”Think global, act local!”
or
the other way around?

A (gentle?) introduction to distributed choreographies

Emilio Tuosto
University of Leicester

MGS@B’ham, 11 - 15 April 2016

April 15, 2016

Distribution
(prelude)

Motivations

Distributed applications are (becoming) ubiquitous...

BBC Sign in | News | Sport | Weather | iPlayer | TV | Radio

NEWS BUSINESS

Home | World | UK | England | N. Ireland | Scotland | Wales | Business | Politics | Health | Education | Sci/Environ

Market Data | Your Money | Economy | Companies

11 January 2012 Last updated at 06:12

Share

Google persuades Spanish bank BBVA to use the cloud

By Tim Weber
Business editor, BBC News website

Spanish banking giant BBVA is switching its 110,000 staff to use Google's range of enterprise software.

The deal is the biggest that the search giant has signed with one company for its cloud-computing services, where software is offered as a service via the internet.

The bank told the BBC it would use Google's tools only for internal communication.

BBVA will keep Google applications separate from customer data

Microsoft Research

Search Microsoft Research

Our research | Connections | Careers | About us

All | Downloads | Events | Groups | News | People | Projects | Publications | Videos

Singularity: Rethinking the Software Stack

Galen C. Hunt and James R. Larus
April 2007

Abstract

Every operating system embodies a collection of design decisions. Many of the decisions behind today's most popular operating systems have remained unchanged, even as hardware and software have evolved. Operating systems form the foundation of almost every software stack, so inadequacies in present systems have a pervasive impact. This paper describes the efforts of the Singularity project to re-examine these design choices in light of advances in programming languages and verification tools. Singularity systems incorporate three key architectural features: software-isolated processes for protection of programs and system services, contract-based channels for communication, and manifest-based programs for verification of system properties. We describe this foundation in detail and sketch the ongoing research in experimental systems that build upon it.

Publication files

osr2007_rethinkingsoftw
df
bibtex.bib

Related people

- Galen Hunt
- Jim Larus

Motivations

... but designing, developing, maintaining, and verifying distributed applications is a tough job

theguardian

News | Sport | Comment | Culture | Business | Money | Life & sty

News › Society › NHS

Abandoned NHS IT system has cost £10bn so far

Bill for abortive plan, described as 'the biggest IT failure ever seen', was originally estimated to be £6.4bn

Rajeev Syal
The Guardian, Wednesday 18 September 2013

Jump to comments (1153)

Implementation

The failure of the CAD system which came to live operation from 26 October 1992 was as a result of cumulative consequences of associated problems (identified above) that joined together to produce a chain of decline in its performance. The absence of near-perfect information upon which the system relied to allocate the required resource to an incident was a key factor to this decline [24]. The problems experienced during the implementation/live operation are summarized below: [23] [24]

- Incomplete software.
- Inability of the CAD software to identify and allocate the nearest available resource.
- The AVLS not being able to identify all the ambulances in the fleet.
- Communication problems among the CAD system, AVLS and Mobile data system.
- Slow operation of the system.
- Locking up of workstations.
- Inaccurate status reporting by ambulance crew when wrong buttons were pressed.
- Use of different vehicle by the crew from the one assigned by the system.

(cf. “London Ambulance Service Software Failure”, Adamu et al.)

Motivations

Why are such applications hard?

Because they

- dynamically (dis)appears
- dynamically combine
- dynamically form “ensembles”
- dynamically reconfigure
- dynamically engage in “conversations”
- and are often developed independently
- ...

Motivations

Why are such applications hard?
Because they

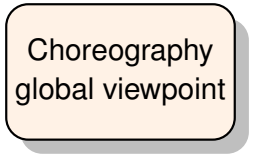
- dynamically (dis)appears
- dynamically combine
- dynamically form “ensembles”
- dynamically reconfigure
- dynamically engage in “conversations”
- and are often developed independently
- ...

Choreography-based approaches (act I)

Quoting W3C...

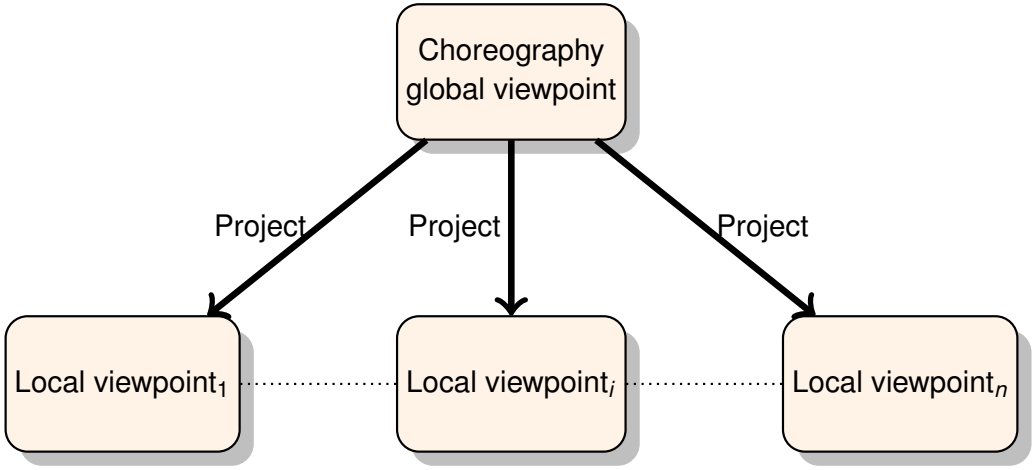
"Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour of all the parties involved. Each party can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

A “top-down” approach



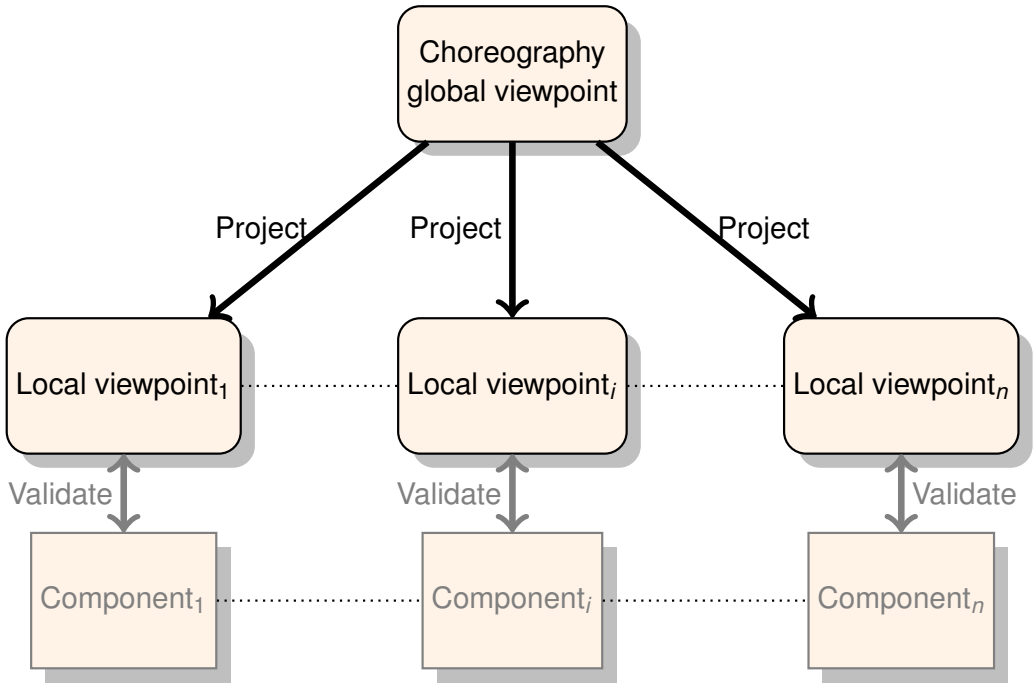
- The process can be iterated
- google for Testable Architectures

A “top-down” approach



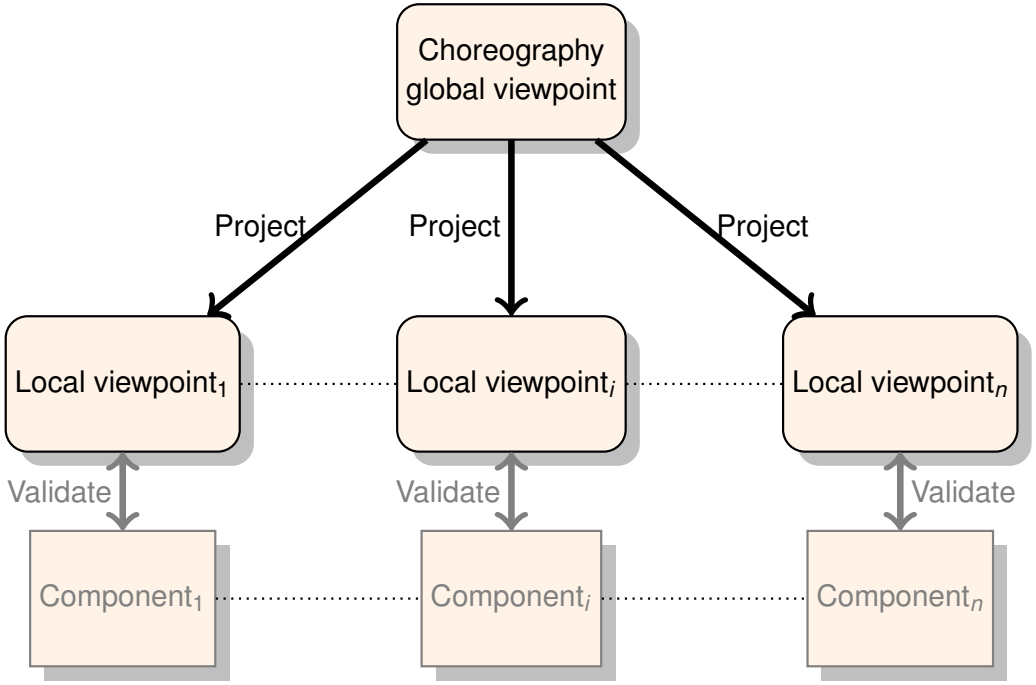
- The process can be iterated
- google for Testable Architectures

A “top-down” approach



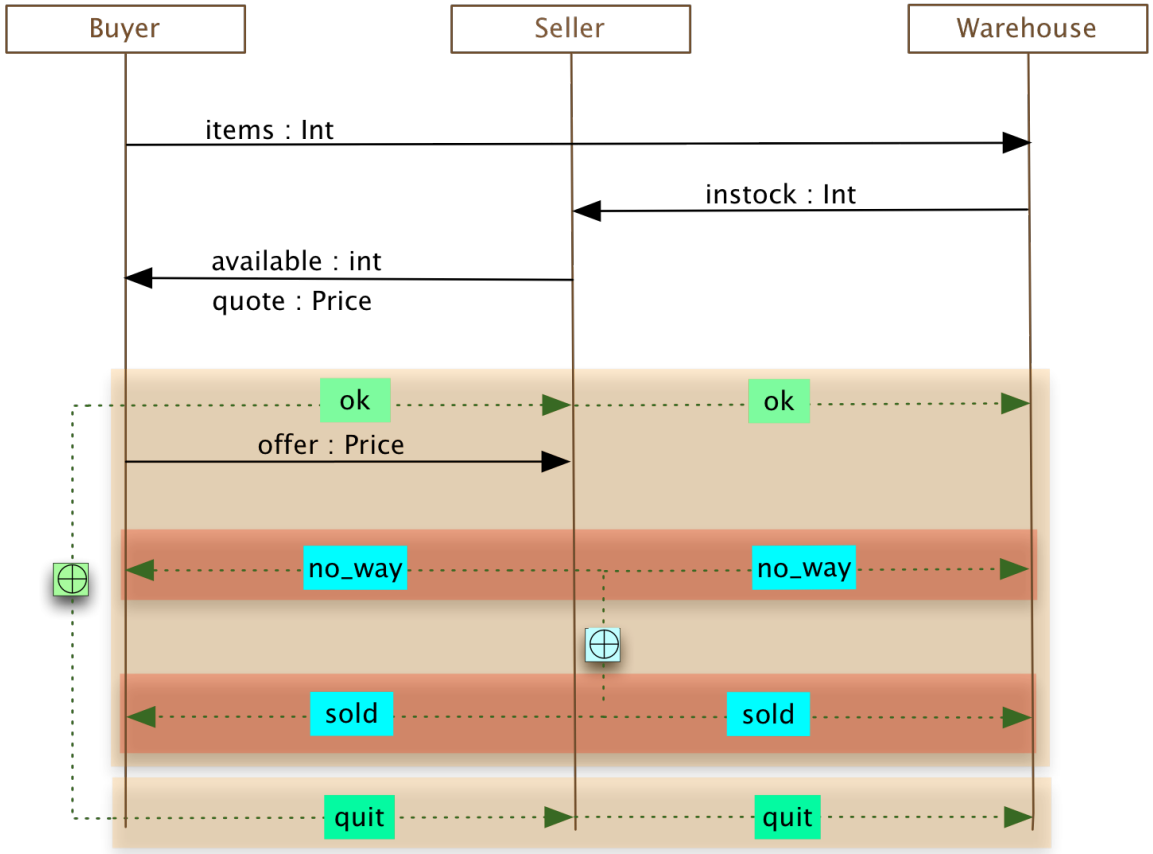
- The process can be iterated
- google for Testable Architectures

A “top-down” approach



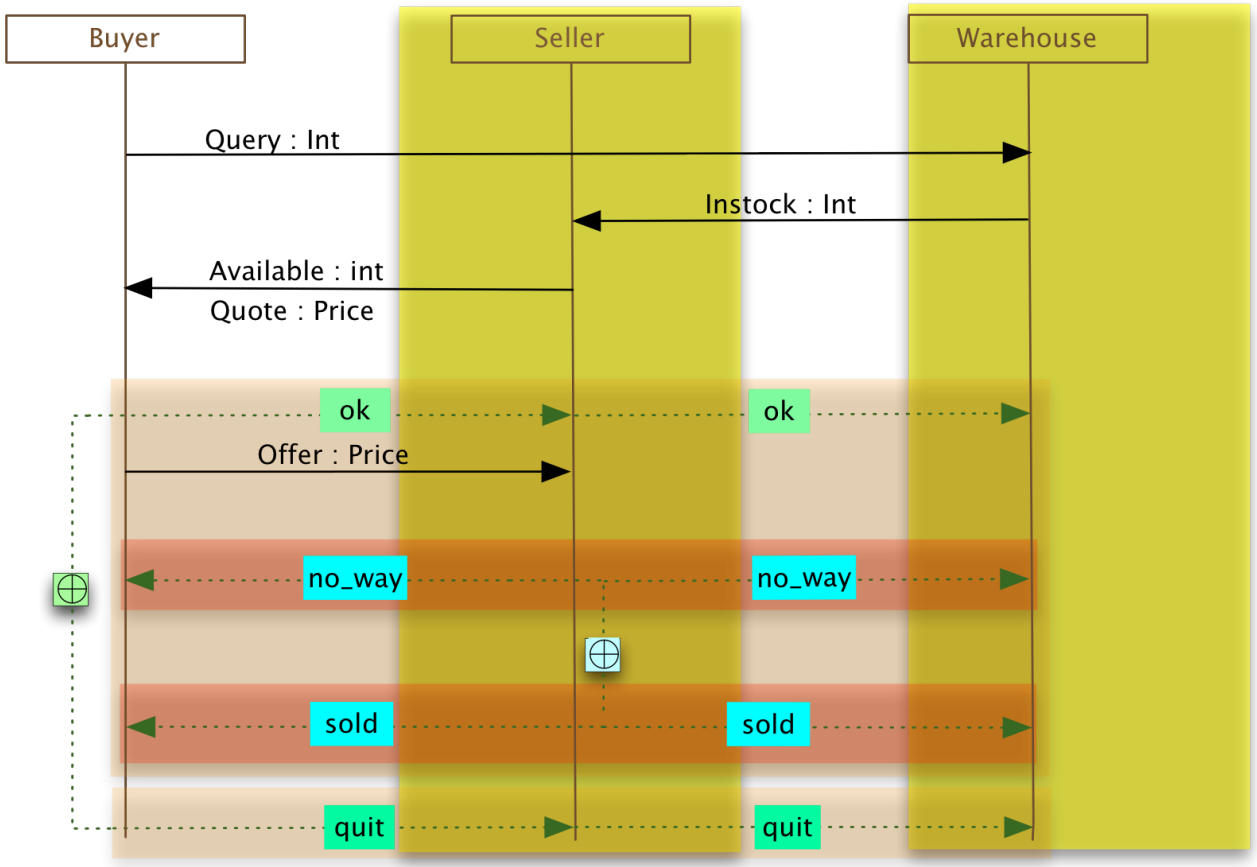
- The process can be iterated
- google for Testable Architectures

An intuitive account...



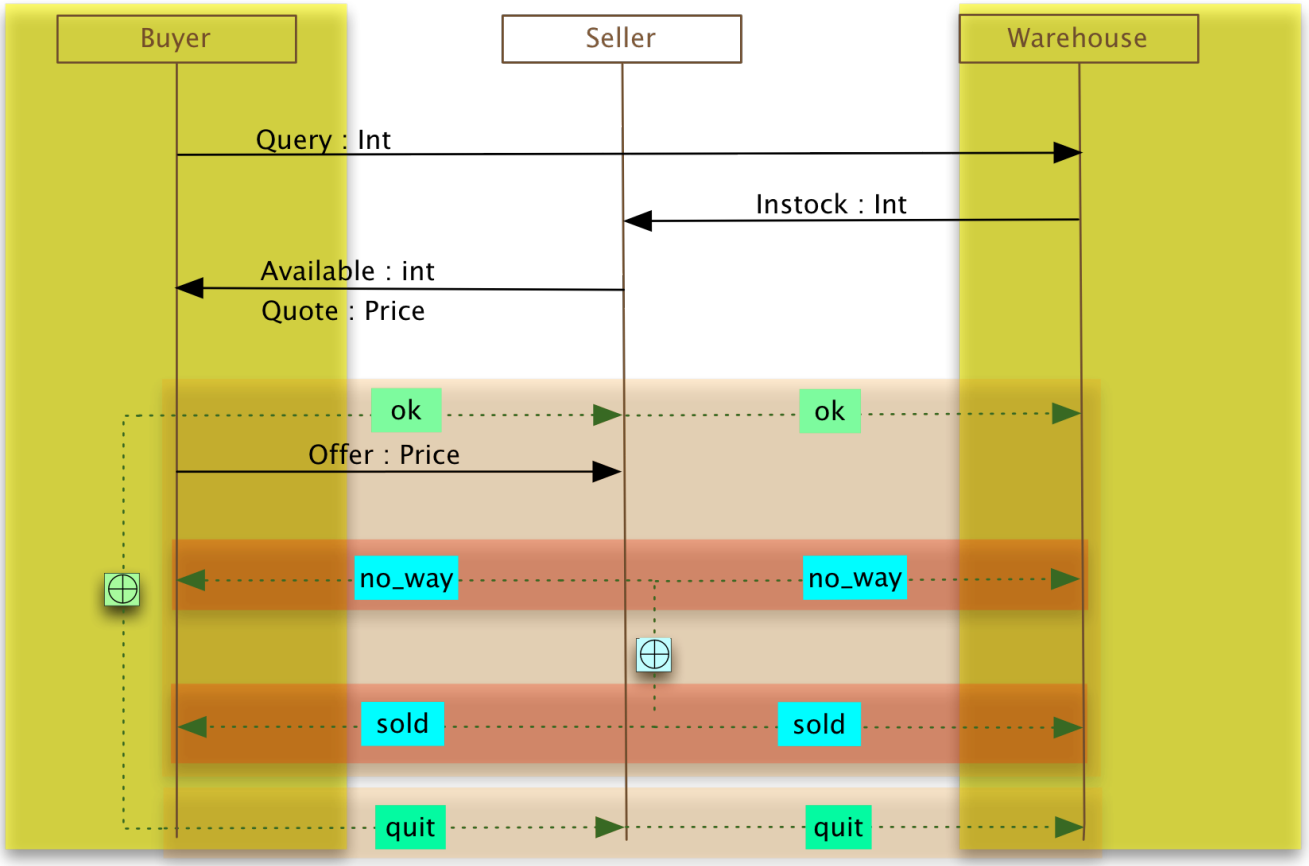
Global viewpoint

An intuitive account...



Projecting on buyer

An intuitive account...



Projecting on seller

Some considerations

Things are more complex:

- recursion/iteration
- not all global viewpoints “make sense”
  (e.g., constraints on values passing)
- interactions are “atomic” at global level, but not at local level
- ...

Desiderata

- progress (graceful termination or no-deadlock)
- no orphan messages
- no unspecified reception
- ...

Quest for precision

To check properties of choreographies, we need more precision

- Type based approaches:
  global and local points of view = behavioural types

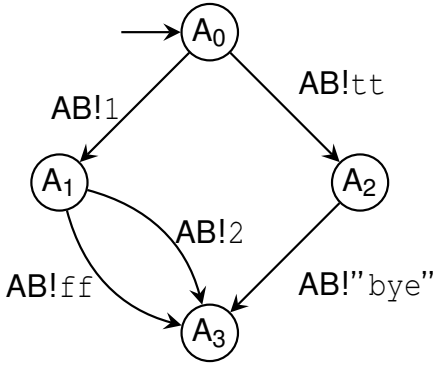
  > *"[...] no conflict (racing) at session channels. To ensure this, the necessary and sufficient condition is that, when a common channel is used in two communications, their sending actions and their receiving actions should respectively be ordered temporally"*
  >
  > *[Honda, Yoshida, Carbone: POPL 2008]*

  This yields a typing-principle, hence

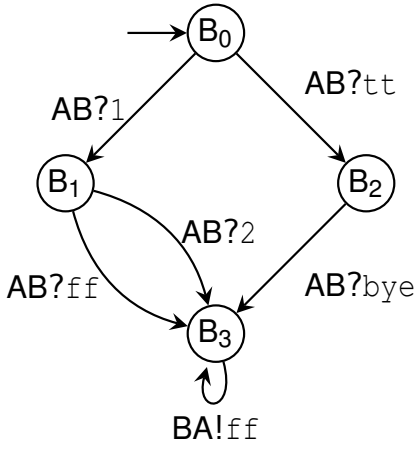
  safety = type checking

Distributed execution (intermezzo)

Communicating finite state machines



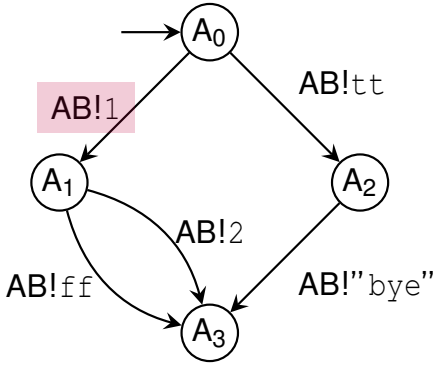
A



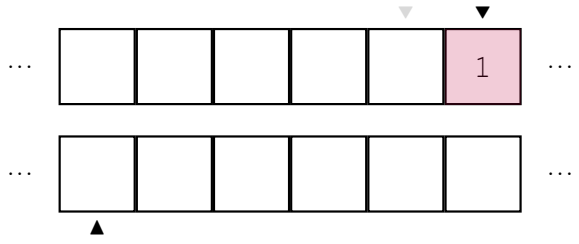
B

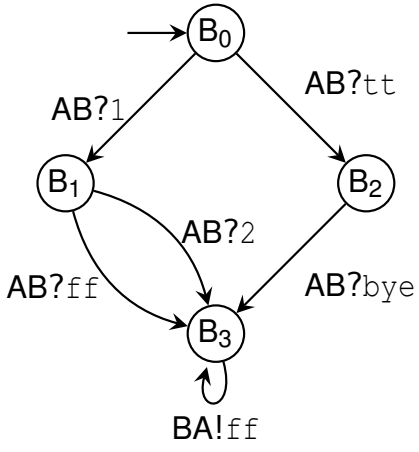
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

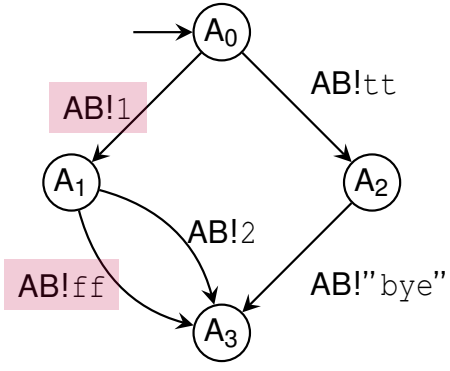




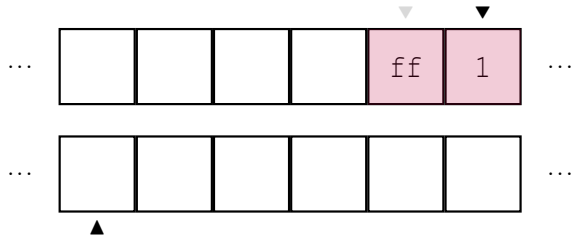
B

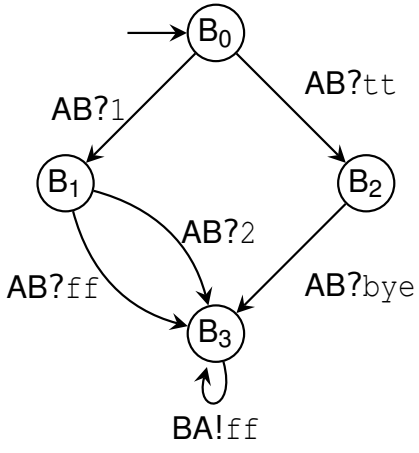
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

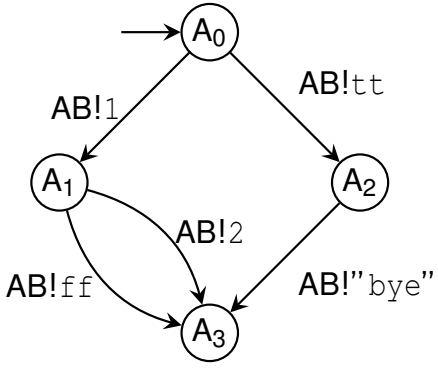




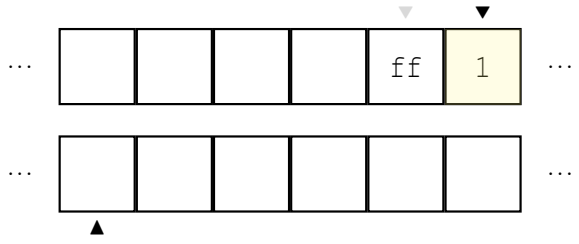
B

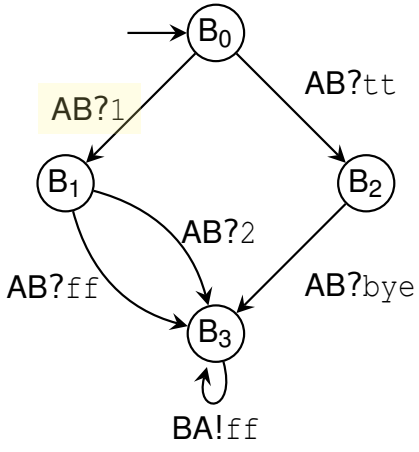
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

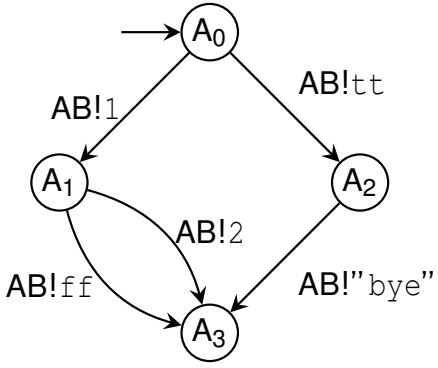




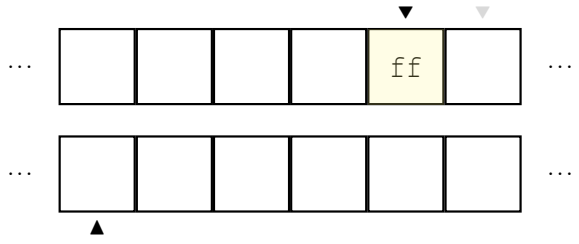
B

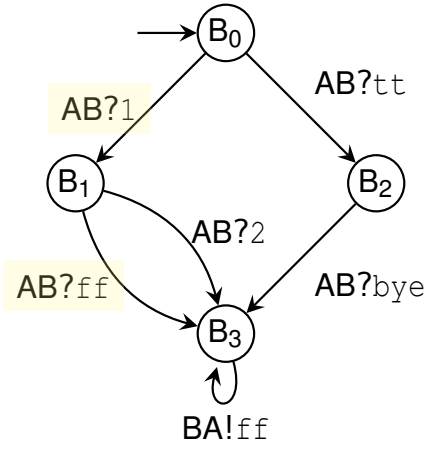
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

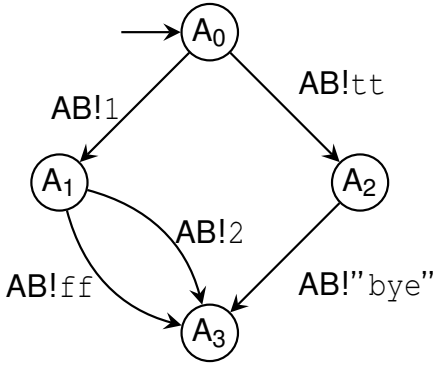




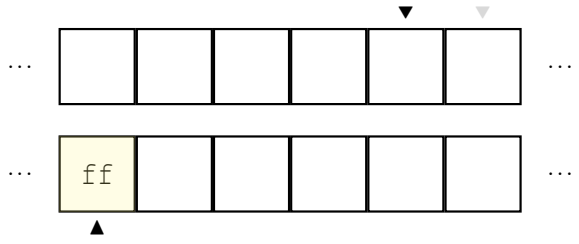
B

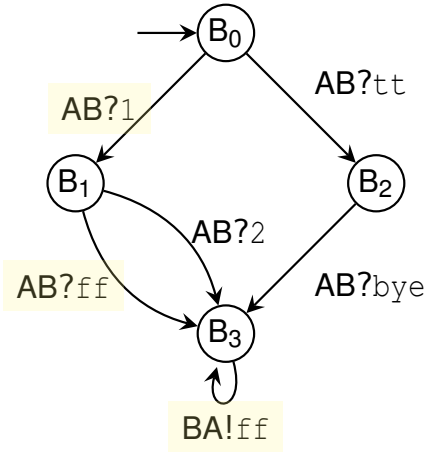
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

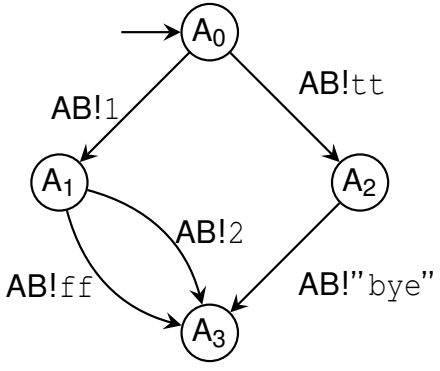




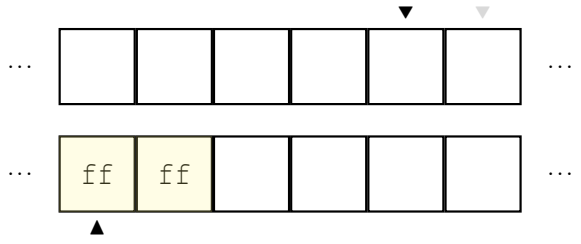
B

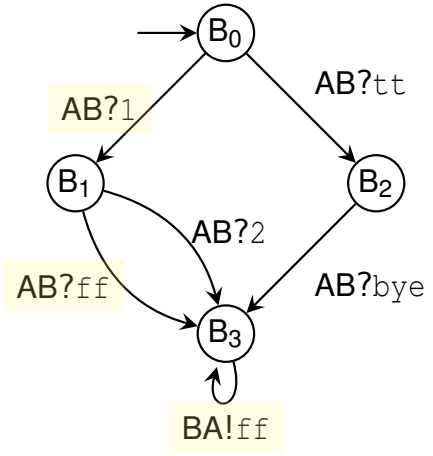
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

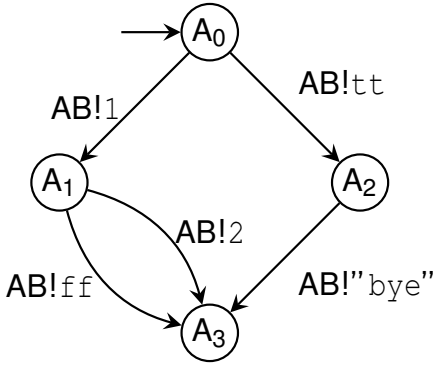




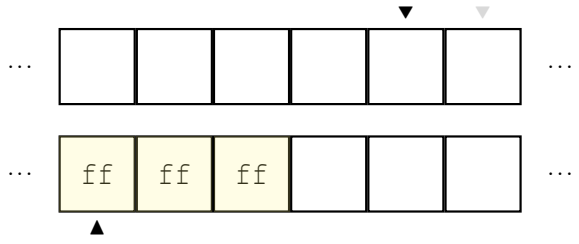
B

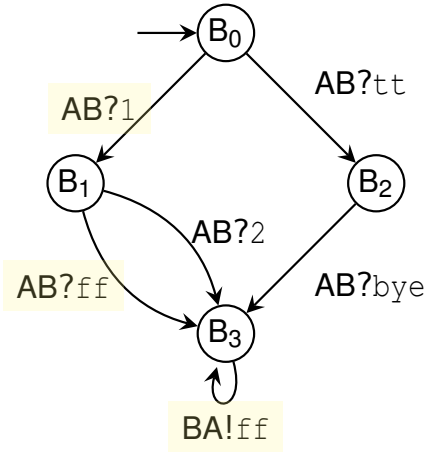
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A





B

cf. Brand & Zafiropulo 1983

Choreographies with data (act II)

Choreographies as types + data

A flavour of a behavioural type:

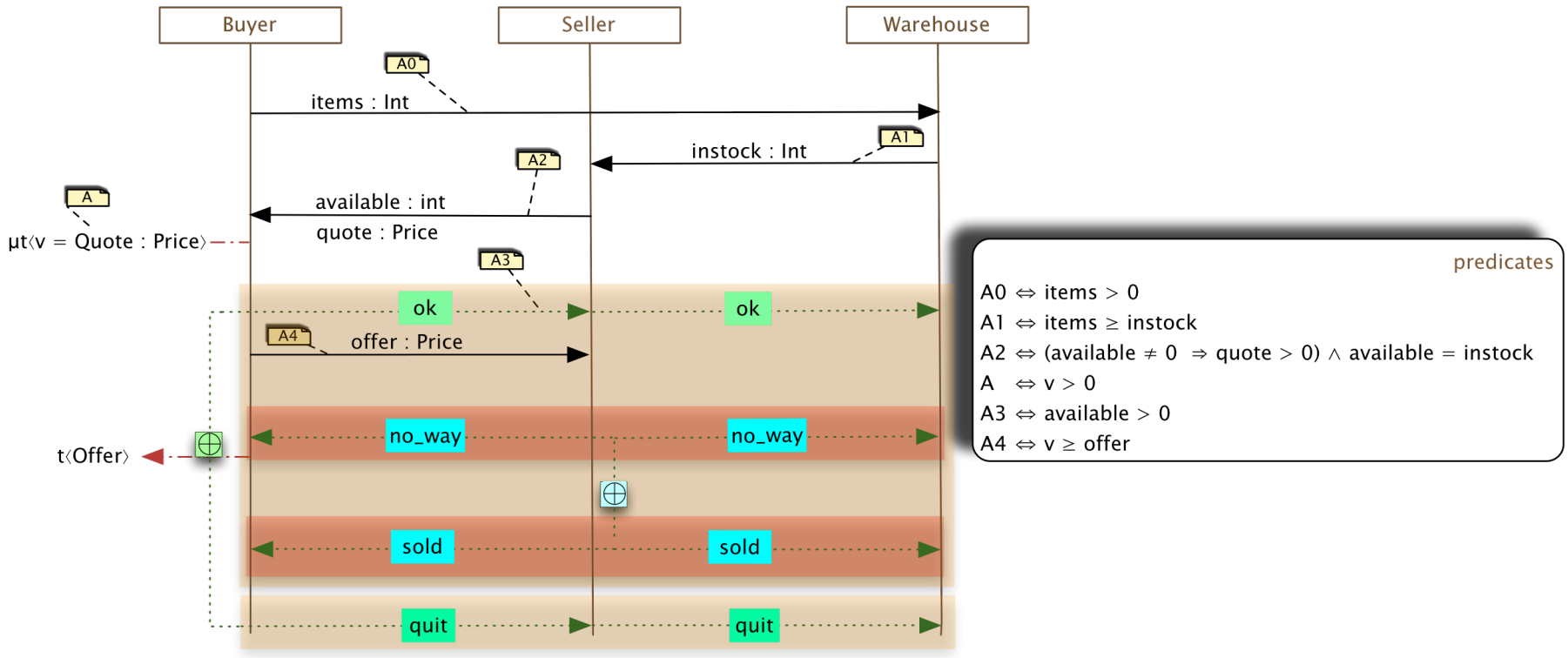
$$
\begin{array}{rcl}
\mathcal{G} & ::= & \mathsf{p} \to \mathsf{q}\colon (x : S)\{A\}.\mathcal{G} \\
& \mid & \mathsf{p} \to \mathsf{q}\colon \left(\{A_j\}\mathsf{l}_j : \mathcal{G}_j \right)_{j \in J} \\
& \mid & \mu\, \mathsf{t}\, (x = e : S)\{A\}.\, \mathcal{G} \\
& \mid & \mathsf{t}\langle e \rangle \\
& \mid & \mathsf{end}
\end{array}
$$

where

- A and A_j are predicates expressing conditions on variables x
- e is an expression denoting values of 'normal' data types S (eg integer, lists, bool, etc)
- note that recursions come with an initialization of recursion parameters $x = e$

[Honda, Bocchi, Tuosto, Yoshida: CONCUR 2010]

Expressiveness of choreographies with data (intuitively)



Data yield troubles

Besides some technical issues (relatively easy to sort out), data make the previous typing principle not enough anymore.

Exercise

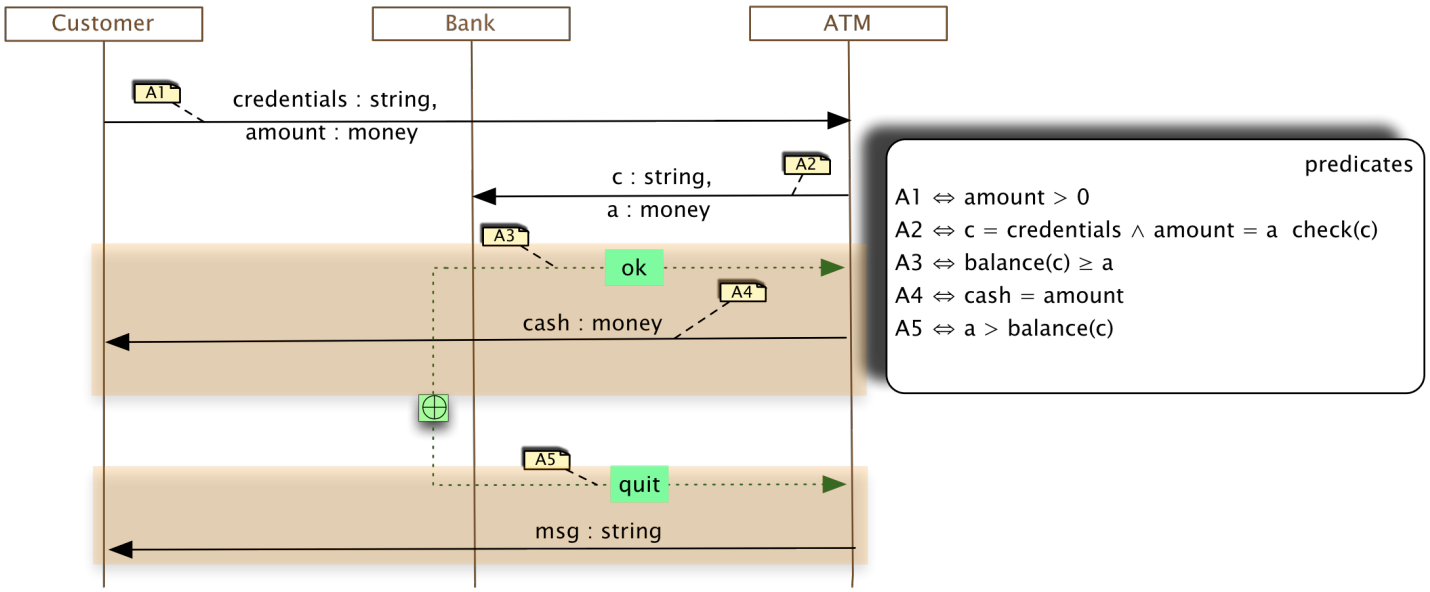
Design an ATM choreography that allows customers to withdraw money.

Data yield troubles

Besides some technical issues (relatively easy to sort out), data make the previous typing principle not enough anymore.

Exercise

Design an ATM choreography that allows customers to withdraw money.

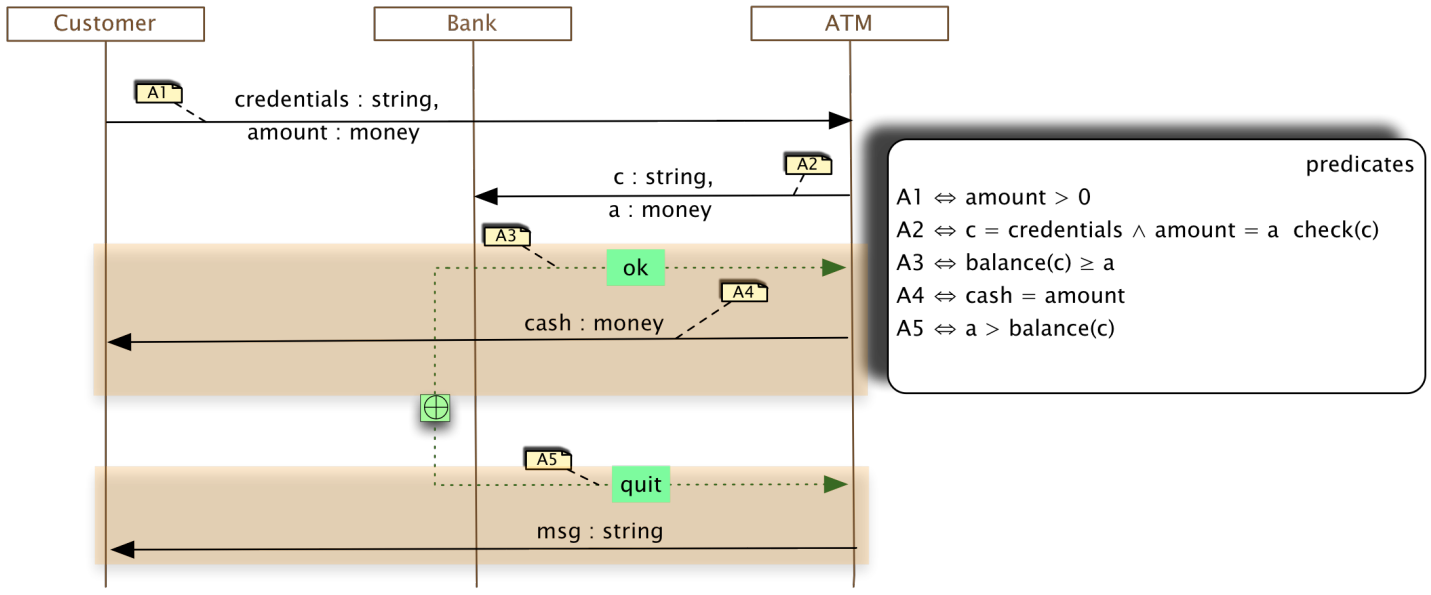


Data yield troubles

Besides some technical issues (relatively easy to sort out), data make the previous typing principle not enough anymore.

Exercise

Design an ATM choreography that allows customers to withdraw money.



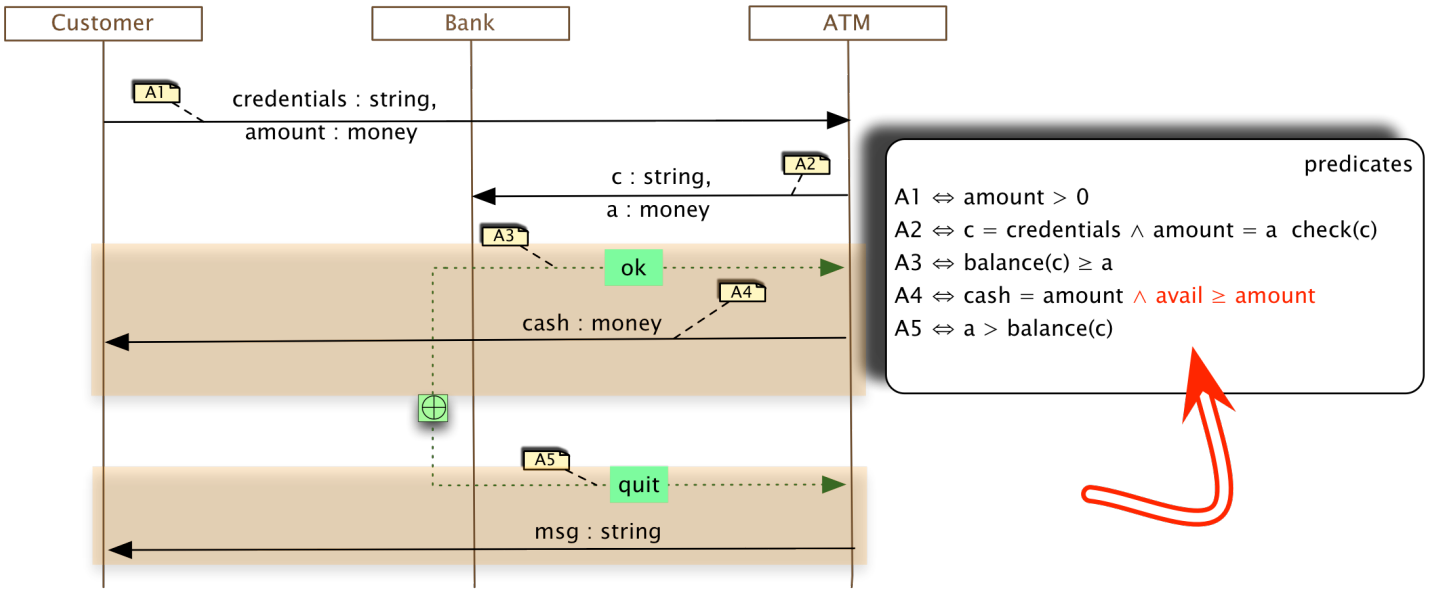
...what if the ATM hasn't enough cash?

Data yield troubles

Besides some technical issues (relatively easy to sort out), data make the previous typing principle not enough anymore.

Exercise

Design an ATM choreography that allows customers to withdraw money.

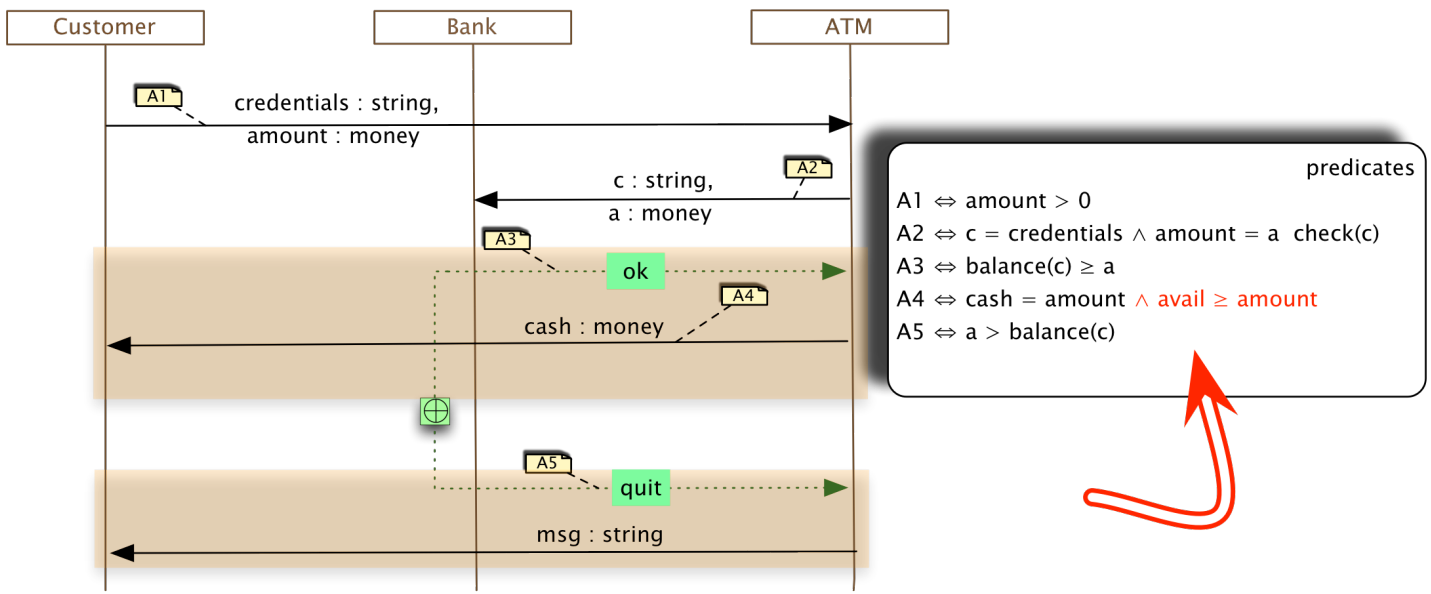


Data yield troubles

Besides some technical issues (relatively easy to sort out), data make the previous typing principle not enough anymore.

Exercise

Design an ATM choreography that allows customers to withdraw money.



easy to “fix”, but...

[Bocchi, Lange, Tuosto: Sci. Ann. Comp. Sci. 22(1) 2012 & ICE2011]

From designs to programs

A key problem is: how do we resolve non-determinism in actual implementations?

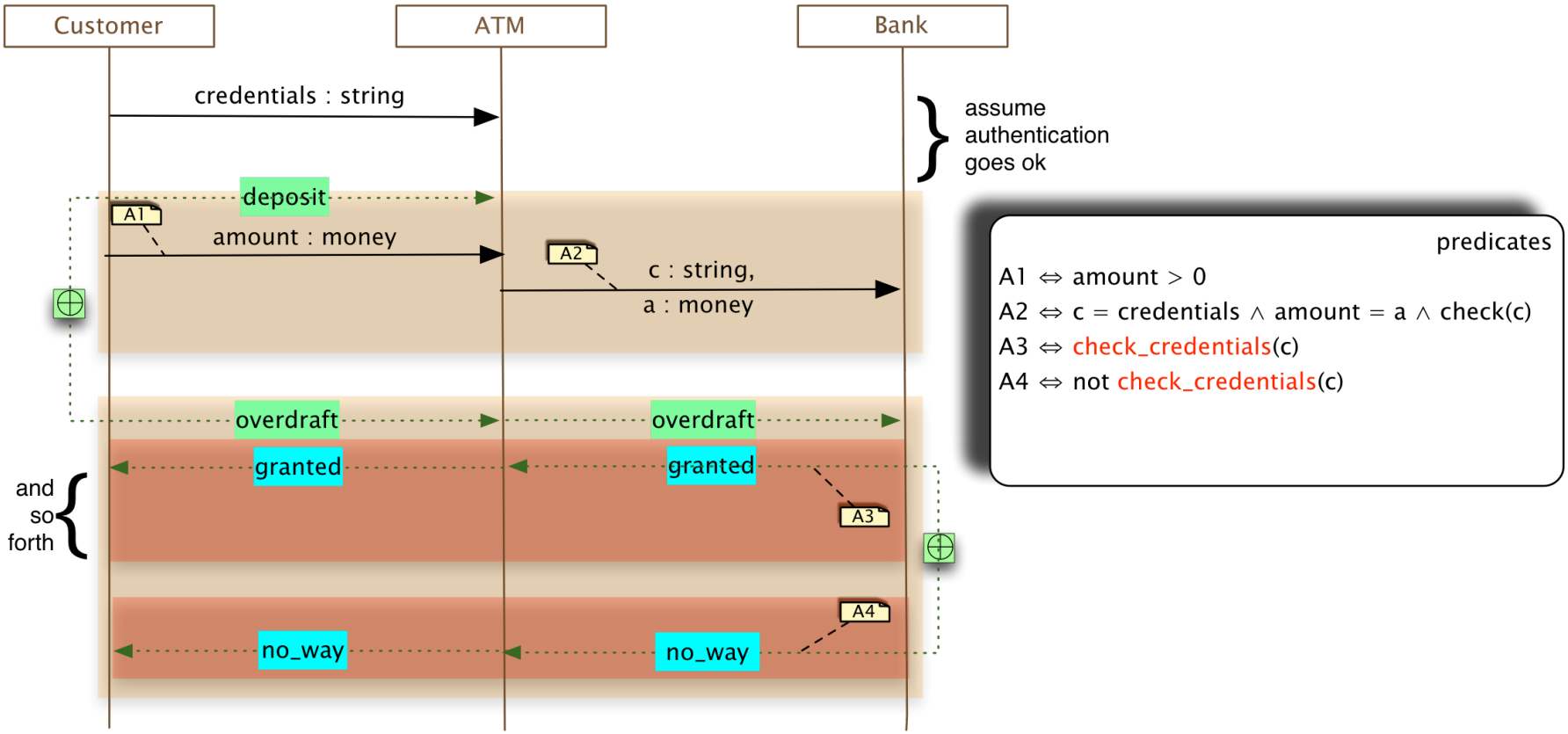
Choreographies (or more generally non-deterministic designs) can be interpreted either

as constraints: in all executions, at least one of the non-deterministic branches has to be executed

as obligations: in all executions, each branch has to occur at least once.

Depending on the applications, the former could be a too weak requirement while the latter a too strong one.

WSI by example



You would like to reject an implementation where check_credentials(c) returns false for all c!

[Bocchi, Melgratti, Tuosto: ESOP 2014]

Choreographies uʍop-əpᴉsdn (finale)

A glimpse of Erlang

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

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


pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  end.


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

Semantics

- Message passing
- FIFO buffers [[mailboxes in Erlang's jargon]]
- Spawn of threads

Asynchrony by design

Erlang is an incarnation of the well-known actor model of Hewitt and Agha...dates back to '73!

A glimpse of Erlang

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

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

```
pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  end.
```

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

Semantics

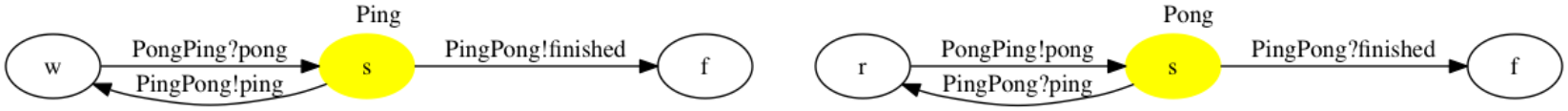
- Message passing
- FIFO buffers [[mailboxes in Erlang's jargon]]
- Spawn of threads

Asynchrony by design

Erlang is an incarnation of the well-known actor model of Hewitt and Agha...dates back to '73!

Friendlier representations

Local behaviour: communicating machines



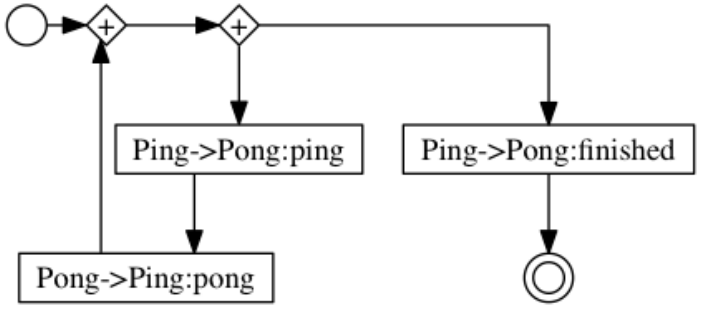
CFSMs (Brand & Zafiropulo 1983!): FIFO buffers as well

ChoSyn

...this is also amenable to tool supported analysis...:

`https://bitbucket.org/emlio_tuosto/gmc-synthesis-v0.2`

Choregraphy: global graph



..."synchronous" distributed workflow (Deniélou and Yoshida 2012)

A glimpse of Erlang

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

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

```
pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  end.
```

```
start() ->
  Pong_PID = spawn(example, pong, []),
  spawn(example, ping, [3, Pong_PID]),
  spawn(example, ping, [2, Pong_PID]).
```

Q:

Is there anyone familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

find the bug in 15 seconds, if you know Erlang

Not versed in Erlang? No worries if you don't see it

A glimpse of Erlang

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

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

```
pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  end.
```

```
start() ->
  Pong_PID = spawn(example, pong, []),
  spawn(example, ping, [3, Pong_PID]),
  spawn(example, ping, [2, Pong_PID]).
```

Q:

Is there anyone familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

find the bug in 15 seconds, if you know Erlang

Not versed in Erlang? No worries if you don't see it

A glimpse of Erlang

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

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

```
pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  end.
```

```
start() ->
  Pong_PID = spawn(example, pong, []),
  spawn(example, ping, [3, Pong_PID]),
  spawn(example, ping, [2, Pong_PID]).
```

Q:

Is there anyone familiar with Erlang?

Q:

Is this program correct?

A:

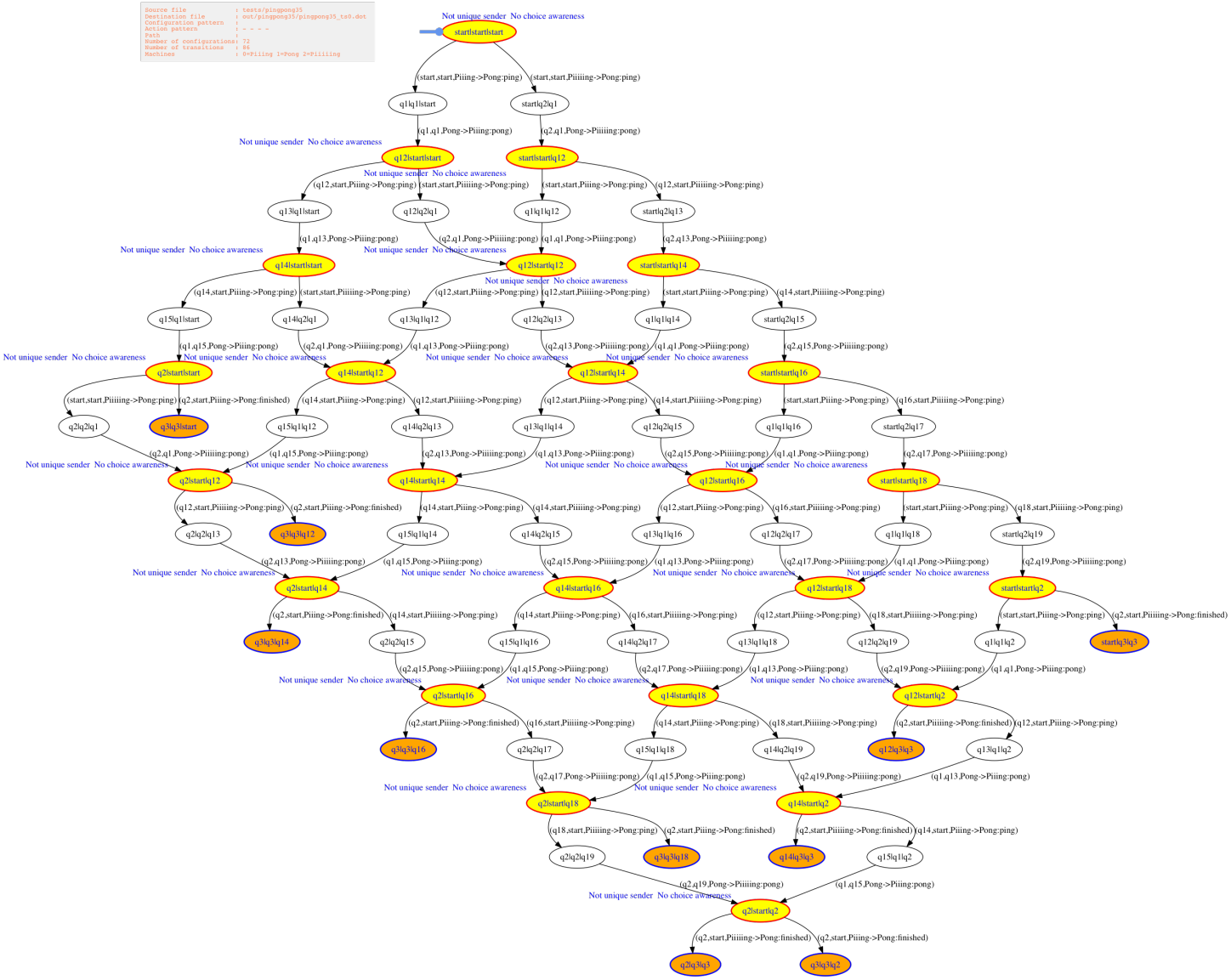
No!

Exercise:

find the bug in 15 seconds, if you know Erlang

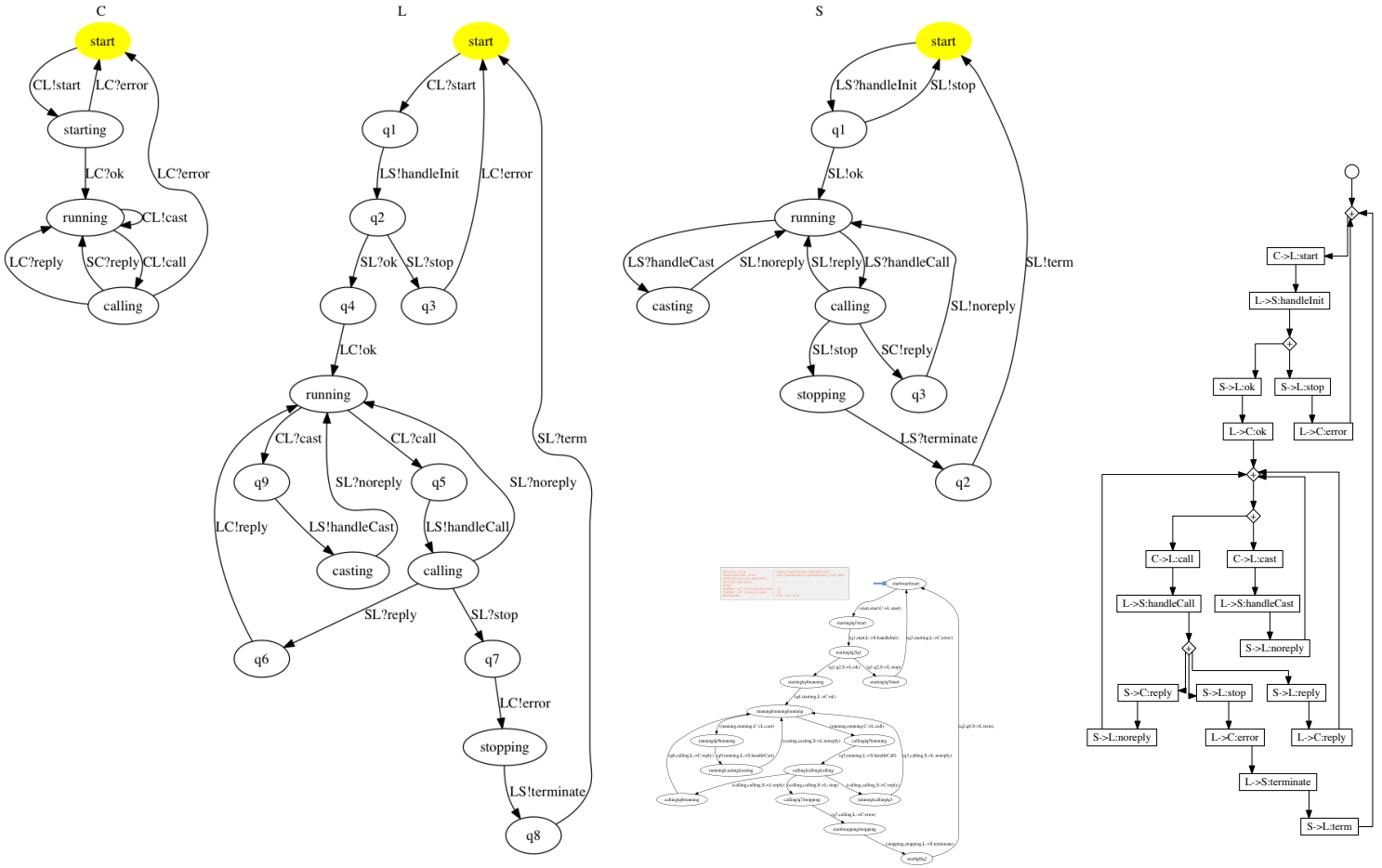
Not versed in Erlang? No worries if you don't see it

Send ping-pong to shell !!! ... I mean, use ChoSyn

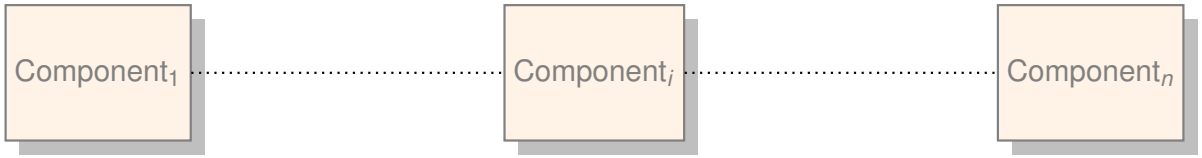


Amended gen_server models

Machines, TS, & Global Graph

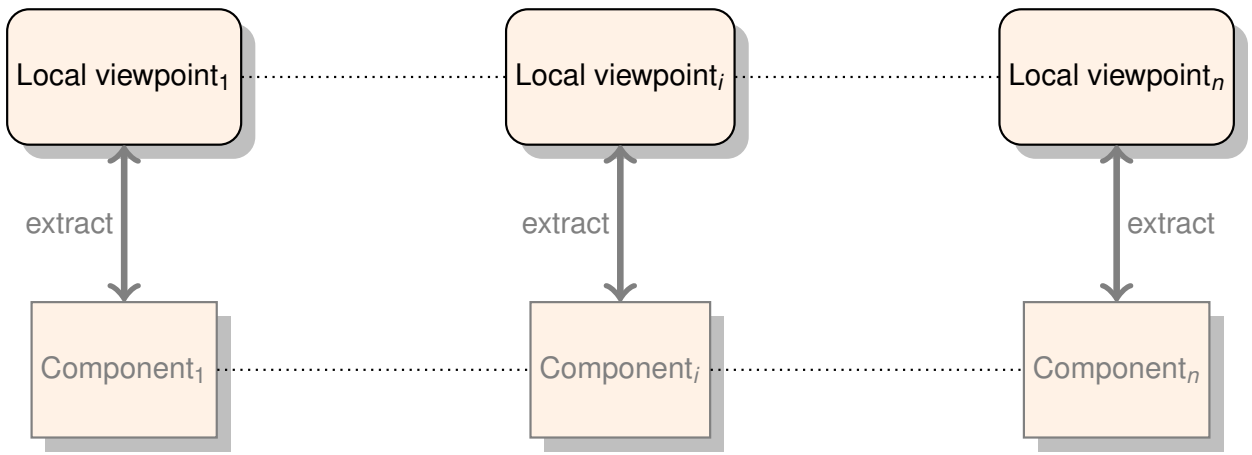


From programs to designs



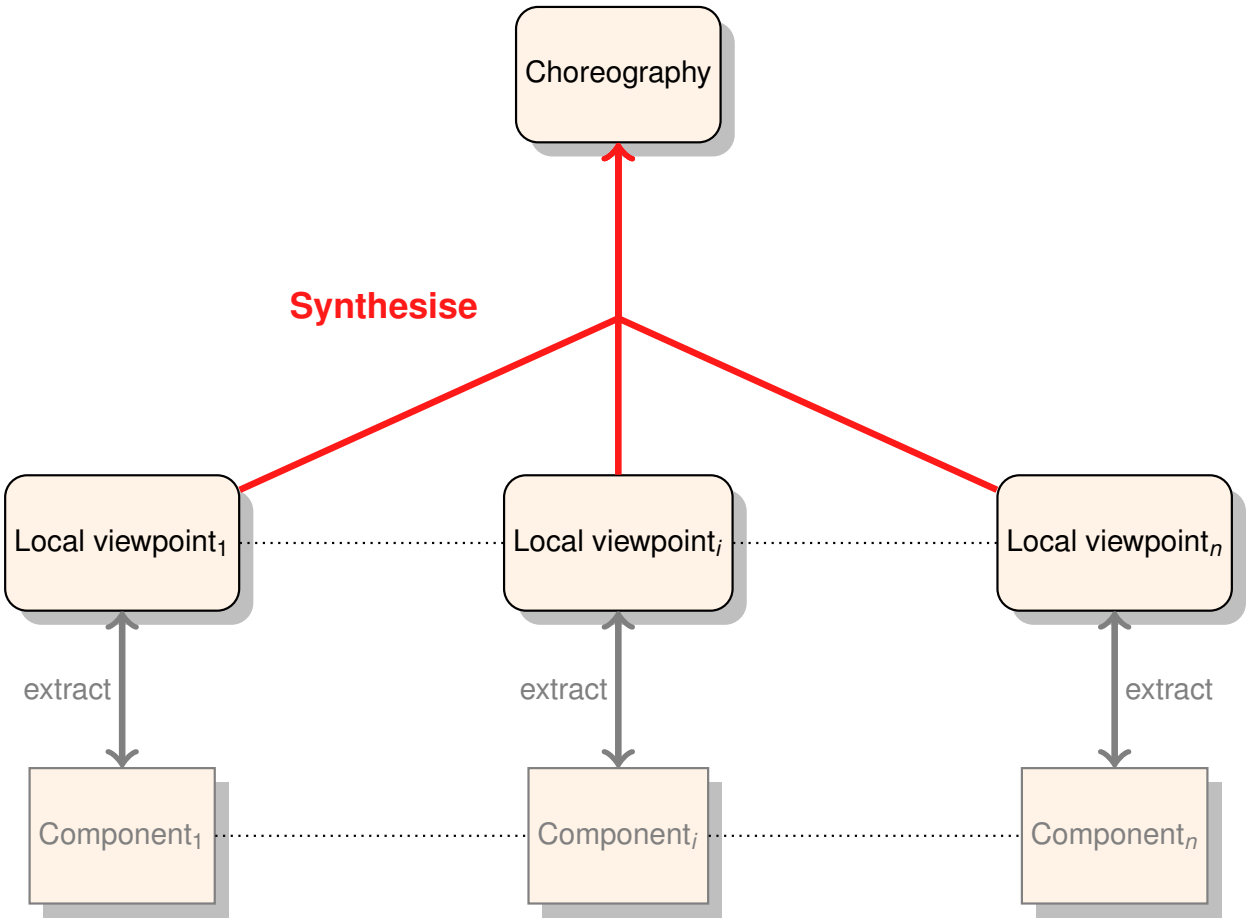
[Lange, Tuosto: CONCUR 2012]

From programs to designs



[Lange, Tuosto: CONCUR 2012]

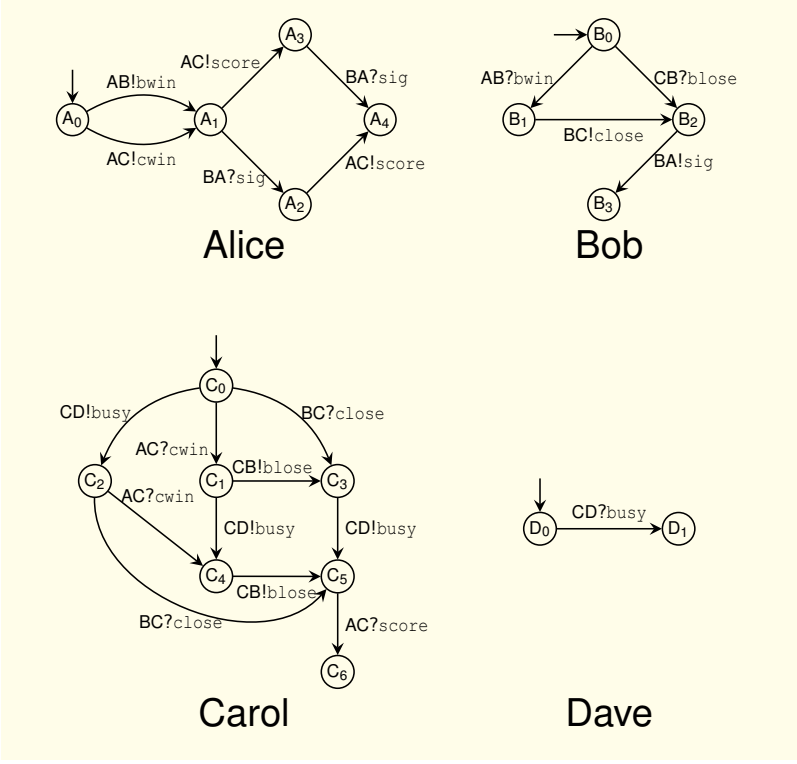
From programs to designs



[Lange, Tuosto: CONCUR 2012]

Synthesis: problem statement

Q: Given a set of CFSMs, do they “form” a choreography?



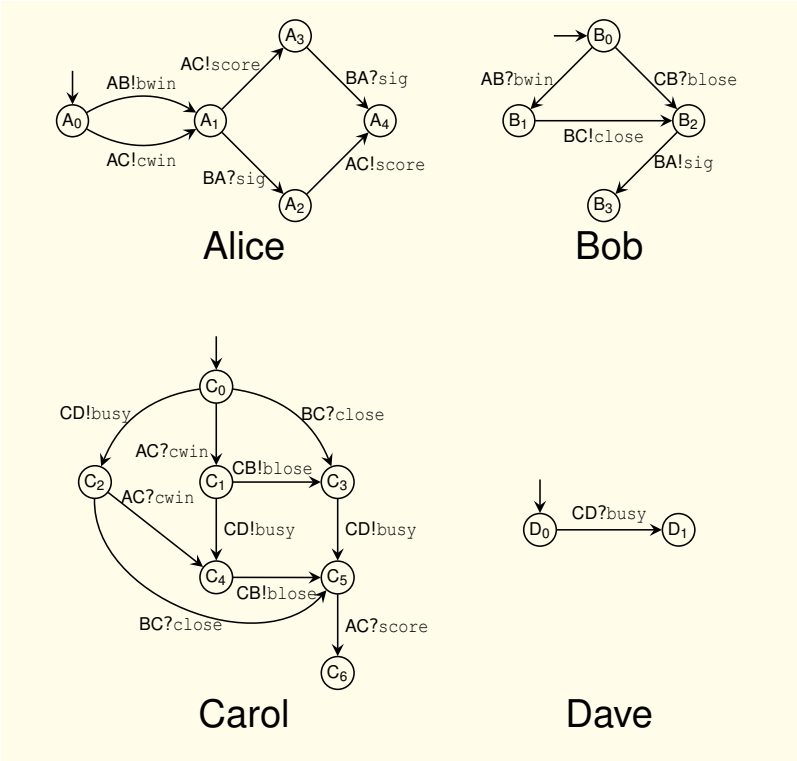
A: Not always...so let’s refine the statement

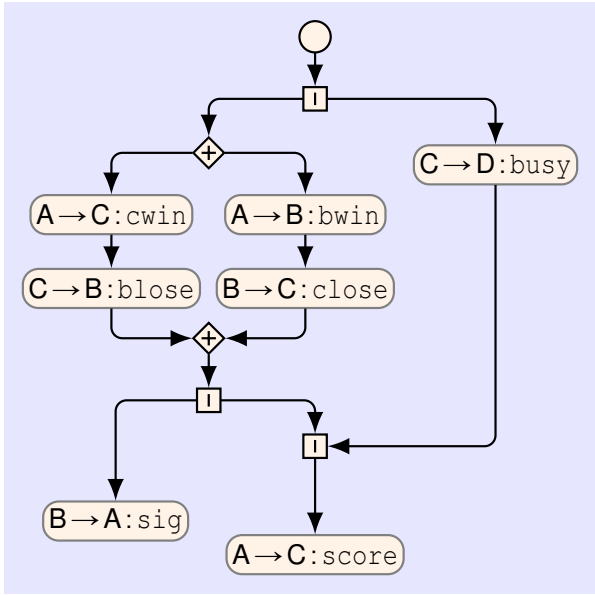
Q: Is there a class of (finite subsets of) CFSMs that “form” choreographies?

[Lange, Tuosto, Yoshida: POPL 2015]

Synthesis: problem statement

Q: Given a set of CFSMs, do they “form” a choreography?





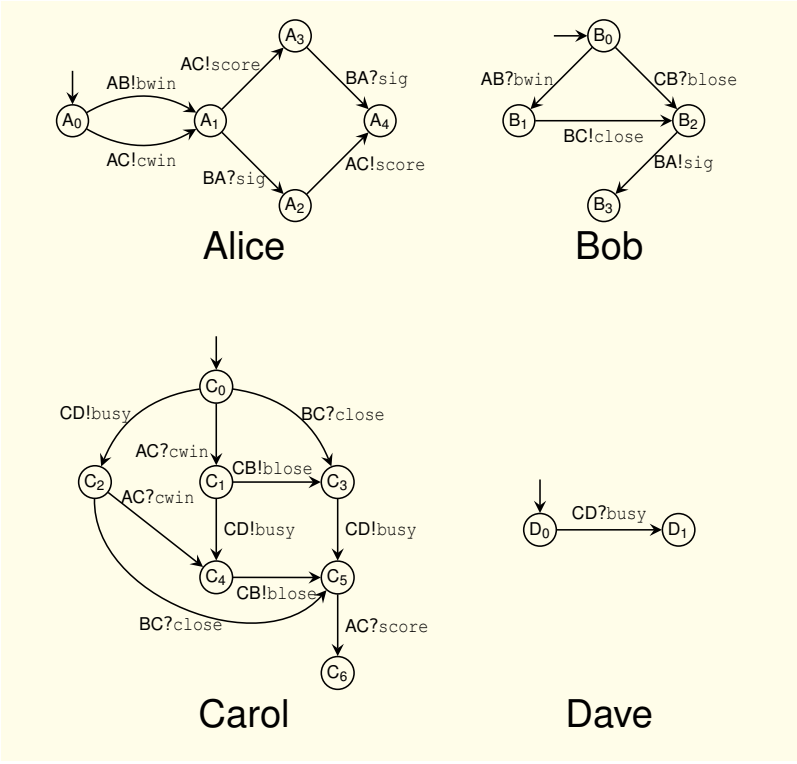
A: Not always...so let’s refine the statement

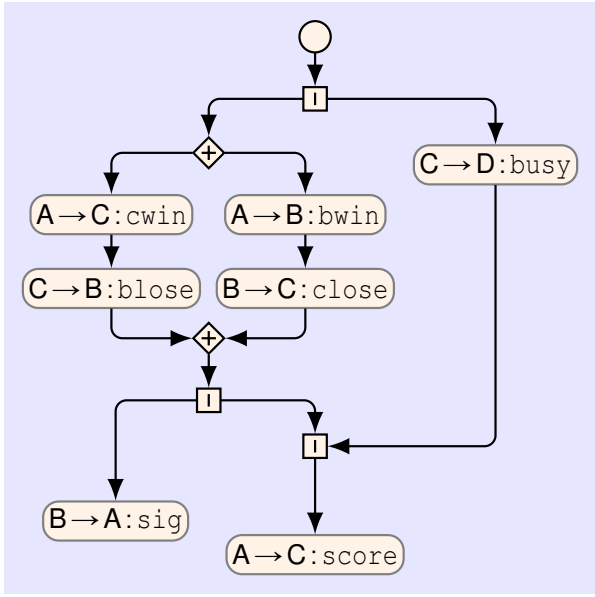
Q: Is there a class of (finite subsets of) CFSMs that “form” choreographies?

[Lange, Tuosto, Yoshida: POPL 2015]

Synthesis: problem statement

Q: Given a set of CFSMs, do they “form” a choreography?





A: Not always...so let’s refine the statement

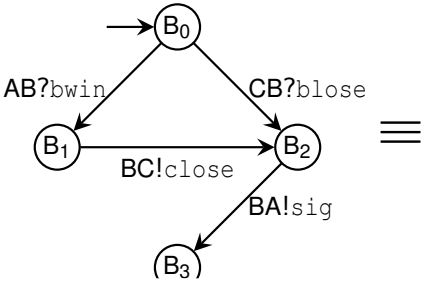
Q: Is there a class of (finite subsets of) CFSMs that “form” choreographies?

[Lange, Tuosto, Yoshida: POPL 2015]

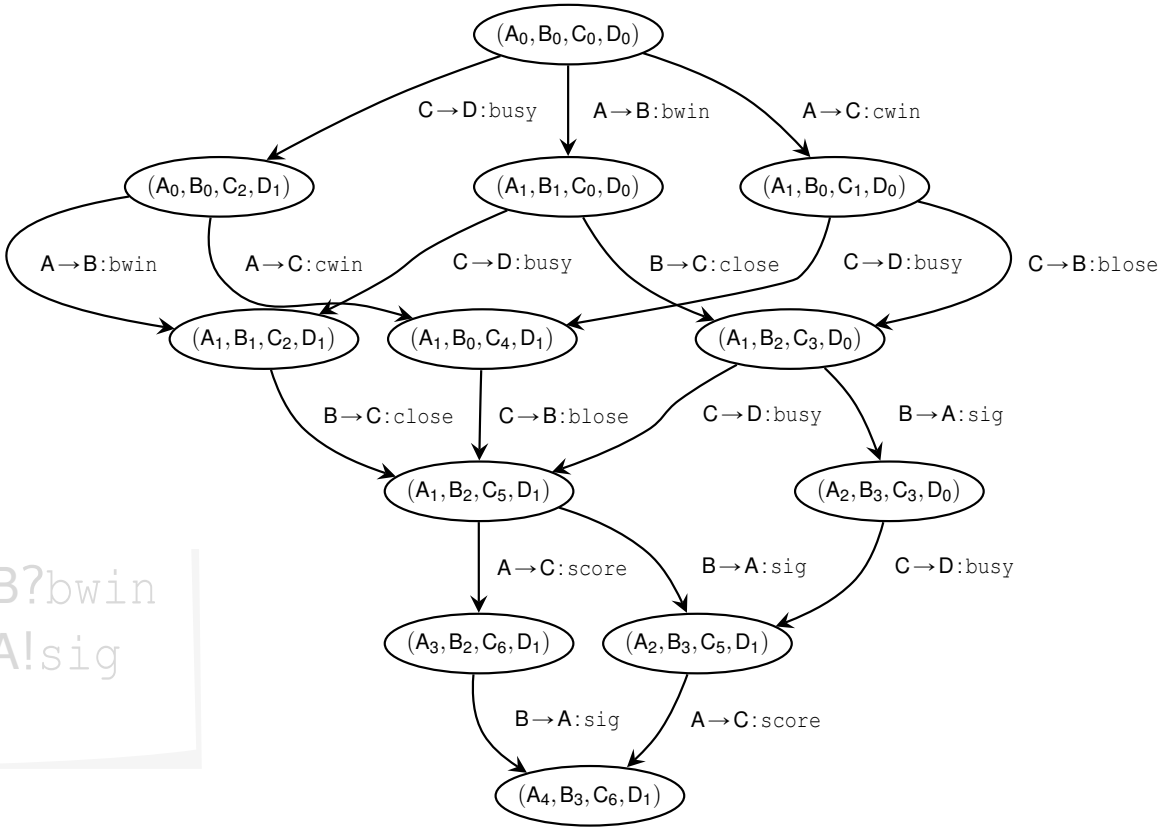
Checking Compatibility: Representability

Representability

- The projected TS $\equiv$ original machine
- Each branching in each machine must be represented in TS



$\equiv$

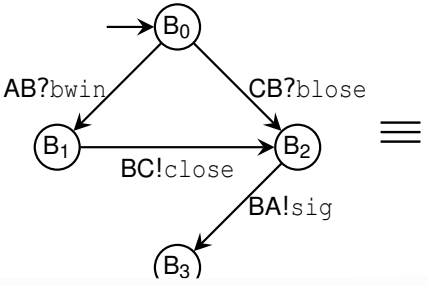


$$(A \to B : \texttt{bwin})\downarrow_B = AB?\texttt{bwin}$$
$$(B \to A : \texttt{sig})\downarrow_B = BA!\texttt{sig}$$
$$(C \to D : \texttt{busy})\downarrow_B = \varepsilon$$

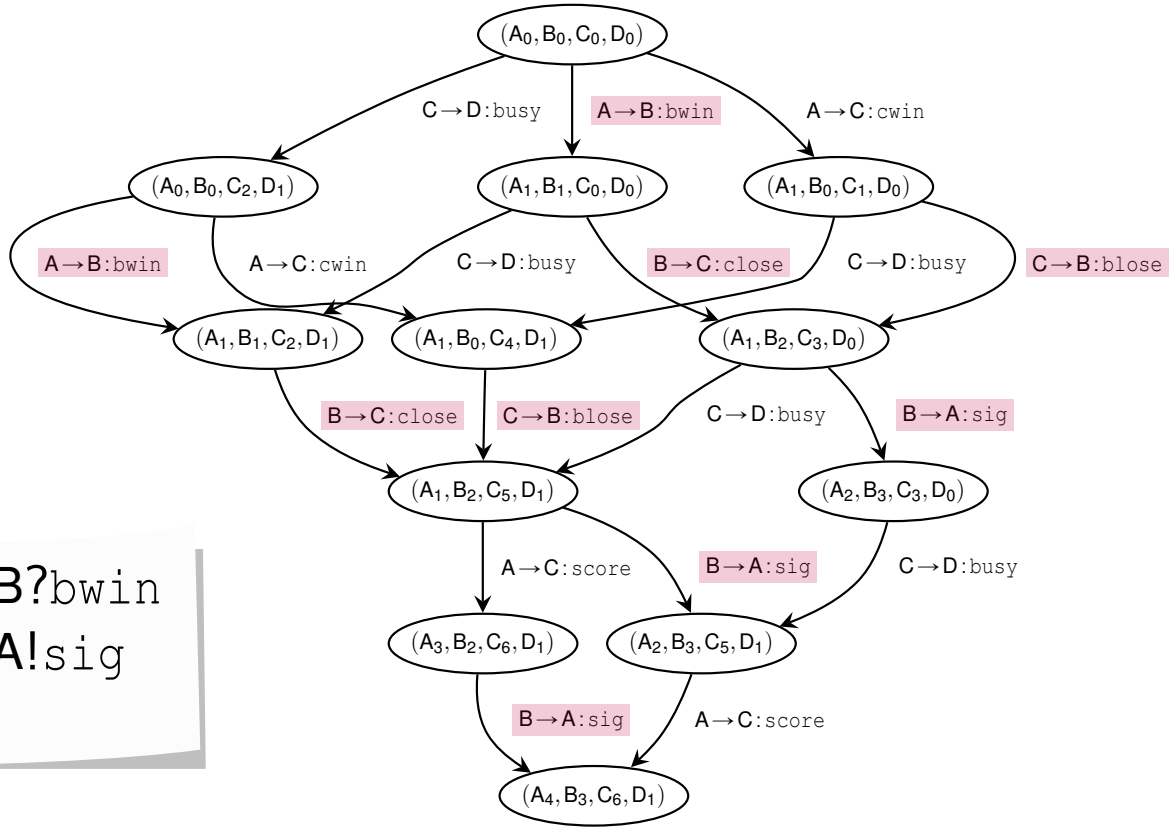
Checking Compatibility: Representability

Representability

- The projected TS $\equiv$ original machine
- Each branching in each machine must be represented in TS



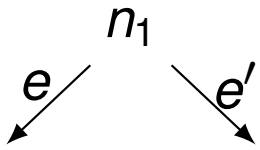
$\equiv$



$$(A \to B : \texttt{bwin})\downarrow_B = AB?\texttt{bwin}$$
$$(B \to A : \texttt{sig})\downarrow_B = BA!\texttt{sig}$$
$$(C \to D : \texttt{busy})\downarrow_B = \varepsilon$$

Checking Compatibility: Branching Property

Branching Property:

each branching 

in TS must be either

- commuting:

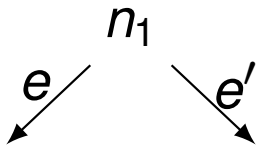
- or, each *last node* n_k

must be a “well-formed” choice, i.e.,

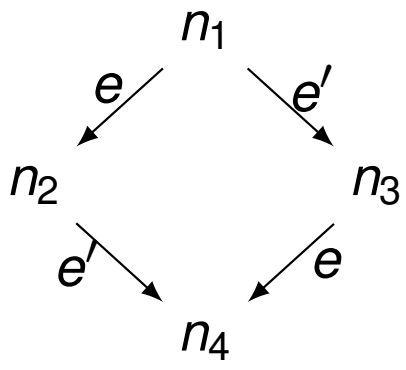
- each participant
  - receives a different message in each branch, or
  - is not involved in the choice
- there is a unique sender

Checking Compatibility: Branching Property

Branching Property:

each branching 

in TS must be either

- commuting: 

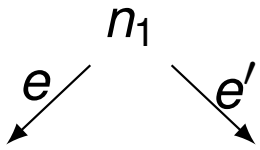
- or, each *last node* n_k

must be a “well-formed” choice, i.e.,

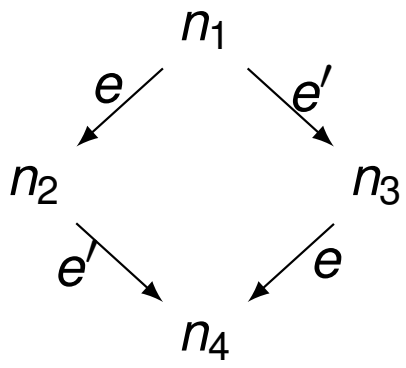
- each participant
  - receives a different message in each branch, or
  - is not involved in the choice
- there is a unique sender

Checking Compatibility: Branching Property

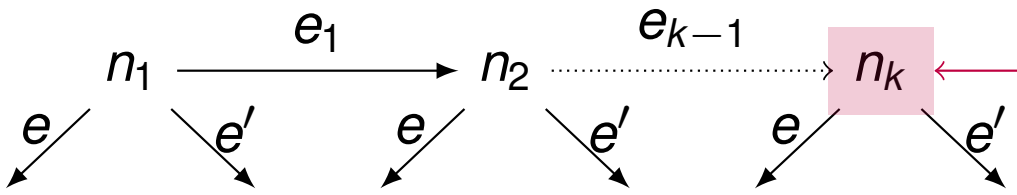
Branching Property:

each branching 

in TS must be either

- commuting: 

- or, each *last node* n_k

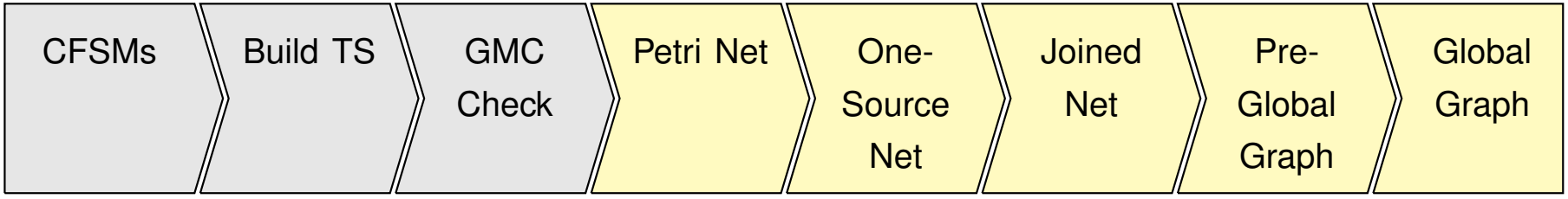


Last node reachable from n_1, from which e and e' can be fired.

must be a "well-formed" choice, i.e.,

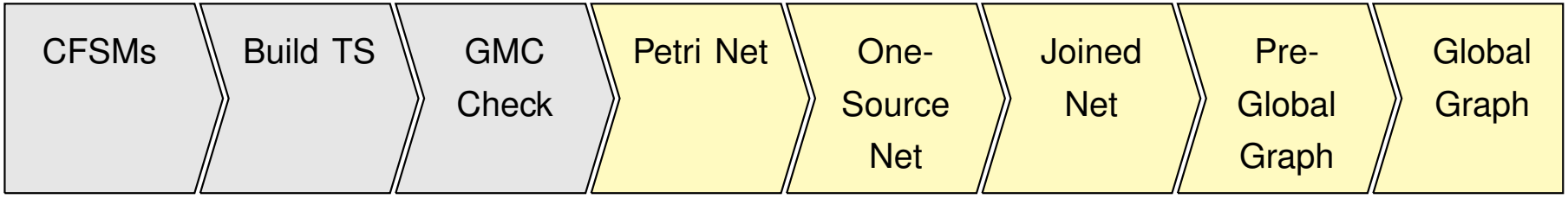
- each participant
  - receives a different message in each branch, or
  - is not involved in the choice
- there is a unique sender

Transformation Workflow

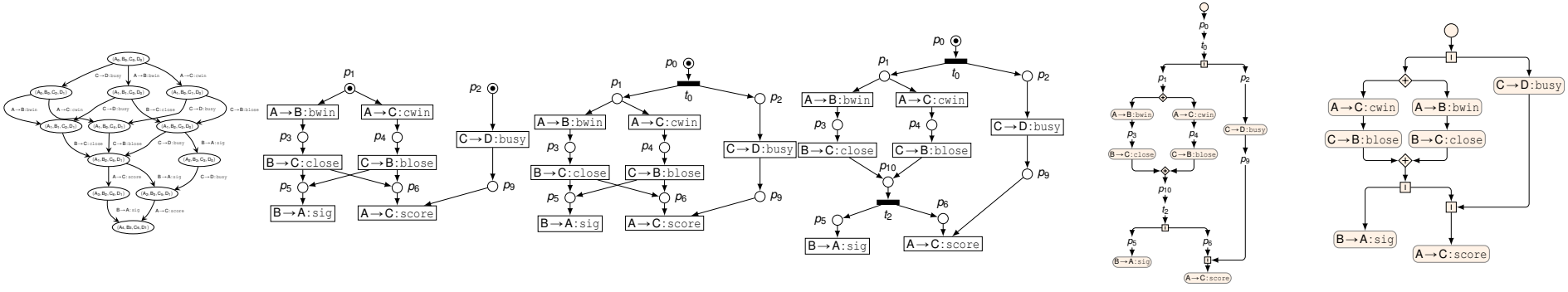


We use the work of Cortadella et al. (IEEE TC'98), based on the theory of regions, to synthesise a **safe** and **extended free-choice** Petri net from the Synchronous Transition System.

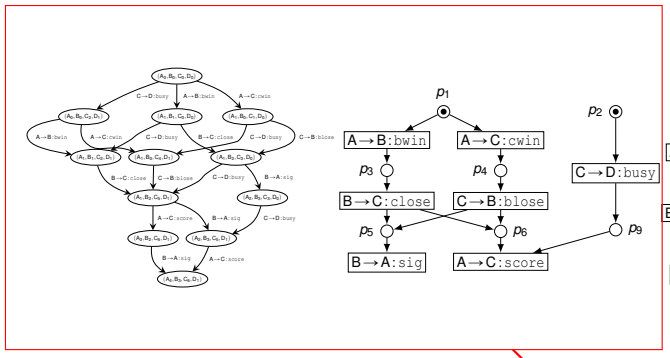
Transformation Workflow

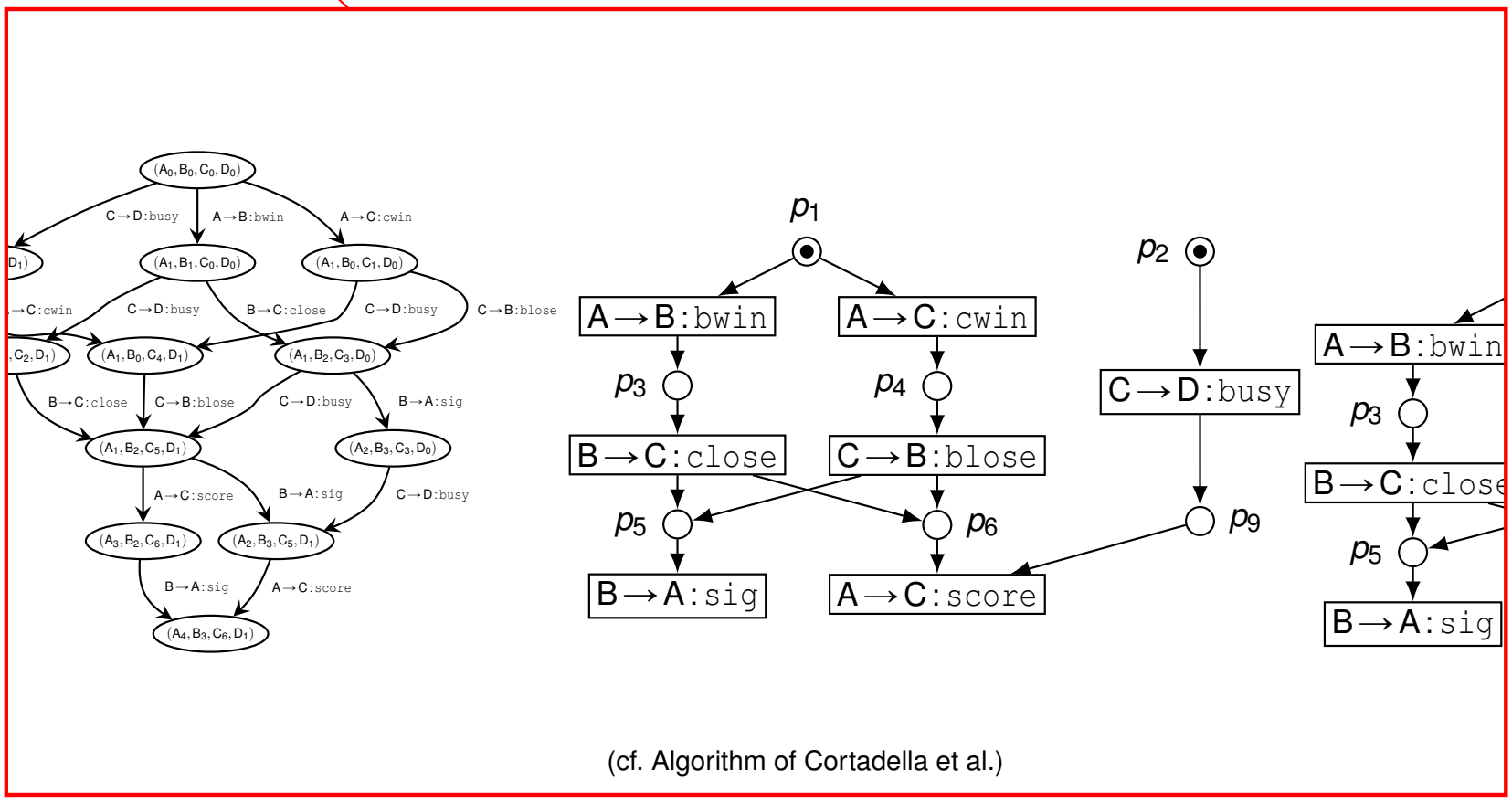


We use the work of Cortadella et al. (IEEE TC'98), based on the theory of regions, to synthesise a **safe** and **extended free-choice** Petri net from the Synchronous Transition System.



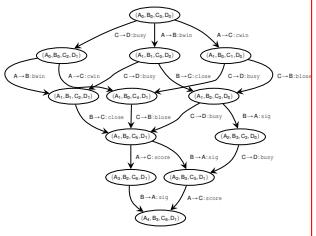
Step 1: TS ⇝ PN

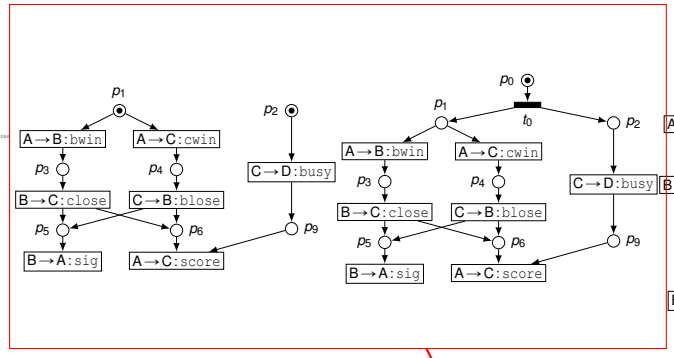


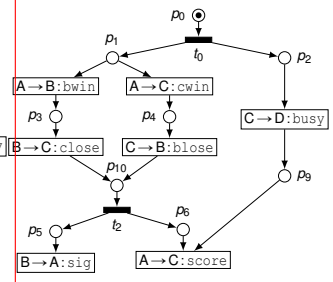


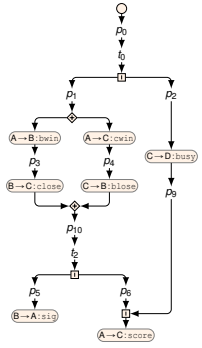
(cf. Algorithm of Cortadella et al.)

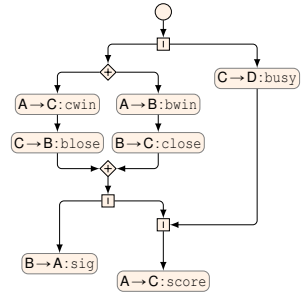
Step 2: PN ⇝ 1-source PN

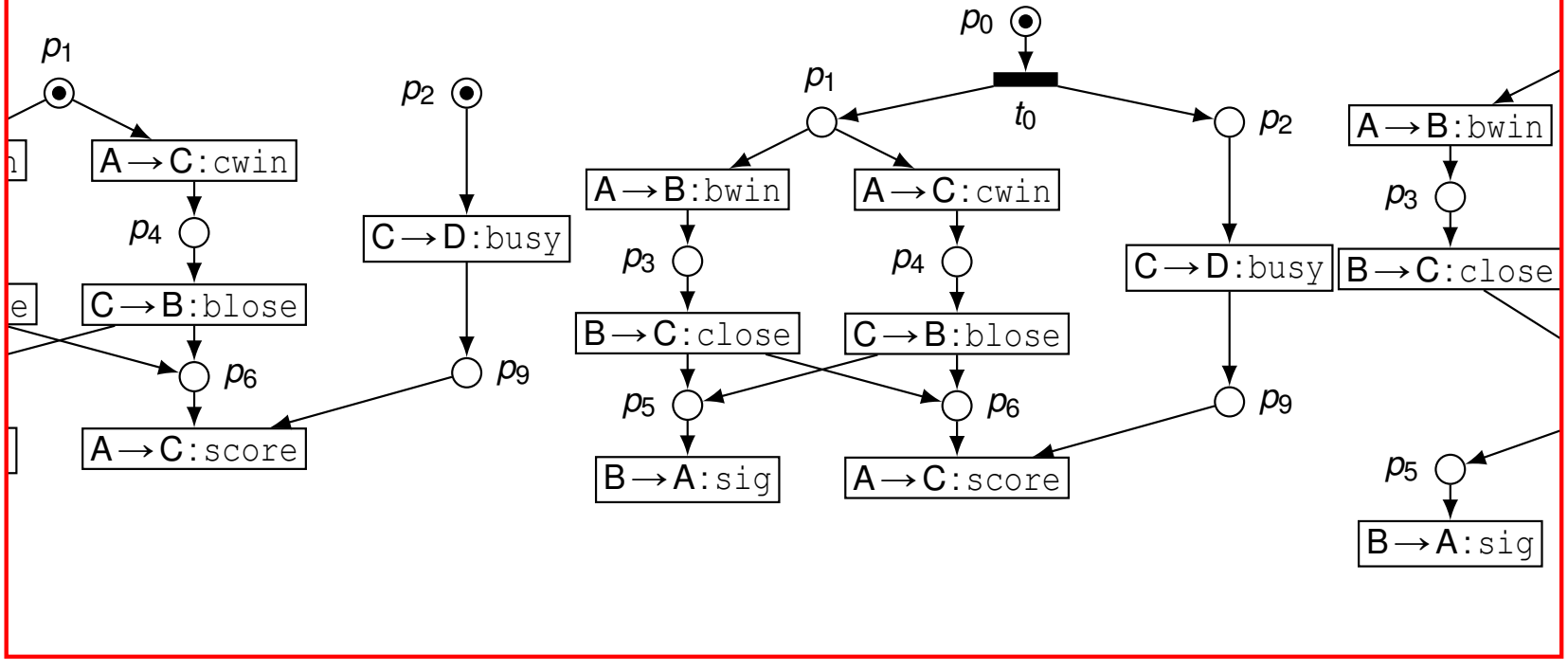




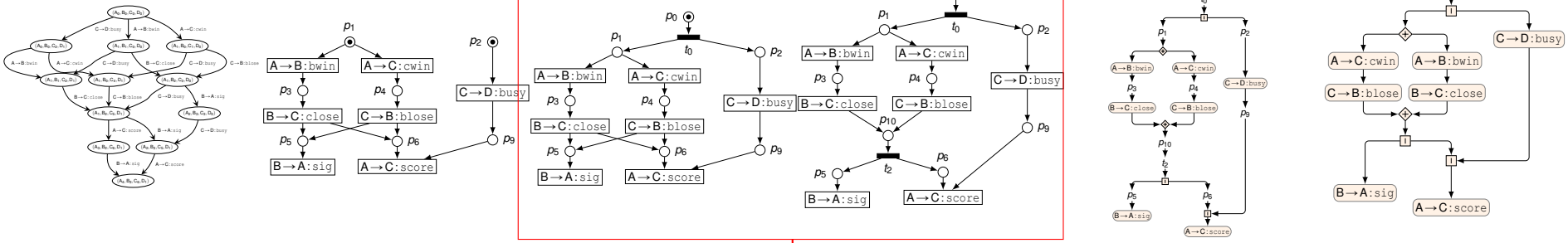


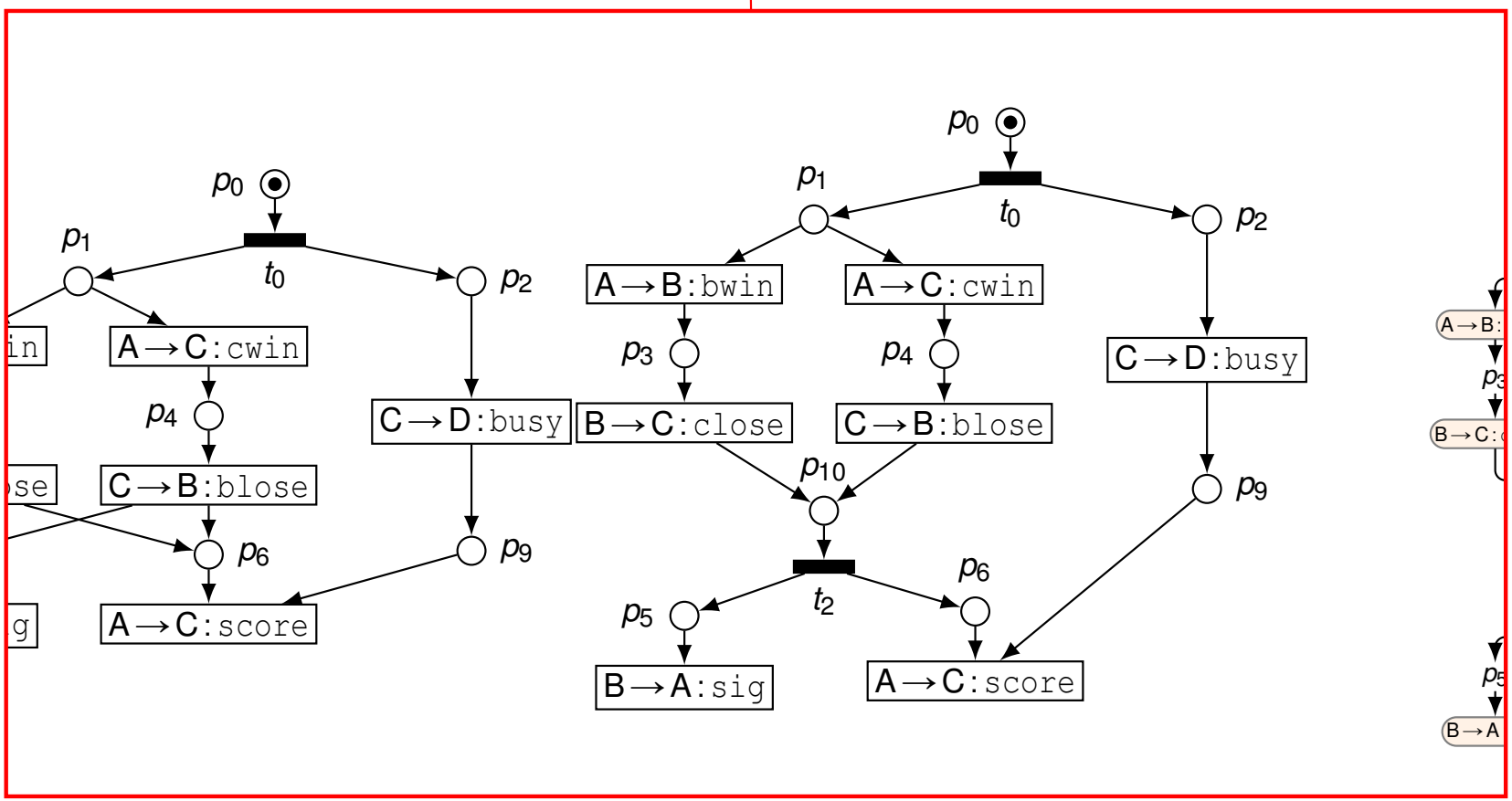




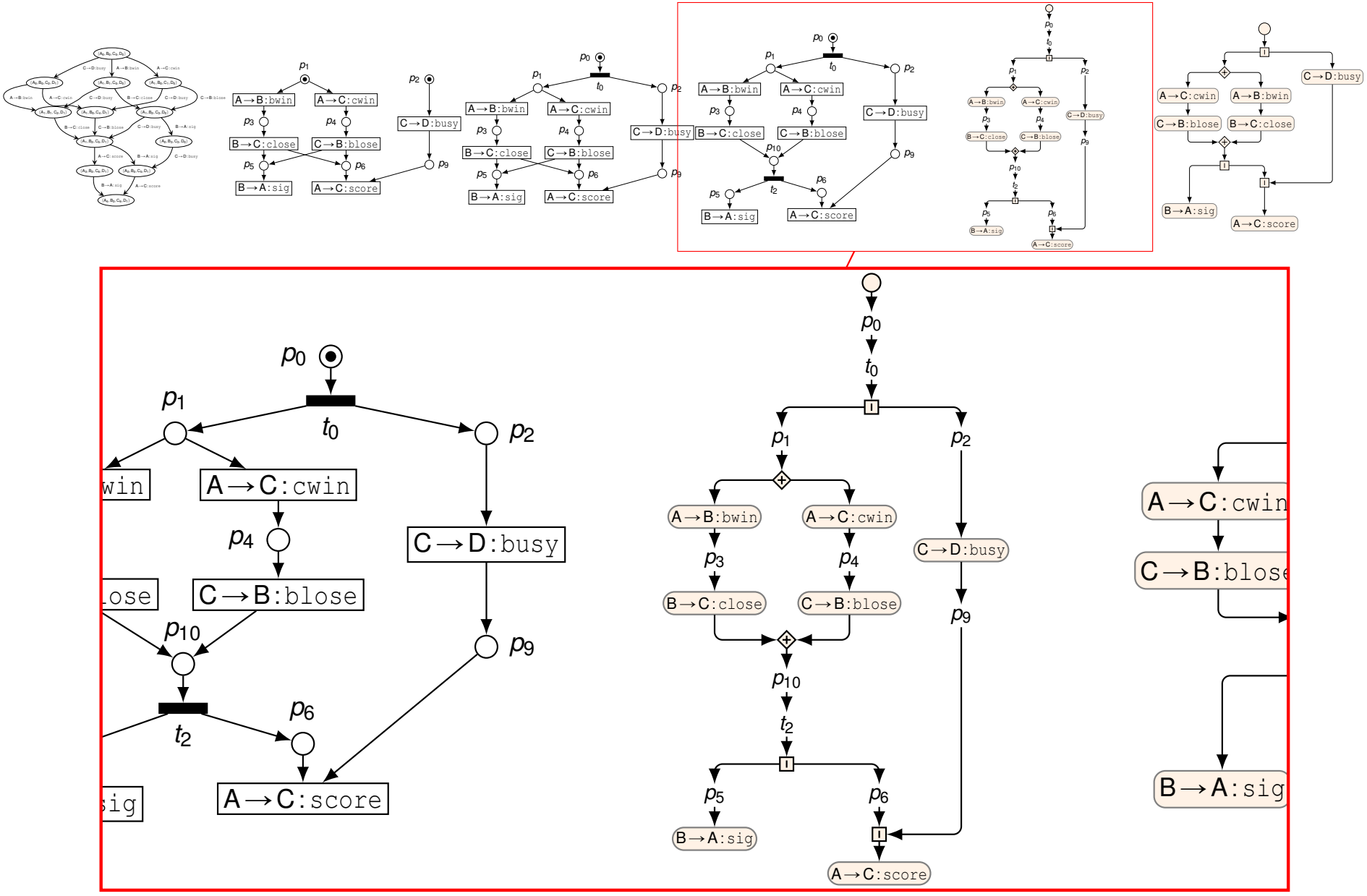


Step 3: 1-source PN ⇝ Joined PN

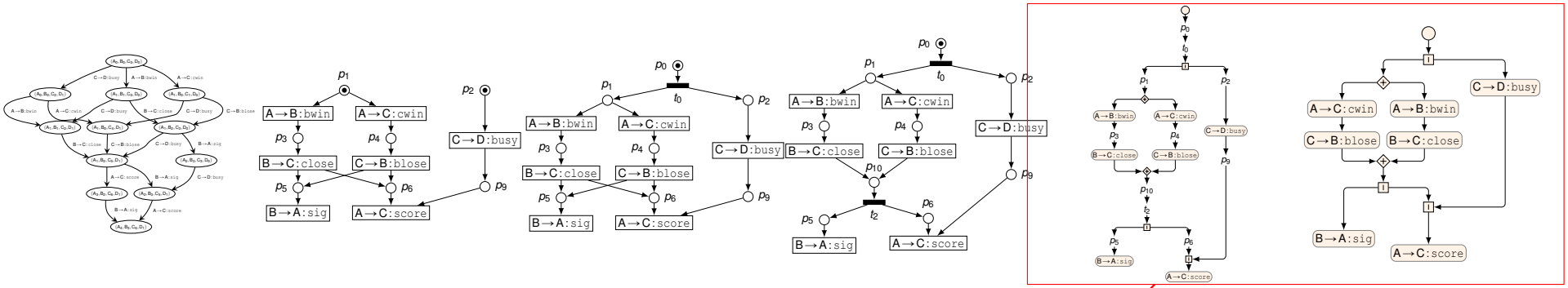


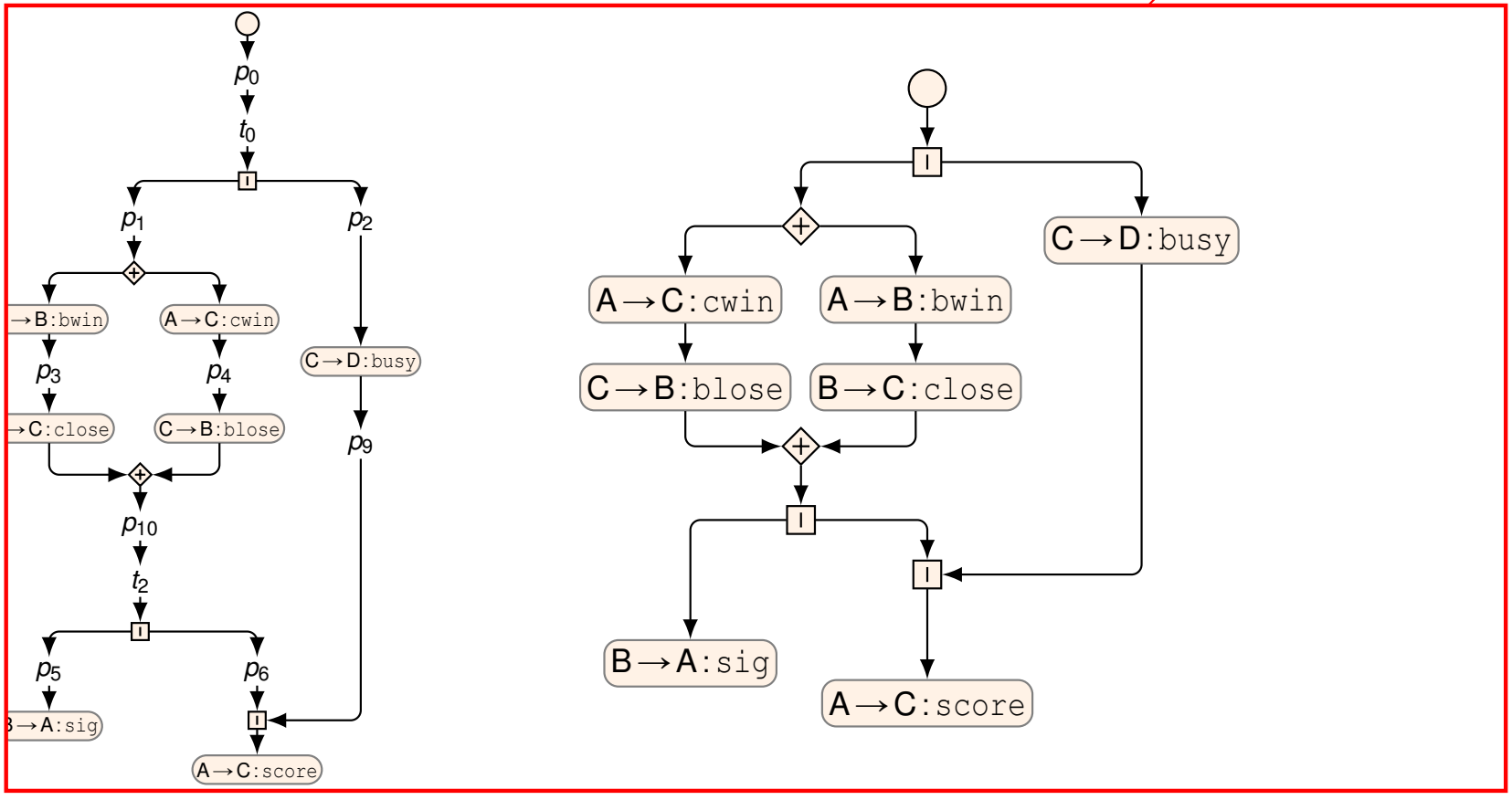


Step 4: Joined PN ⇝ Pre-Interaction Graph



Step 5: Pre-Interaction Graph ⇝ Interaction Graph



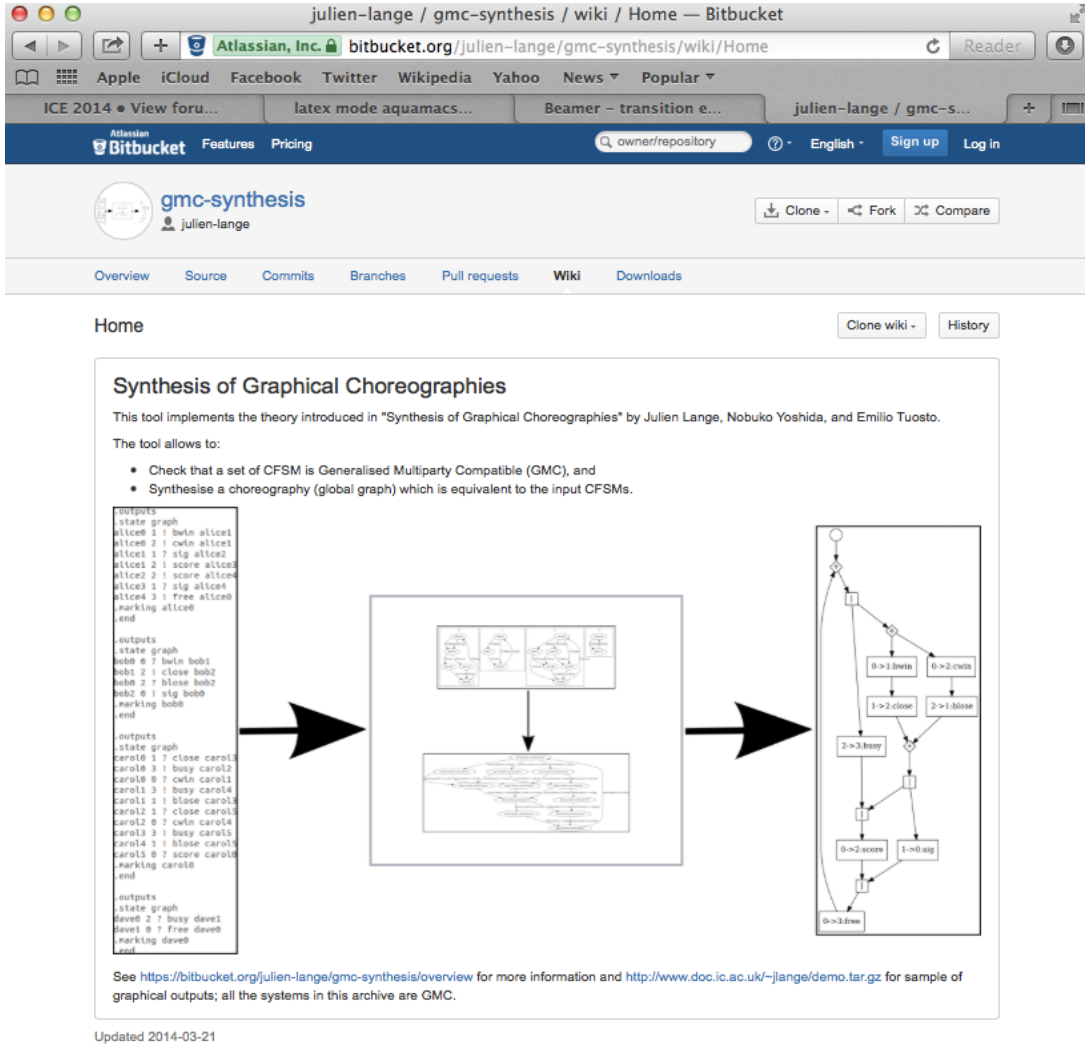


Conclusions

Our tool

Check out *ChoSyn* at

`https://bitbucket.org/emlio_tuosto/gmc-synthesis-v0.2`



Thanks to...

Vasco & Kohei

for writing the paper I used

for teaching me choreographies

# of course

YOU & MGS!

my friends/colleagues

Julien
Lange

Laura
Bocchi

Paula
Severi

Kohei
Honda

Nobuko
Yoshida

Hernan
Melgratti

Mariangiola
Dezani-Ciancaglini

Luca
Padovani

Roberto
Guanciale

Questions?