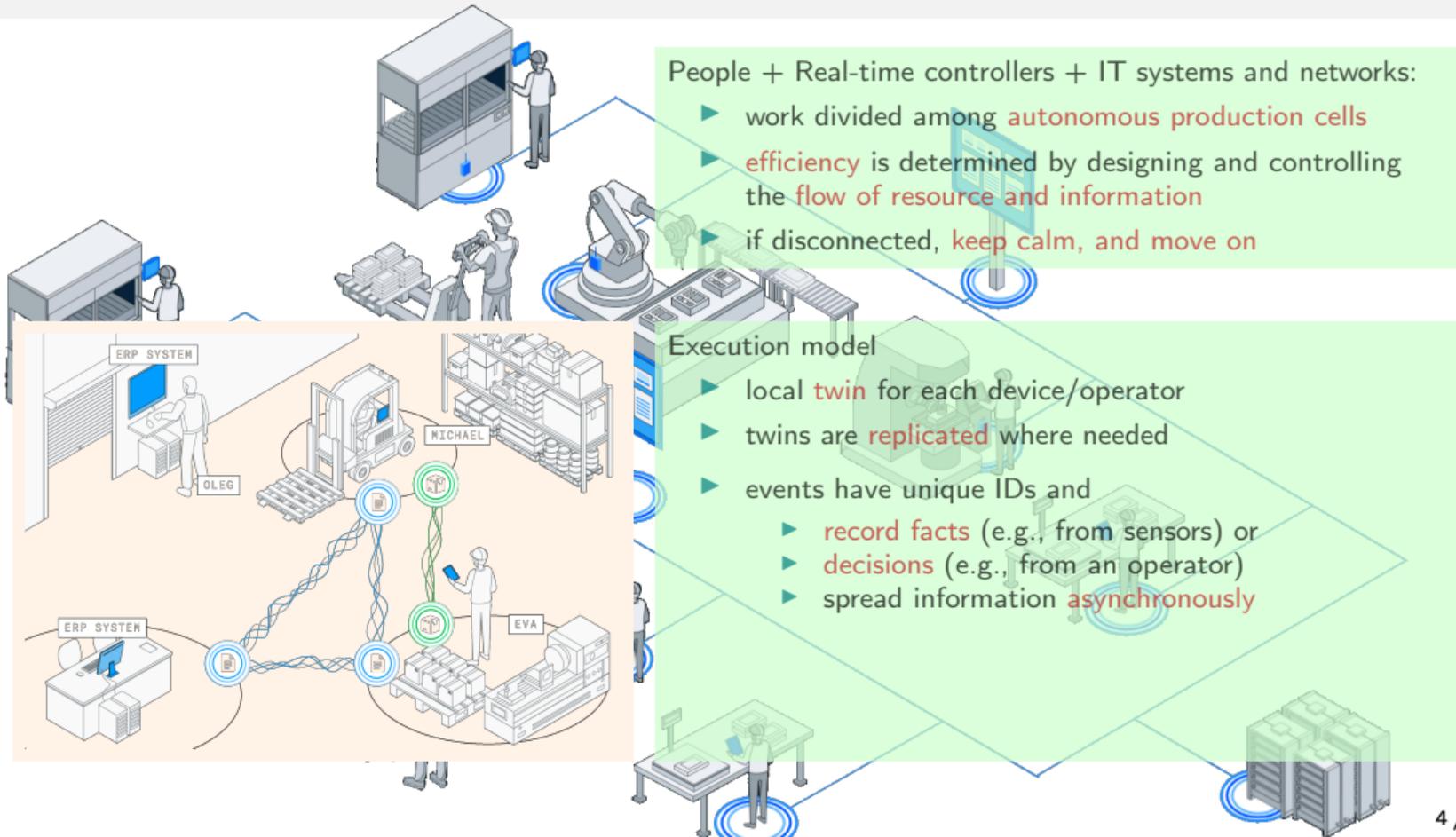
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



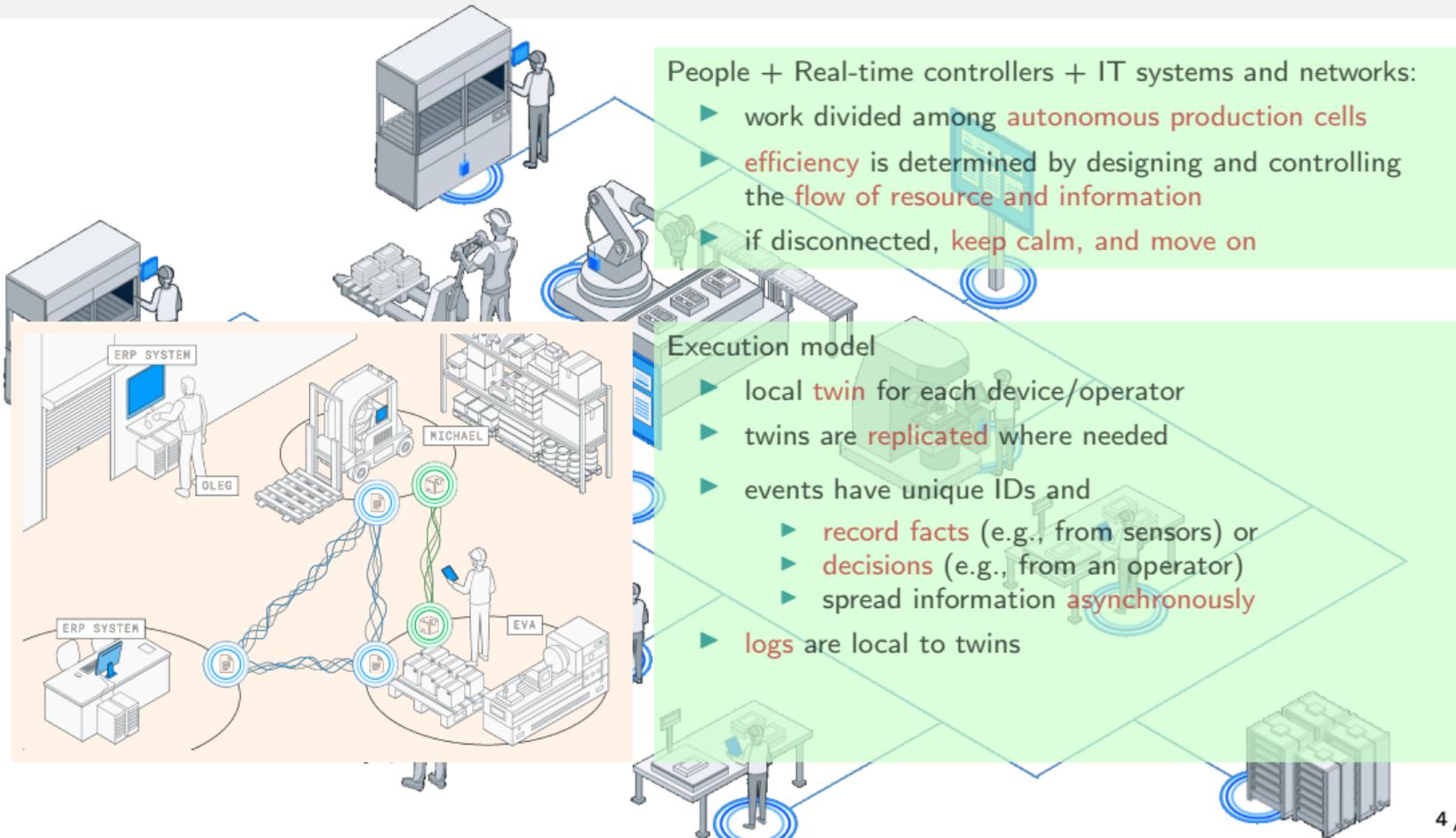
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



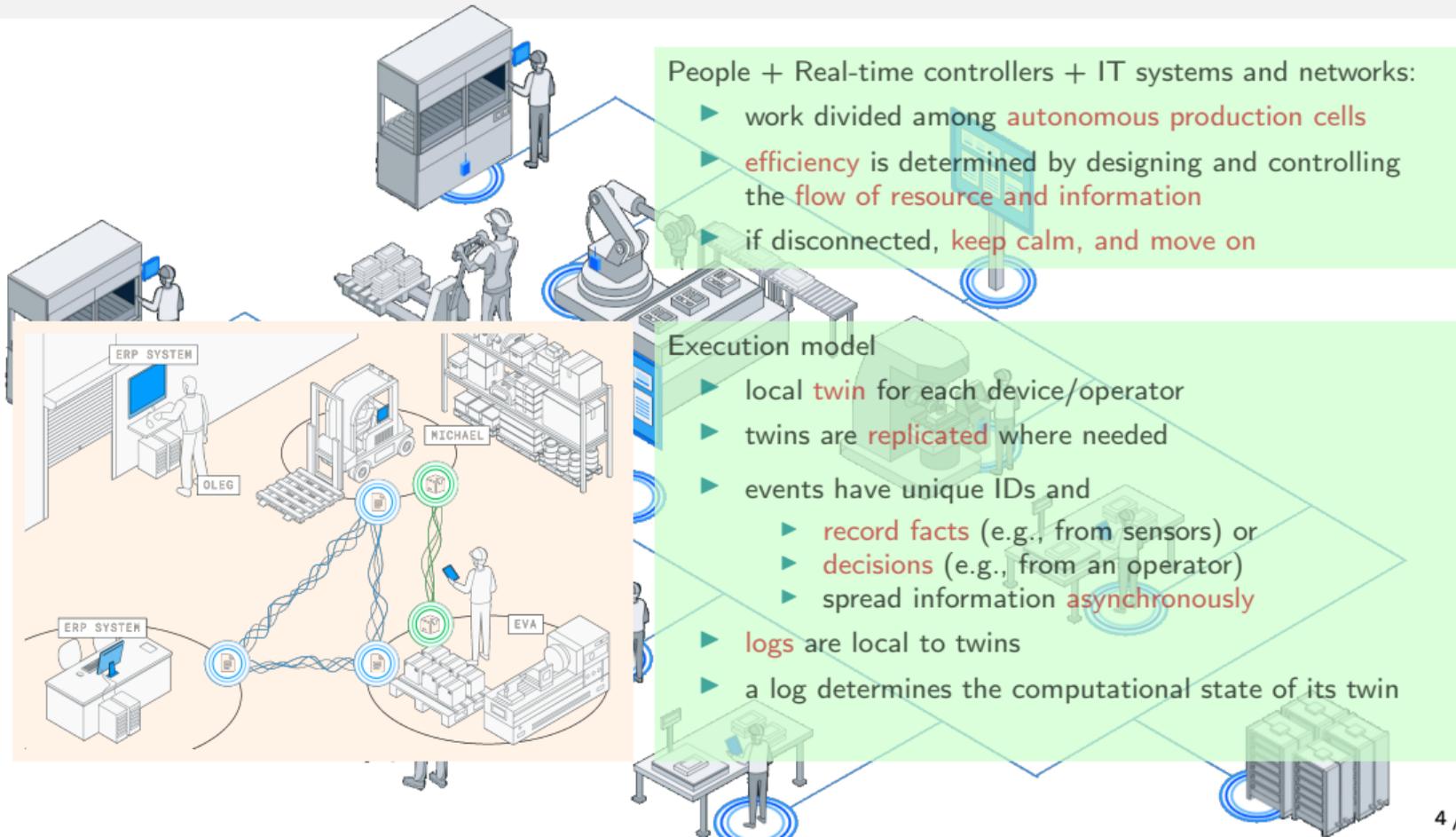
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



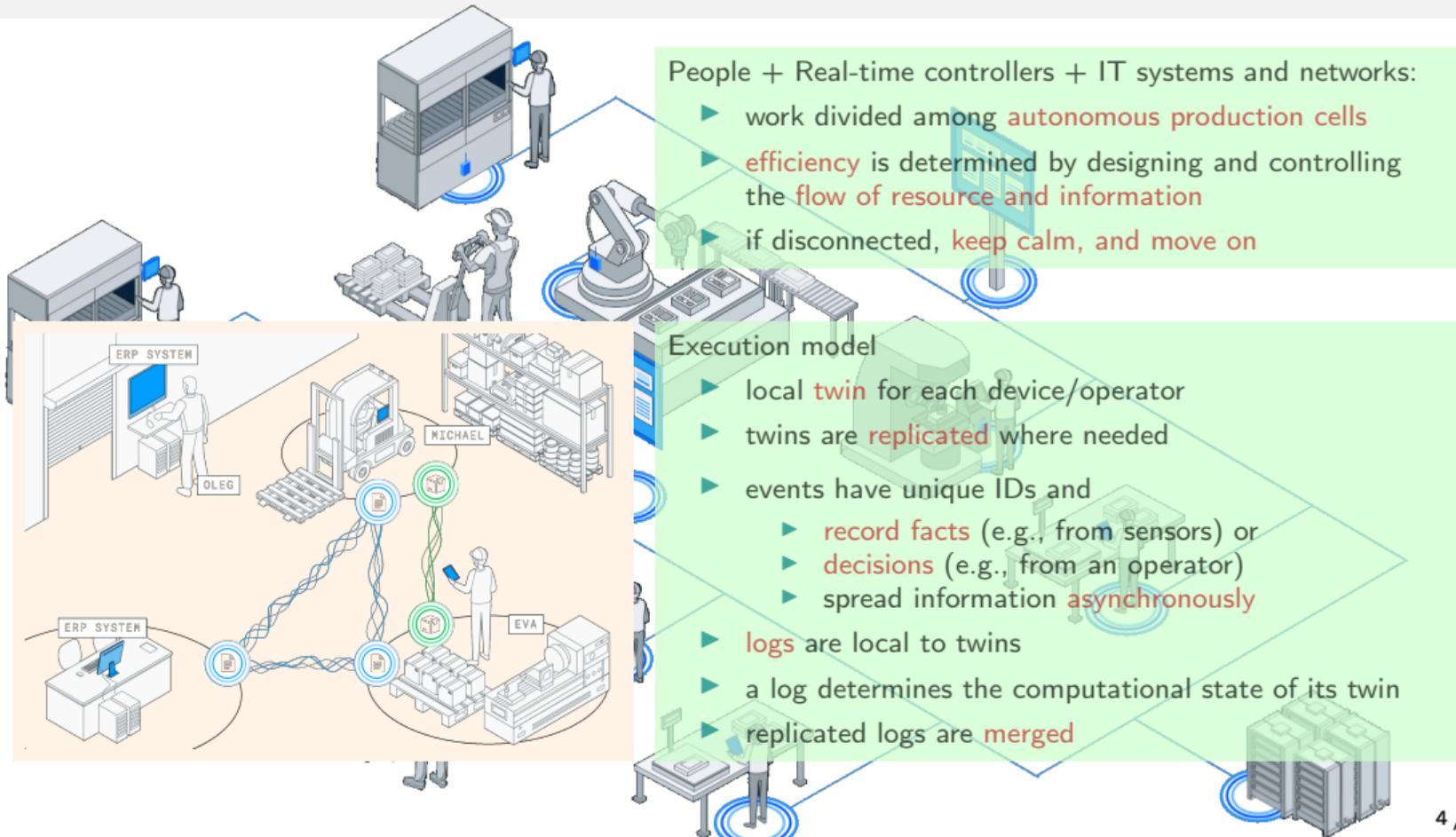
An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



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 [1]

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

Events

e

$src(e)$

Logs

$e_1 \cdot e_2 \cdot \dots$

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 $\ell = e_1 \dots e_n$ $e_i <_\ell e_j \iff i < j$

Ingredients (II): log shipping

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

Ingredients (II): log shipping

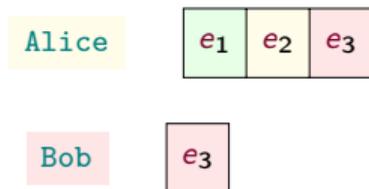
Machines **emit** logs upon **execution of commands** (we'll see how in a moment)

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**:

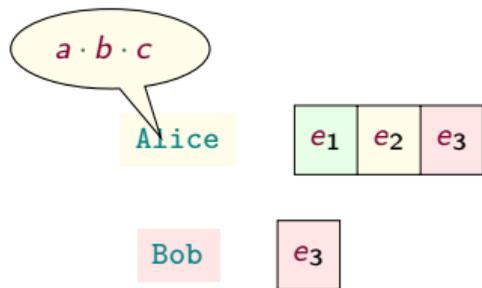
1st Phase: emitted events are appended to the local log of the emitting machine



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**:

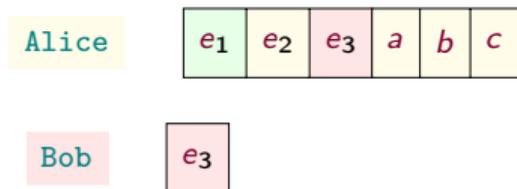
1st Phase: emitted events are appended to the local log of the emitting machine



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

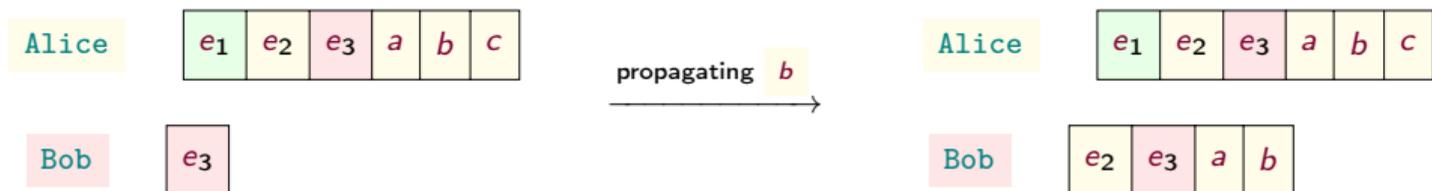


Ingredients (II): log shipping

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

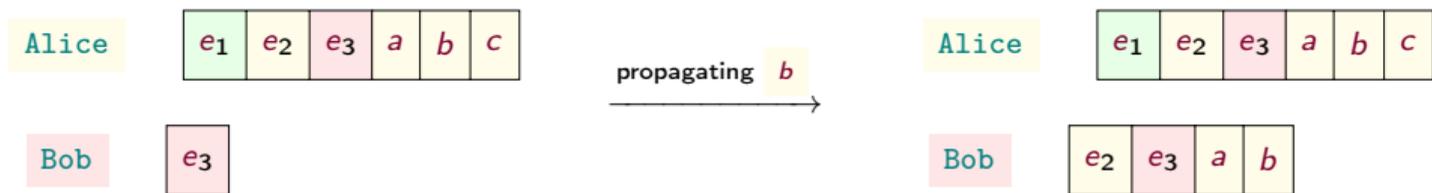


Ingredients (II): log shipping

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

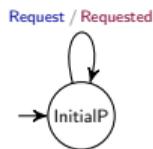
Let's build a machine `InitialP` for coordinating a taxi service.



`InitialP` =

Machines' syntax by example

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



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

Machines' syntax by example

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



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

`AuctionP` =

Machines' syntax by example

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

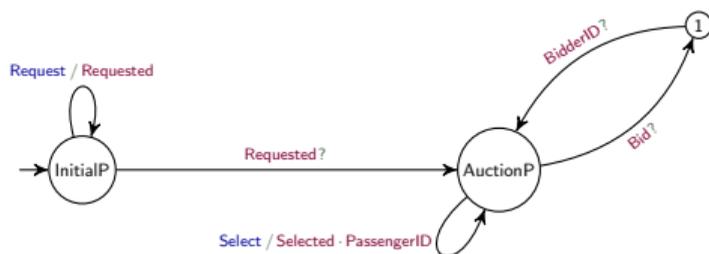


`InitialP` = `Request` \mapsto `Requested` · [`Requested?` `AuctionP`]

`AuctionP` = `Select` \mapsto `Selected` · `PassengerID` ·

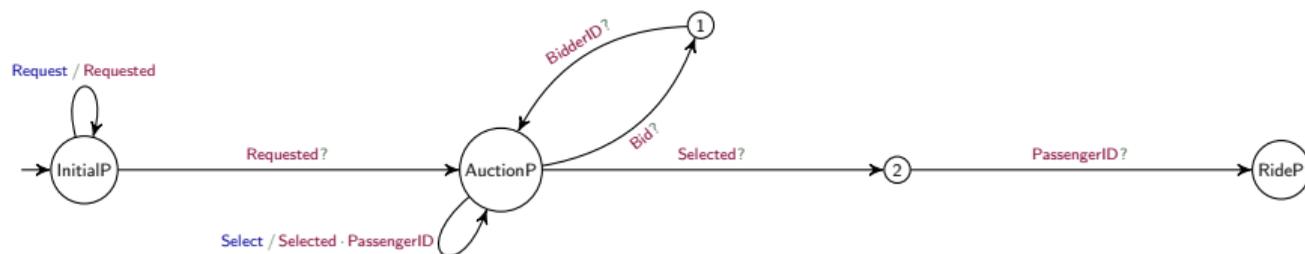
Machines' syntax by example

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


$$\text{InitialP} = \text{Request} \mapsto \text{Requested} \cdot [\text{Requested? } \underline{\text{AuctionP}}]$$
$$\text{AuctionP} = \text{Select} \mapsto \text{Selected} \cdot \text{PassengerId} \cdot [\text{Bid? BidderId? } \underline{\text{AuctionP}}]$$

Machines' syntax by example

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



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

`AuctionP` = `Select` \mapsto `Selected` · `PassengerID` · [
 `Bid?` `BidderID?` `AuctionP`
 &
 `Selected?` `PassengerID?` `RideP`
]

`RideP` = ...

Machines' semantics

So, think of $M = \kappa \cdot [t_1? M_1 \& \dots \& t_n? M_n]$ 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 t_i)

Machines' semantics

So, think of $M = \kappa \cdot [t_1? M_1 \& \dots \& t_n? M_n]$ 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 t_i)

State transition function :

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

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

Machines' semantics

So, think of $M = \kappa \cdot [t_1?M_1 \& \dots \& t_n?M_n]$ 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 t_i)

State transition function :

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

That is

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

Machines' semantics

So, think of $M = \kappa \cdot [t_1?M_1 \& \dots \& t_n?M_n]$ 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 t_i)

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

Machines' semantics

So, think of $M = \kappa \cdot [t_1?M_1 \& \dots \& t_n?M_n]$ 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 t_i)

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

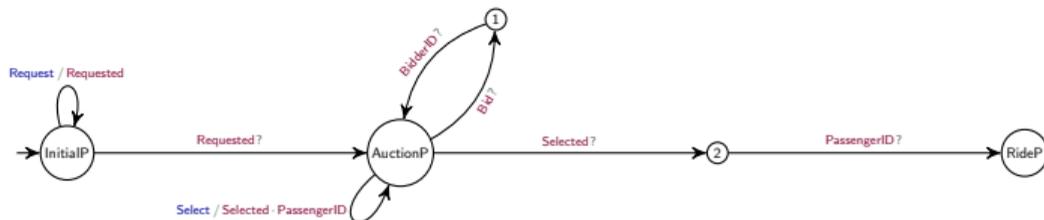
M with local log ℓ is in the implicit state $\delta(M, \ell)$ reached after processing each event in ℓ

That is

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

An example

Take the machine `InitialP` (slide 9) with a local log $\ell = \text{ignoreMe} \cdot \text{ignoreMeToo}$ where $\not\vdash \text{ignoreMe} : \text{Requested}$ and $\not\vdash \text{ignoreMeToo} : \text{Requested}$

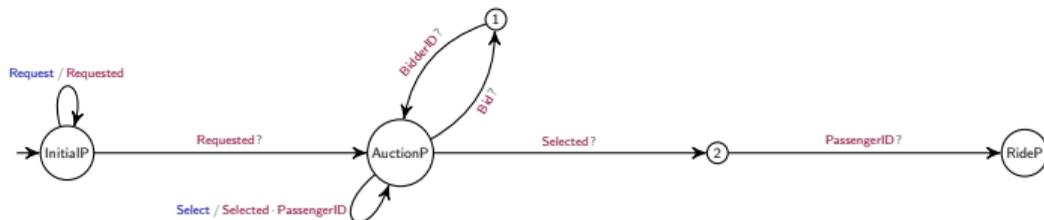


By definition of δ

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

An example

Take the machine `InitialP` (slide 9) with a local log $\ell = \text{ignoreMe} \cdot \text{ignoreMeToo}$ where $\nexists \text{ignoreMe} : \text{Requested}$ and $\nexists \text{ignoreMeToo} : \text{Requested}$



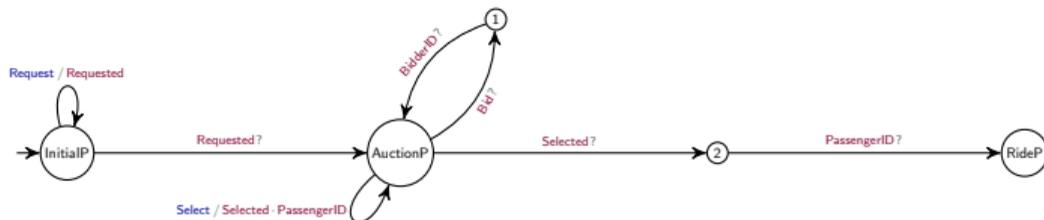
By definition of δ

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

hence

An example

Take the machine `InitialP` (slide 9) with a local log $\ell = \text{ignoreMe} \cdot \text{ignoreMeToo}$ where $\nVdash \text{ignoreMe} : \text{Requested}$ and $\nVdash \text{ignoreMeToo} : \text{Requested}$



By definition of δ

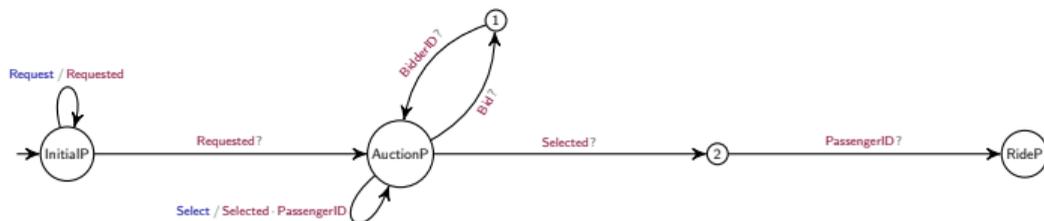
- ▶ $\delta(\text{InitialP}, \ell) = \text{InitialP}$
- ▶ $\delta(\text{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} \delta(\text{InitialP}, \ell)$
- ▶ $(\text{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} (\text{InitialP}, \ell \cdot \text{Requested})$
with $\vdash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

hence

hence

An example

Take the machine `InitialP` (slide 9) with a local log $\ell = \text{ignoreMe} \cdot \text{ignoreMeToo}$ where $\nVdash \text{ignoreMe} : \text{Requested}$ and $\nVdash \text{ignoreMeToo} : \text{Requested}$



By definition of δ

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

hence

hence

Exercise

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

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 (\mathbf{S}, ℓ) where

- ▶ \mathbf{S} maps each index $1 \leq i \leq n$ to a pair (M_i, ℓ_i)
- ▶ ℓ is the (global) log

Swarms

A swarm (of size n) is a pair (\mathbf{S}, ℓ) where

- ▶ \mathbf{S} maps each index $1 \leq i \leq n$ to a pair (M_i, ℓ_i)
- ▶ ℓ is the (global) log

Notation

$$M_1 \boxed{\ell_1} \mid \dots \mid M_n \boxed{\ell_n} \mid \ell$$

Swarms

A swarm (of size n) is a pair (\mathbf{S}, ℓ) where

- ▶ \mathbf{S} maps each index $1 \leq i \leq n$ to a pair (M_i, ℓ_i)
- ▶ ℓ is the (global) log

Notation

$M_1 \boxed{\ell_1} | \dots | M_n \boxed{\ell_n} | \ell$

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 (\mathbf{S}, ℓ) where

- ▶ \mathbf{S} maps each index $1 \leq i \leq n$ to a pair (M_i, ℓ_i)
- ▶ ℓ is the (global) log

Notation

$M_1 \boxed{\ell_1} | \dots | M_n \boxed{\ell_n} | \ell$

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 (\mathbf{S}, ℓ) where

- ▶ \mathbf{S} maps each index $1 \leq i \leq n$ to a pair (M_i, ℓ_i)
- ▶ ℓ is the (global) log

Notation

$M_1 \boxed{\ell_1} | \dots | M_n \boxed{\ell_n} | \ell$

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 $M_1 \ell_1 \mid \dots \mid M_n \ell_n \mid \ell$ 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 $M_1 \boxed{\ell_1} \mid \dots \mid M_n \boxed{\ell_n} \mid \ell$ is coherent if $\ell = \bigcup_{1 \leq i \leq n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \leq i \leq n$

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 ℓ_1 appear in the same order in ℓ_2

That is

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

Coherence

A swarm $M_1 \boxed{\ell_1} \mid \dots \mid M_n \boxed{\ell_n} \mid \ell$ is coherent if $\ell = \bigcup_{1 \leq i \leq n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \leq i \leq n$

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 ℓ_1 appear in the same order in ℓ_2

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 8 and consider a swarm

$$\dots | \text{Alice} \begin{array}{|c|c|c|c|c|c|} \hline e_1 & e_2 & e_3 & a & b & c \\ \hline \end{array} | \dots | e_1 \cdot e_2 \cdot e_3 \cdot \ell \quad (1)$$

Under which condition is (1) coherent?

Merging logs

Exercise

Recall slide 8 and consider a swarm

$$\dots \mid \begin{array}{|c|} \hline \text{Alice} \\ \hline \end{array} \begin{array}{|c|c|c|c|c|c|} \hline e_1 & e_2 & e_3 & a & b & c \\ \hline \end{array} \mid \dots \mid e_1 \cdot e_2 \cdot e_3 \cdot l \quad (1)$$

Under which condition is (1) coherent?

The propagation of newly generated events happens by merging logs:

Log merging: $l_1 \bowtie l_2 = \{l \mid l \subseteq l_1 \cup l_2 \text{ and } l_1 \sqsubseteq l \text{ and } l_2 \sqsubseteq l\}$

Semantics of swarms

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

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

Semantics of swarms

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

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

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

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

Semantics at work (I)

If

$$B \boxed{b} \xrightarrow{c/1} B \boxed{b \cdot d \cdot e} \quad \text{with} \quad \vdash d \cdot e : 1$$

Semantics at work (I)

If

$$B \boxed{b} \xrightarrow{c/1} B \boxed{b \cdot d \cdot e} \quad \text{with} \quad \vdash d \cdot e : 1$$

then, by [Local]

$$A \boxed{a} \mid B \boxed{b} \mid C \boxed{c} \mid b \cdot a \cdot c \xrightarrow{c/1} A \boxed{a} \mid B \boxed{b \cdot d \cdot e} \mid C \boxed{c} \mid \ell$$

Semantics at work (I)

If

$$B \boxed{b} \xrightarrow{c/l} B \boxed{b \cdot d \cdot e} \quad \text{with} \quad \vdash d \cdot e : l$$

then, by [Local]

$$A \boxed{a} \mid B \boxed{b} \mid C \boxed{c} \mid b \cdot a \cdot c \xrightarrow{c/l} A \boxed{a} \mid B \boxed{b \cdot d \cdot e} \mid C \boxed{c} \mid \ell$$

for all

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

Semantics at work (I)

If

$$B \boxed{b} \xrightarrow{c/1} B \boxed{b \cdot d \cdot e} \quad \text{with} \quad \vdash d \cdot e : 1$$

then, by [Local]

$$A \boxed{a} \mid B \boxed{b} \mid C \boxed{c} \mid b \cdot a \cdot c \xrightarrow{c/1} A \boxed{a} \mid B \boxed{b \cdot d \cdot e} \mid C \boxed{c} \mid \ell$$

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 17, consider

$$A[a] \mid B[b] \mid C[c] \mid b \cdot a \cdot c \xrightarrow{c/1} A[a] \mid B[b \cdot d \cdot e] \mid C[c] \mid \overbrace{b \cdot a \cdot d \cdot e \cdot c}^{=l}$$

Semantics at work (II)

With reference to slide 17, consider

$$A[a] \mid B[b] \mid C[c] \mid b \cdot a \cdot c \xrightarrow{c/1} A[a] \mid B[b \cdot d \cdot e] \mid C[c] \mid \overbrace{b \cdot a \cdot d \cdot e \cdot c}^{=l}$$

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 17, consider

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

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

Let's propagate $d \cdot e$

$$A[a] | B[b \cdot d \cdot e] | C[c] | \ell \begin{array}{l} \xrightarrow{\tau} A[b \cdot a \cdot d \cdot e] | B[b \cdot d \cdot e] | C[c] | \ell \\ \xrightarrow{\tau} A[a] | B[b \cdot d \cdot e] | C[b \cdot d \cdot e \cdot c] | \ell \end{array}$$

Semantics at work (II)

With reference to slide 17, consider

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

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

Let's propagate $d \cdot e$

$$A[a] | B[b \cdot d \cdot e] | C[c] | \ell \begin{array}{l} \xrightarrow{\tau} A[b \cdot a \cdot d \cdot e] | B[b \cdot d \cdot e] | C[c] | \ell \\ \xrightarrow{\tau} A[a] | B[b \cdot d \cdot e] | C[b \cdot d \cdot e \cdot c] | \ell \end{array}$$

Exercise

In both cases b must be shipped too. Why?

And why is event a not shipped to C together with the events from B ?

Semantics at work (II)

With reference to slide 17, consider

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

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

Let's propagate $d \cdot e$

$$A[a] | B[b \cdot d \cdot e] | C[c] | \ell \begin{array}{l} \xrightarrow{\tau} A[b \cdot a \cdot d \cdot e] | B[b \cdot d \cdot e] | C[c] | \ell \\ \xrightarrow{\tau} A[a] | B[b \cdot d \cdot e] | C[b \cdot d \cdot e \cdot c] | \ell \end{array}$$

Exercise

In both cases b must be shipped too. Why?

And why is event a not shipped to C together with the events from B ?

Exercise

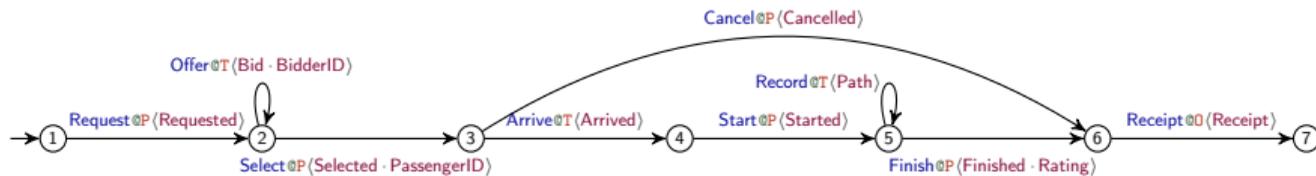
Can we propagate just event e ?

Swarm protocols as FSA

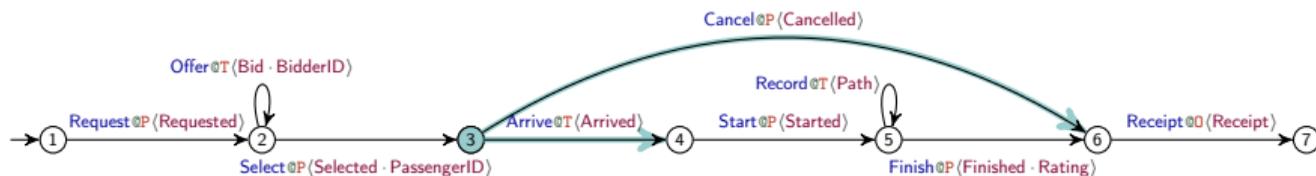
Like for machines, a swarm protocols $G = \sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle$. G_i has an associated FSA:

- ▶ 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 $c_i @ R_i \langle \mathbf{1}_i \rangle$, written $G \xrightarrow{c_i / \mathbf{1}_i} G_i$

An example



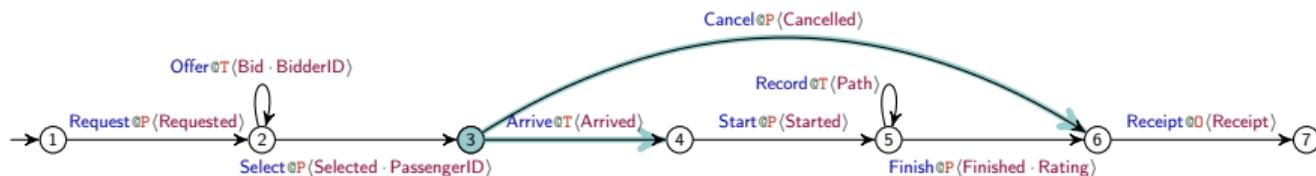
An example



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!

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

Semantics of swarm protocols

One rule only!

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

where

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

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

Semantics of swarm protocols

One rule only!

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

where

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

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

Semantics of swarm protocols

One rule only!

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

where

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

Logs to be consumed "atomically", hence $\delta(G, \ell)$ 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
- ▶ 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

$$\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{l}_i \rangle . G_i \right) \downarrow_{\mathbf{R}} = \kappa \cdot [\&_{i \in I} \mathbf{l}_i ? G_i \downarrow_{\mathbf{R}}]$$

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

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

$$\left(\sum_{i \in I} c_i @_{R_i} \langle \mathbf{l}_i \rangle \cdot G_i \right) \downarrow_{\mathbf{R}} = \kappa \cdot [\&_{i \in I} \mathbf{l}_i? G_i \downarrow_{\mathbf{R}}]$$

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

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

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

where

Another attempt



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

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

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

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

where

$$\kappa = \{c_i / \mathbf{1}_i \mid R_i = R \text{ and } i \in I\}$$

$$J = \{i \in I \mid \text{filter}(\mathbf{1}_i, \sigma(R)) \neq \epsilon\}$$

$$\text{filter}(\mathbf{1}, E) = \begin{cases} \epsilon, & \text{if } \mathbf{t} = \epsilon \\ \mathbf{t} \cdot \text{filter}(\mathbf{1}', E) & \text{if } \mathbf{t} \in E \text{ and } \mathbf{1} = \mathbf{t} \cdot \mathbf{1}' \\ \text{filter}(\mathbf{1}, E) & \text{otherwise} \end{cases}$$

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])

⇒ differences in how machines perceive the (state of the) computation

Causality

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

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

*If R should have a command enabled after c_i
then $\sigma(R)$ contains some event type emitted by c_i*

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

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 σ . For each branch $i \in I$ of $G = \sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i$

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

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

Determinacy $R \in \sigma G_i \implies \mathbf{1}_i[0] \in \sigma(R)$

Well-formedness = Causality + Determinacy - Confusion

Trading consistency for availability has implications:

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

⇒ 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 $G = \sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i$

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

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

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

Confusion freeness there is a unique subtree G' of G emitting t
for each t starting a log emitted by a command in G

```

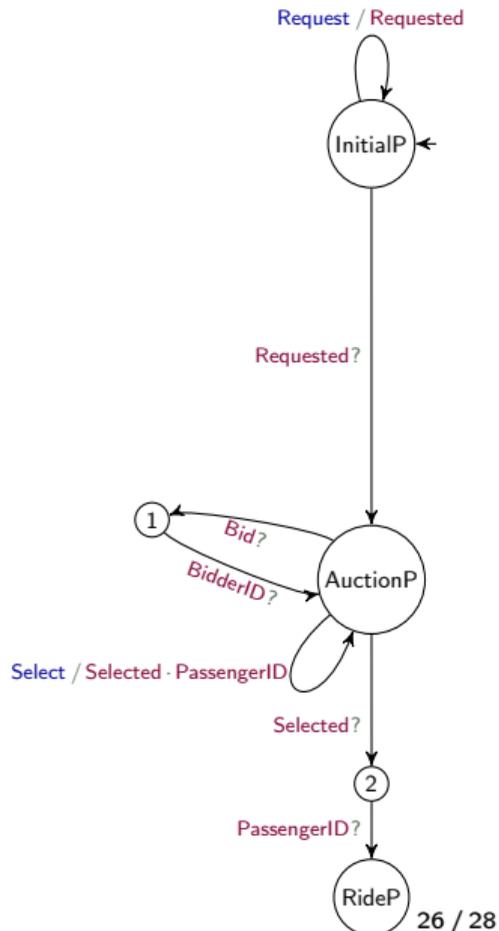
// analogous for other events; "type" property matches type name (checked by tool)
type Requested = { type: 'Requested'; pickup: string; dest: string }
type Events = Requested | Bid | BidderID | Selected | ...

/** 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 })
  }
  onSelect(ev: Selected, id: PassengerID) {
    return new RideP(this.id, ev.taxiID)
  }
}

@proto('taxiRide')
export class RideP extends State<Events> { ... }

```

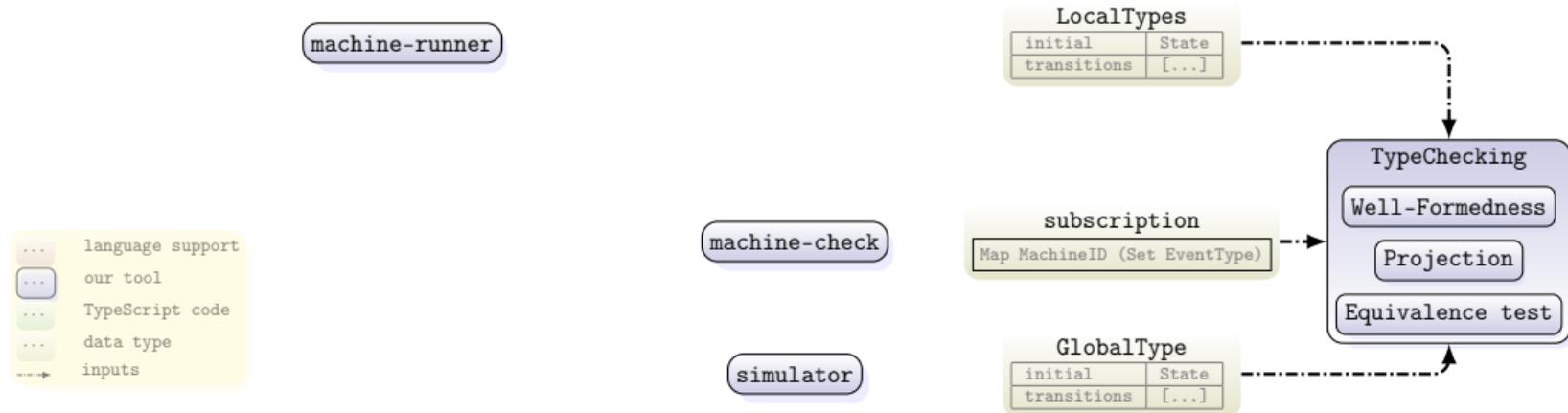


Architecture [2]



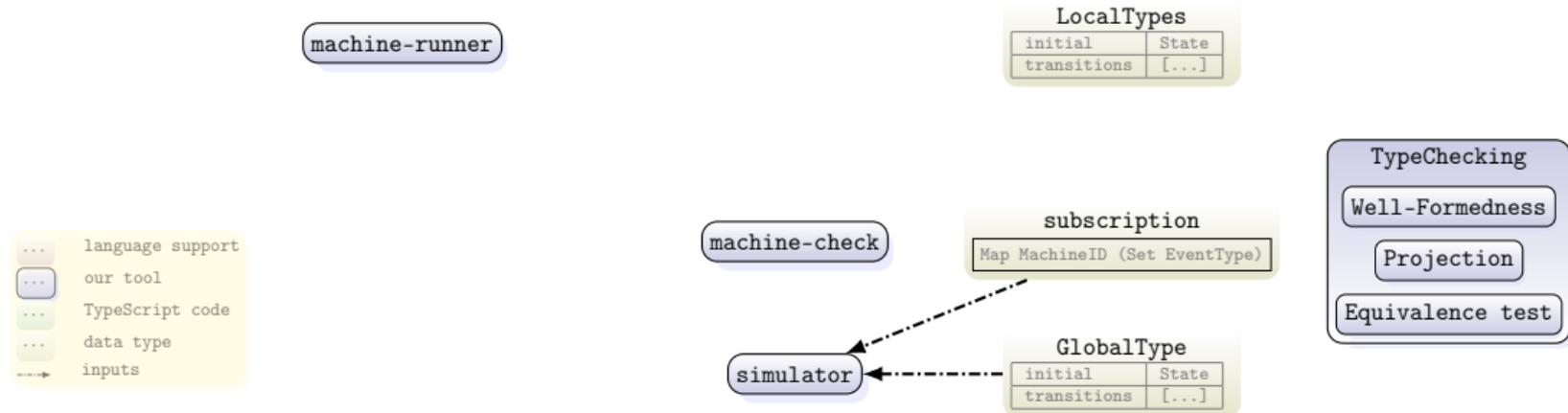
- ▶ `TypeChecking` implements the functionalities of our typing discipline
- ▶ `simulator` simulates the semantics of swarm realisations
- ▶ `machine-check` and `machine-runner` integrate our framework in the Actyx platform

Architecture [2]



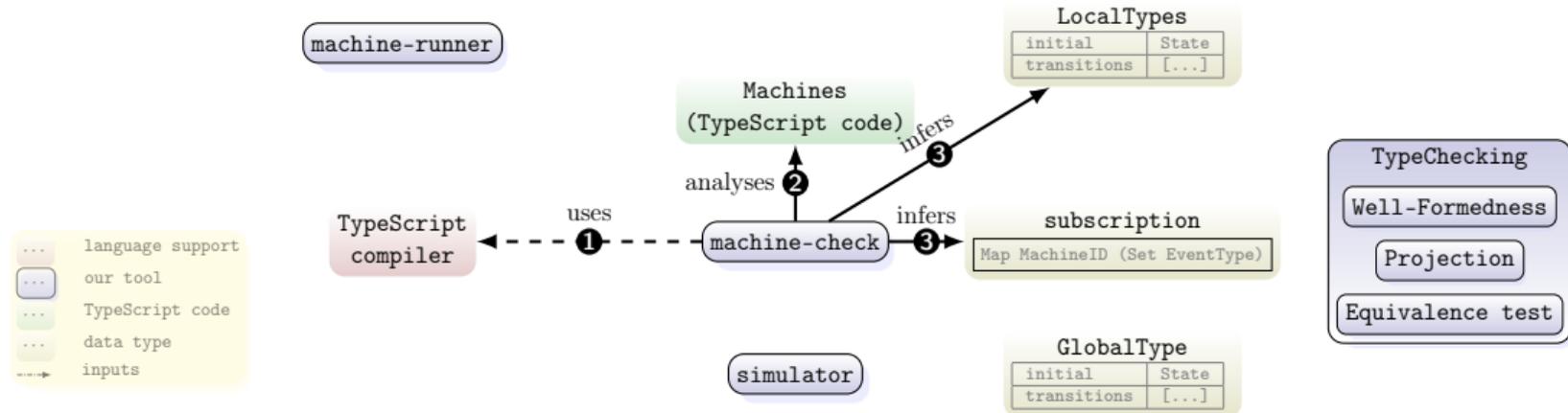
- ▶ TypeChecking implements the functionalities of our typing discipline
- ▶ simulator simulates the semantics of swarm realisations
- ▶ machine-check and machine-runner integrate our framework in the Actyx platform

Architecture [2]



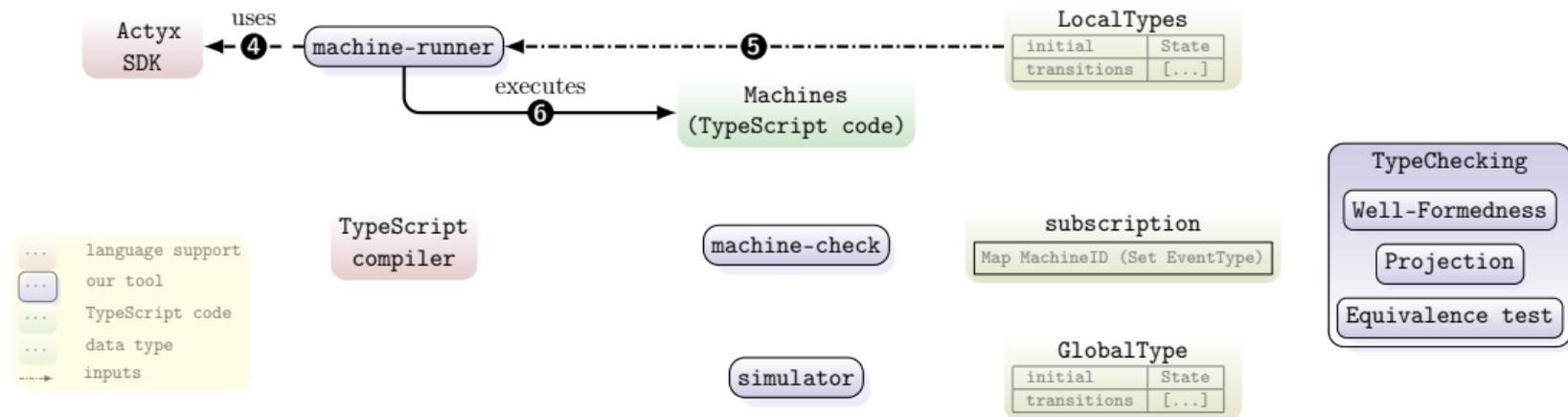
- ▶ TypeChecking implements the functionalities of our typing discipline
- ▶ simulator simulates the semantics of swarm realisations
- ▶ machine-check and machine-runner integrate our framework in the Actyx platform

Architecture [2]



- ▶ 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 [2]



- ▶ 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

- [1] 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.
- [2] 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.