Formal Methods for Communication Protocols

Harnessing distributed software design with behavioural contracts

Emilio Tuosto @ GSSI

– Lecture 1 –

3 - 12 March, 2026 - Novi Sad

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani
DESK Developing Shared Knowledge

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

– Motivations –

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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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]

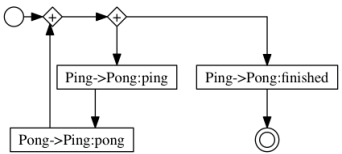
CFSMs: FIFO buffers as well

Friendlier representations with (formal) models

Local behaviour: communicating machines [2]

CFSMs: FIFO buffers as well

Choregraphy: global graph [3, 7]



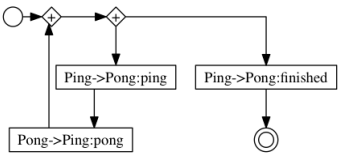
...“synchronous” distributed workflow

Friendlier representations with (formal) models

Local behaviour: communicating machines [2]

CFSMs: FIFO buffers as well

Choregraphy: global graph [3, 7]



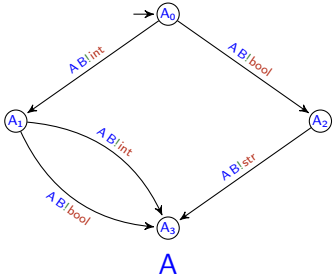
...“synchronous” distributed workflow

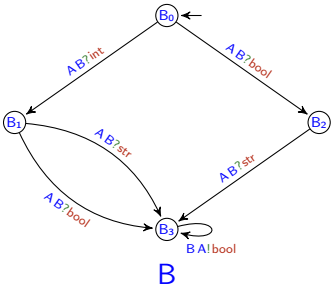
Research direction

How do we extract such models from code or documentation?

Our first formal model...informally

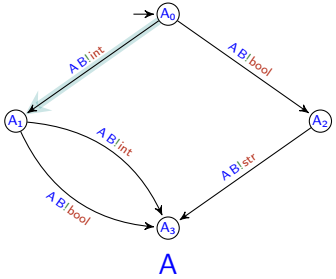
Communicating systems

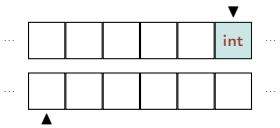


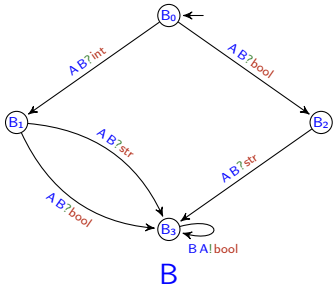


Our first formal model...informally

Communicating systems

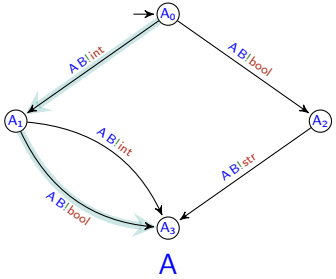


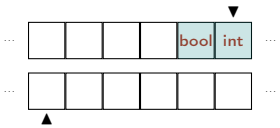


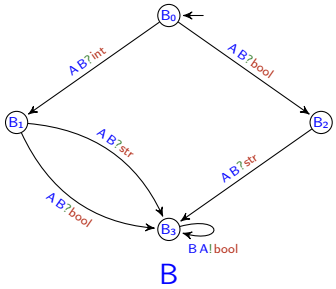


Our first formal model...informally

Communicating systems

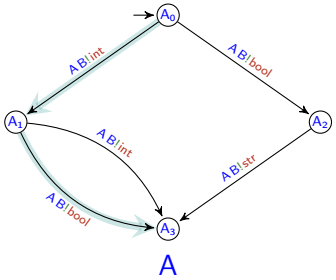


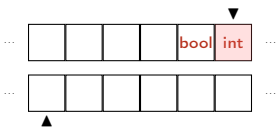


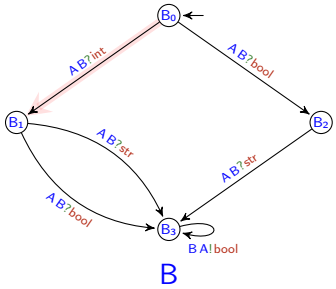


Our first formal model...informally

Communicating systems

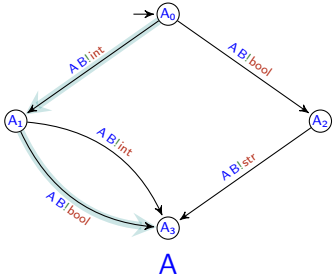


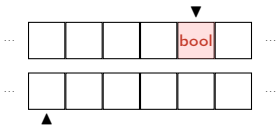


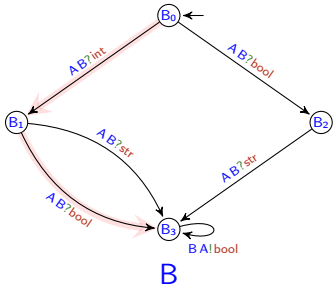


Our first formal model...informally

Communicating systems

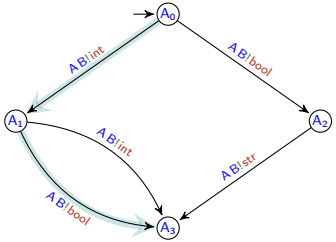


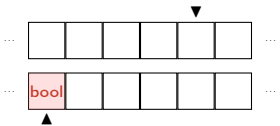


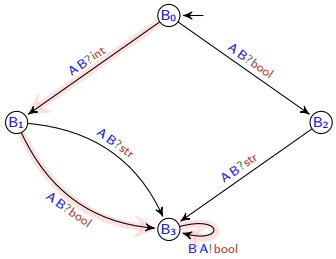


Our first formal model...informally

Communicating systems

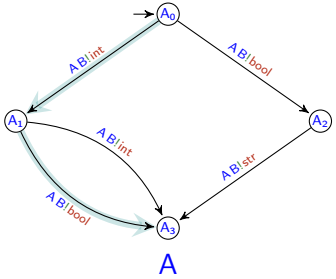


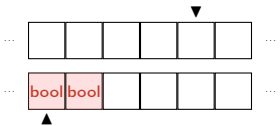


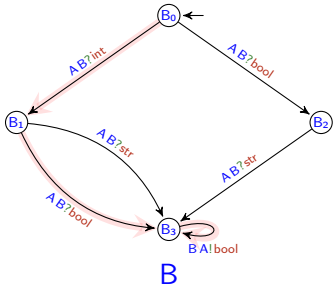


Our first formal model...informally

Communicating systems

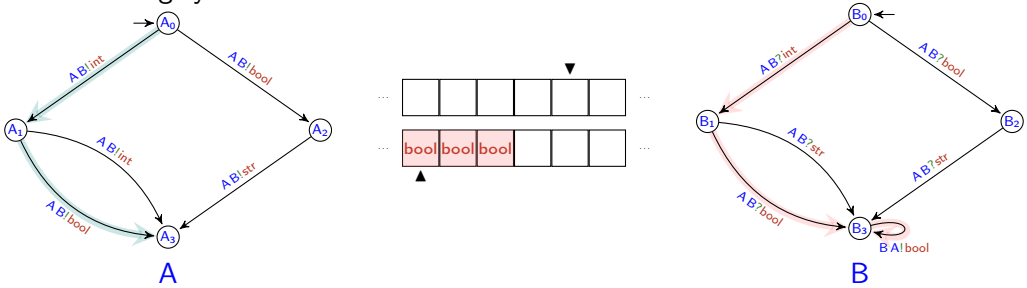






Our first formal model...informally

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?

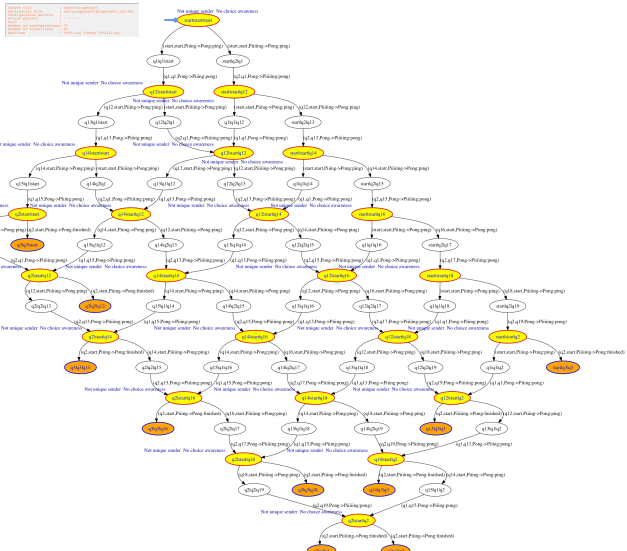
A:

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

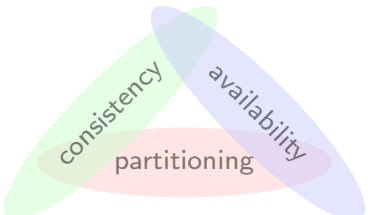
What does 'software' actually mean?

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?

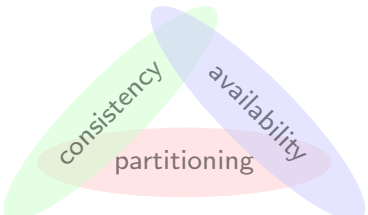
FM theory in practice



The CAP theorem

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

FM theory in practice



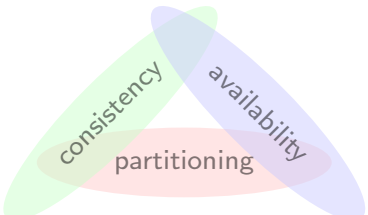
The CAP theorem

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

With some "solutions"

- Centralisation points (not that appealing)

FM theory in practice



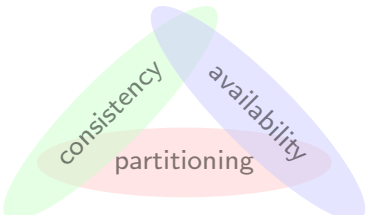
The CAP theorem

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



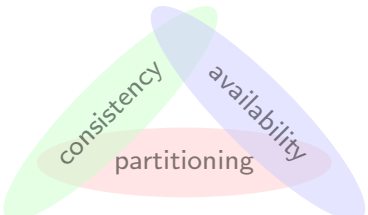
The CAP theorem

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



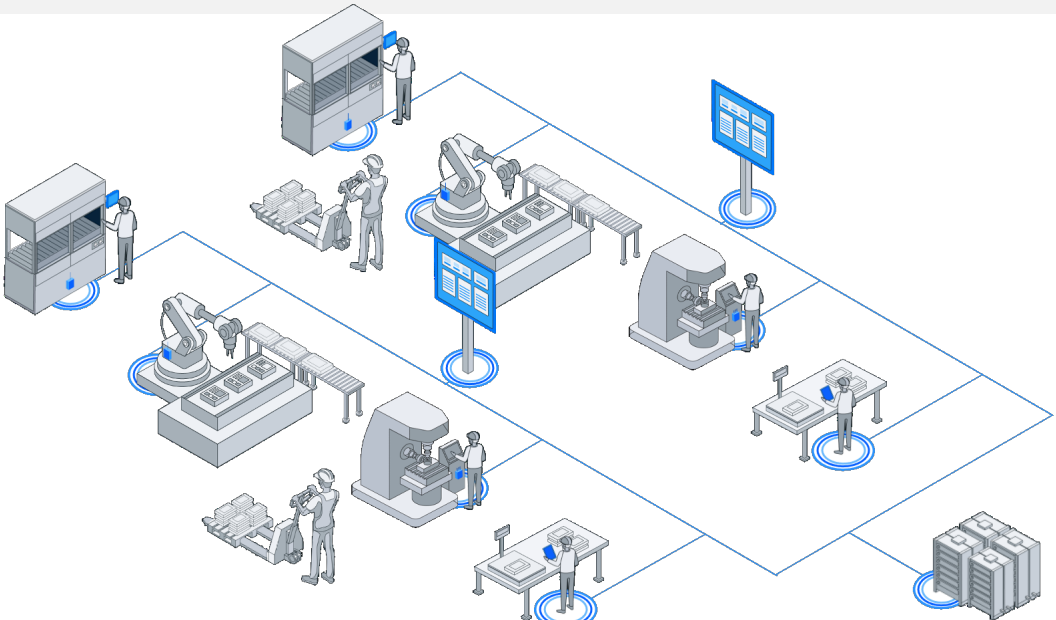
The CAP theorem

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)
- ...

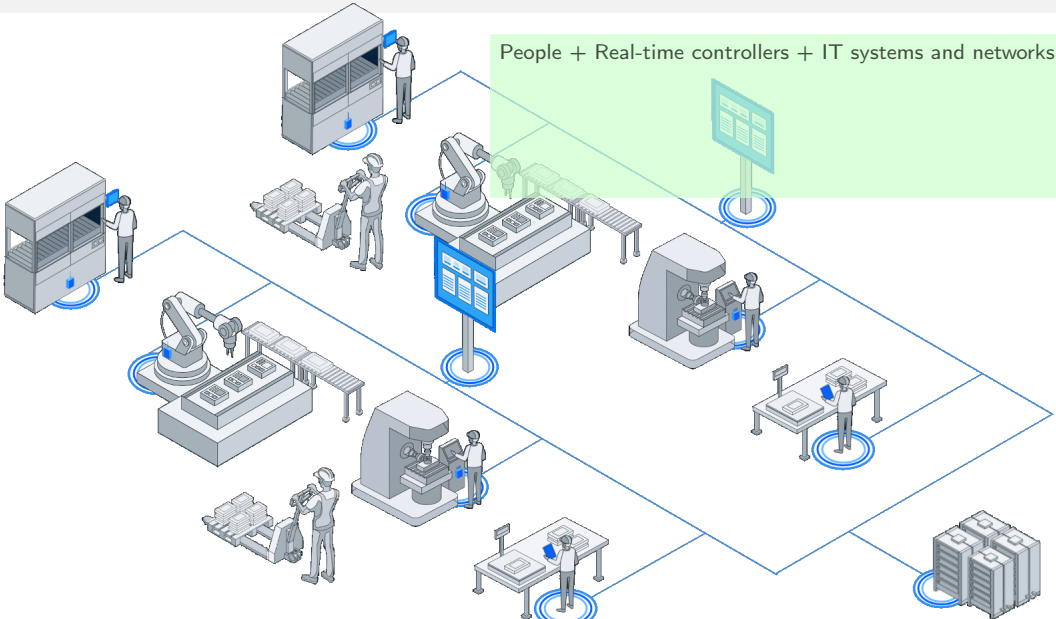
An execution model for a collaborative environment



(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

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

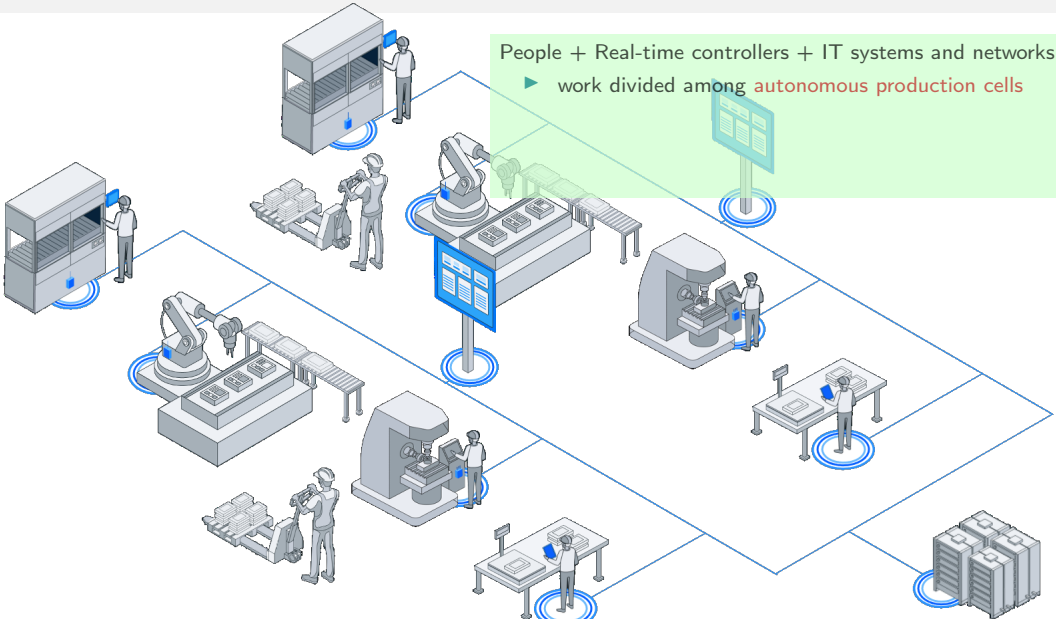


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

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

- work divided among autonomous production cells

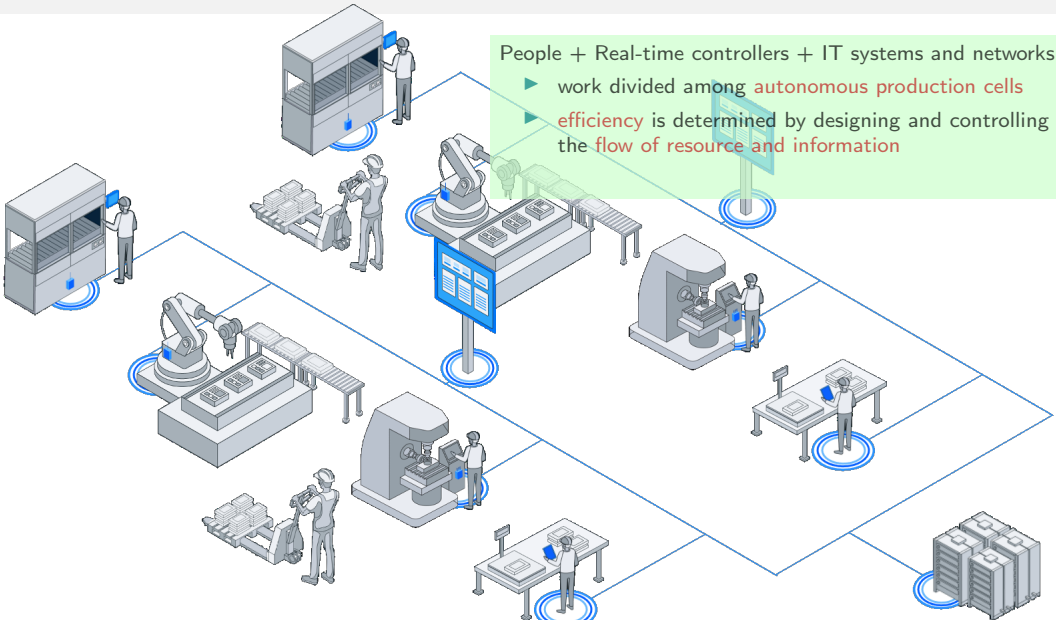


(the pictures are courtesy of Actyx AG)

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

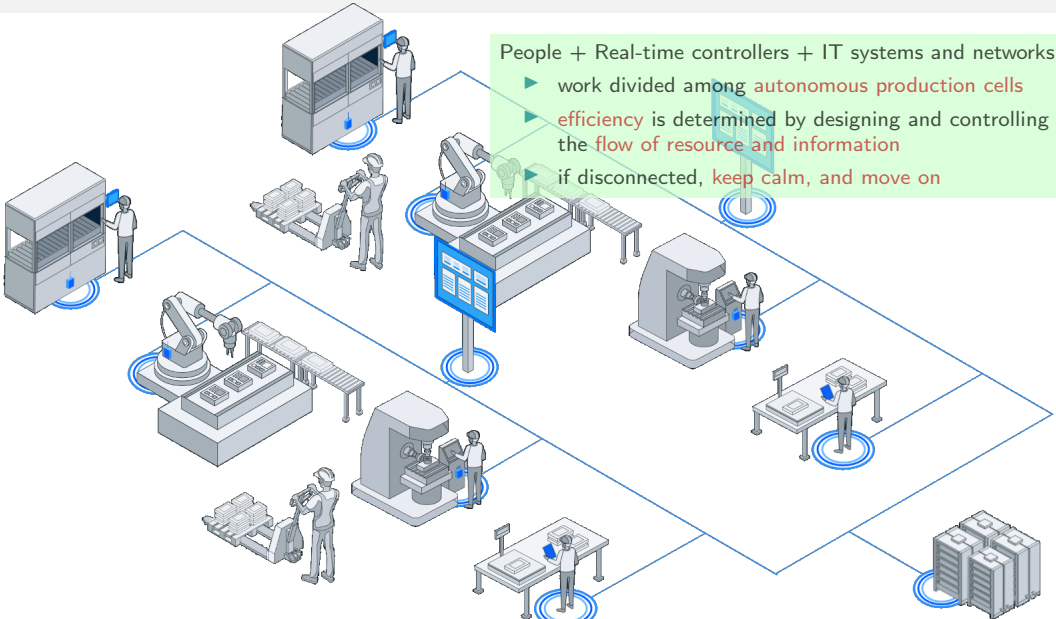


(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

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

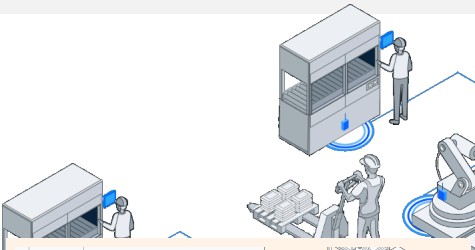
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on



(the pictures are courtesy of Actyx AG)

An execution model for a collaborative environment

(the pictures are courtesy of Actyx AG)



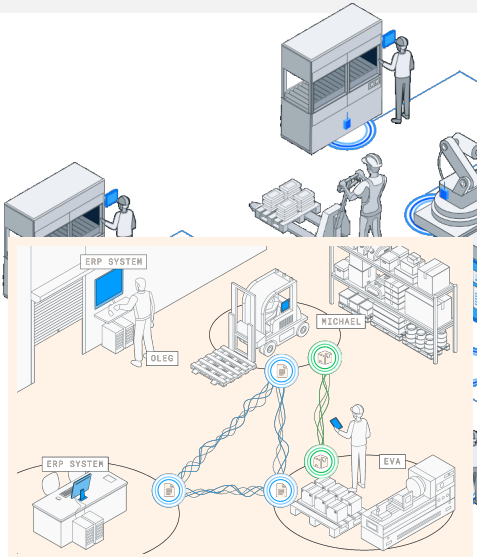
People + Real-time controllers + IT systems and networks:

- 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:

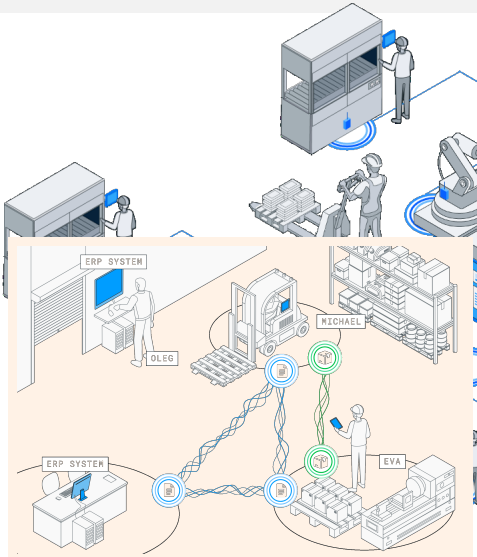
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

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:

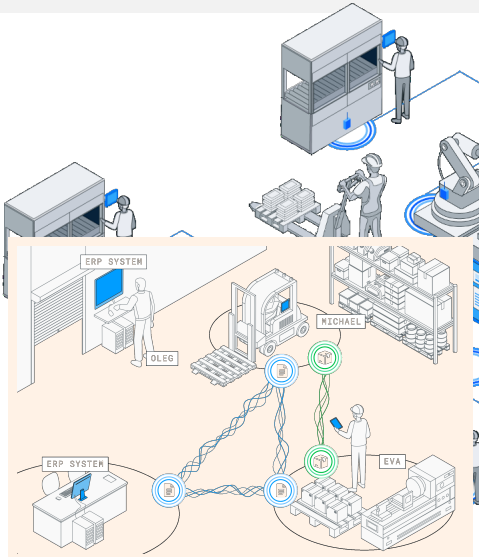
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

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:

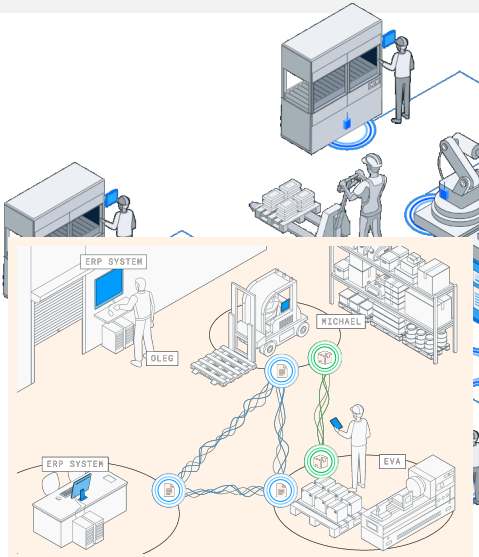
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

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:

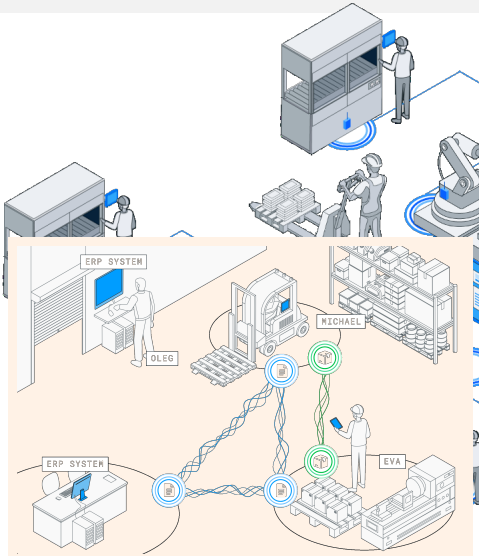
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

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:

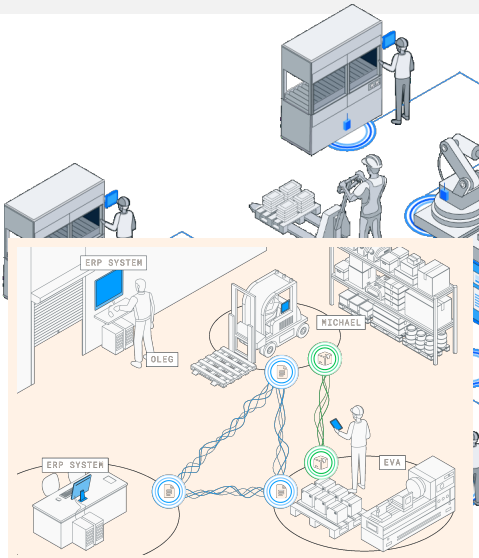
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

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

A motto

design (well)

+

project

+

run

A motto

design (well)

+

project

+

run

Another motto

execute

+

propagate

+

merge

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] 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.

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

[6] 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.

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

[8] 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 [4]. available at `http://www.cs.le.ac.uk/people/et52/jlamp-with-proofs.pdf`.