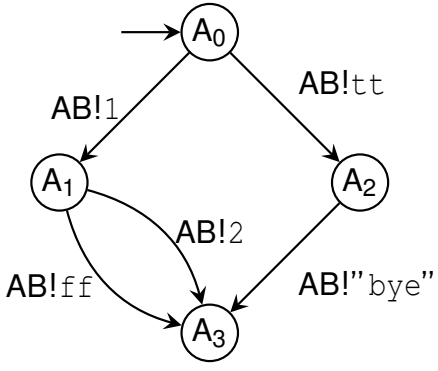
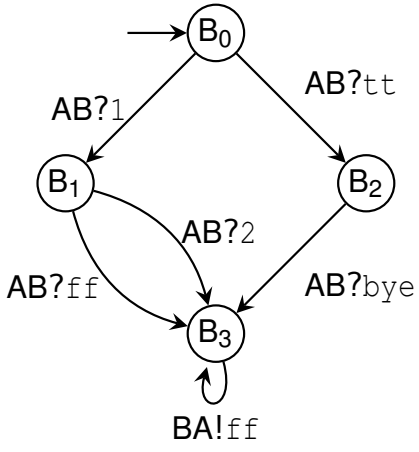
Communicating finite state machines



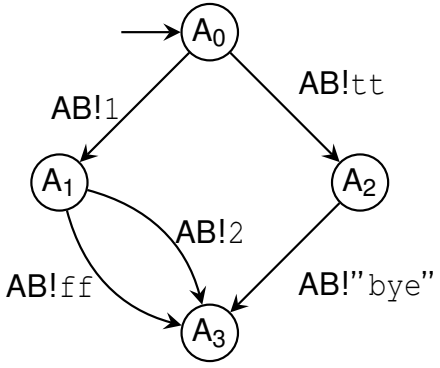
A



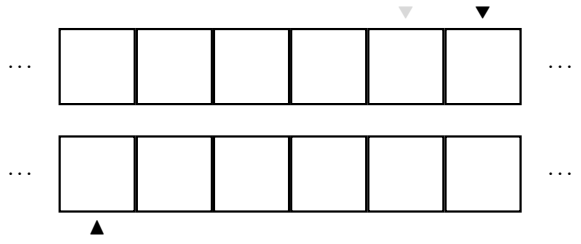
B

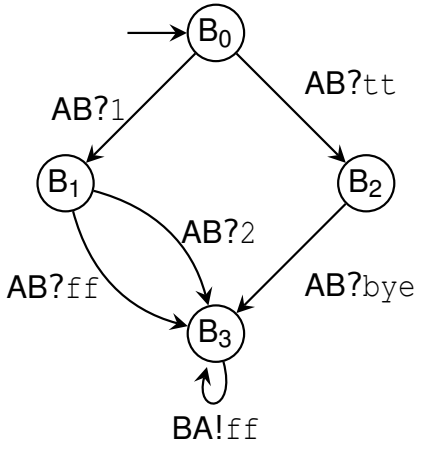
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

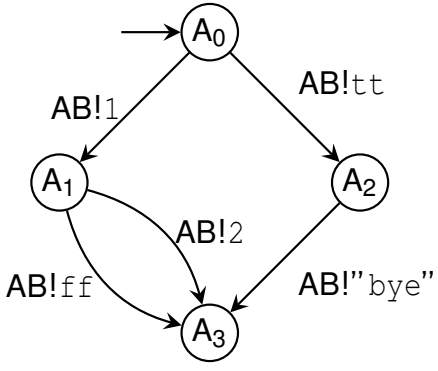




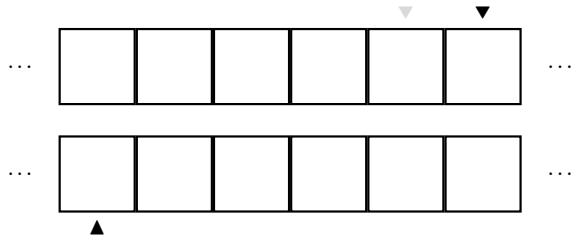
B

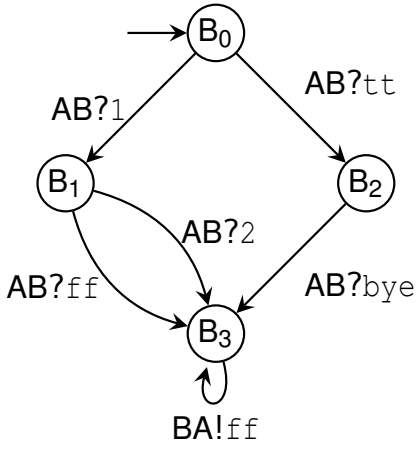
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

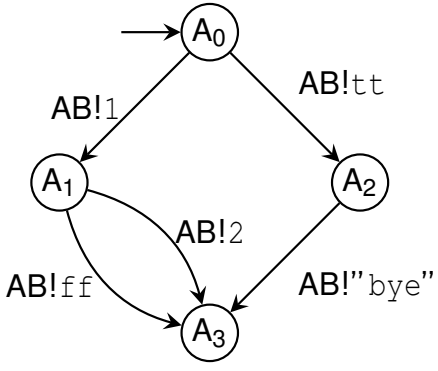




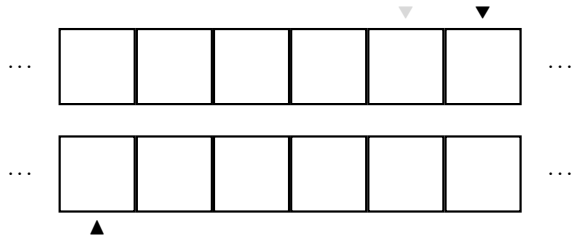
B

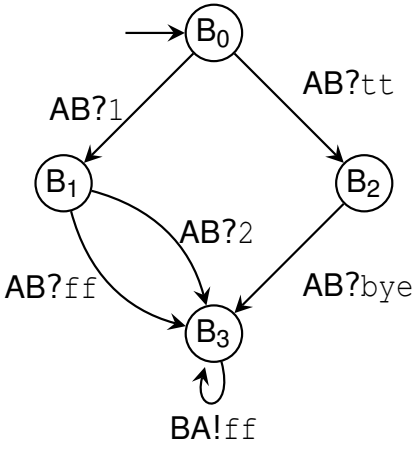
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A



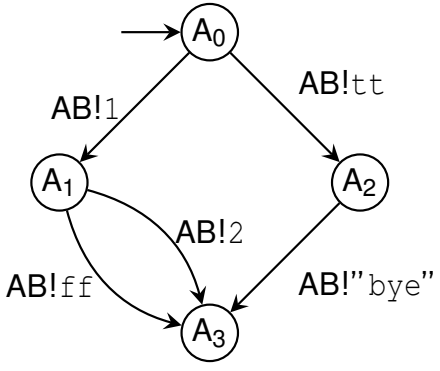


B

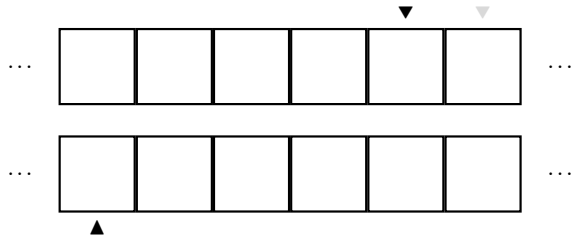
ff

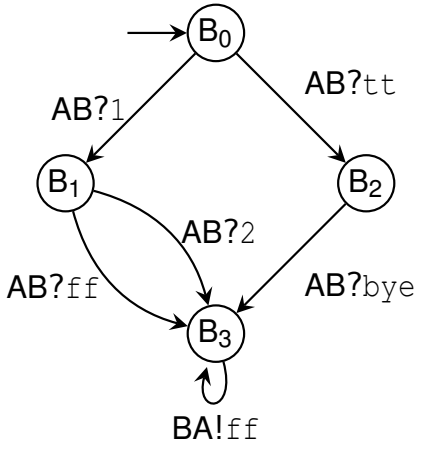
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

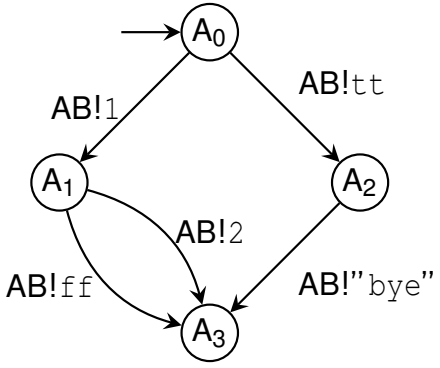




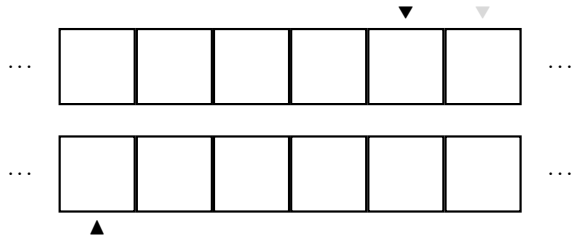
B

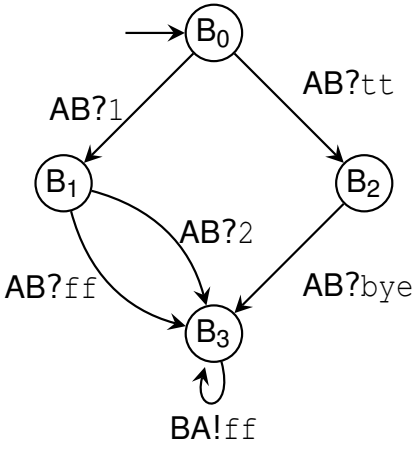
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

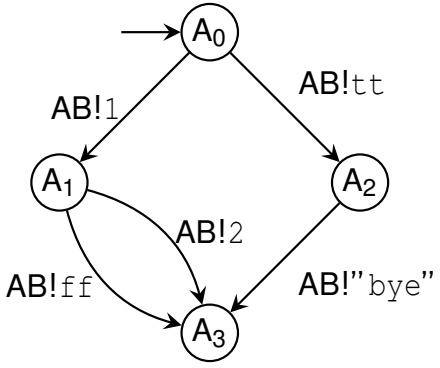




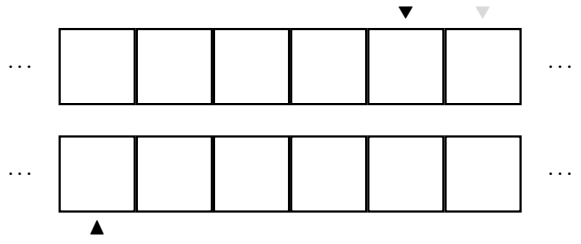
B

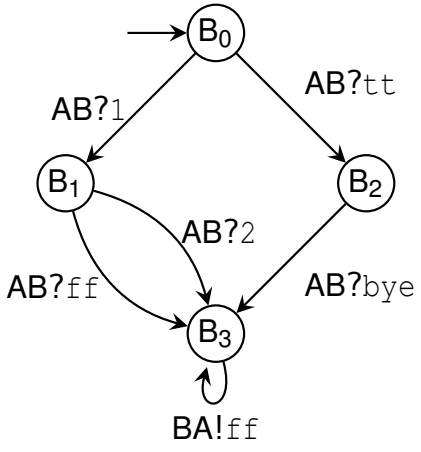
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A

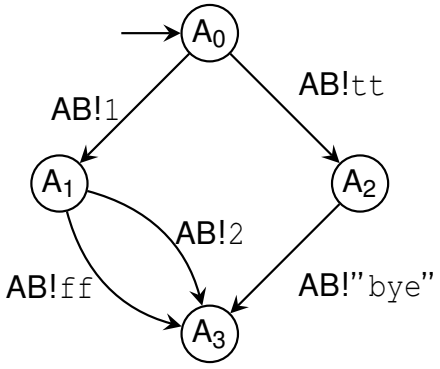




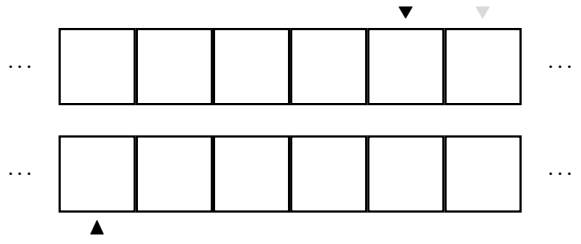
B

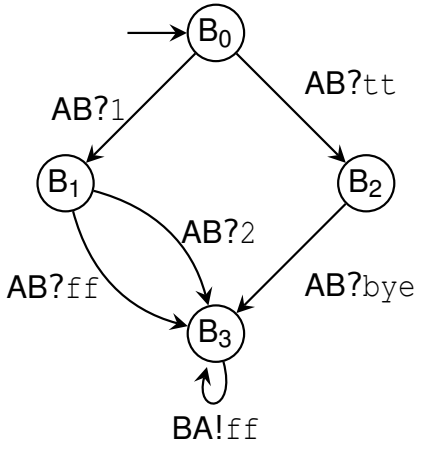
cf. Brand & Zafiropulo 1983

Communicating finite state machines



A





B

cf. Brand & Zafiropulo 1983

Choreographies 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]]
- Spawn of threads

Asynchrony by design

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

Exercise:

What does this program do?

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]]
- Spawn of threads

Asynchrony by design

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

Exercise:

What does this program do?

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]]
- Spawn of threads

Asynchrony by design

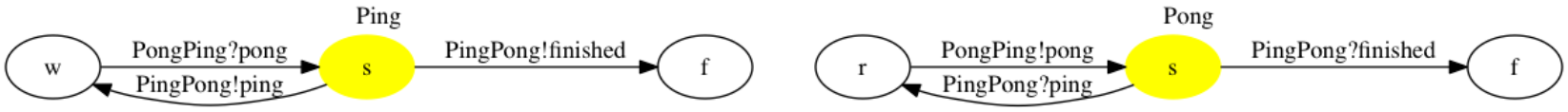
Erlang is an incarnation of the well-known actor model of Hewitt and Agha...dates back to ’73!

Exercise:

What does this program do?

Friendlier representations

Local behaviour: communicating machines

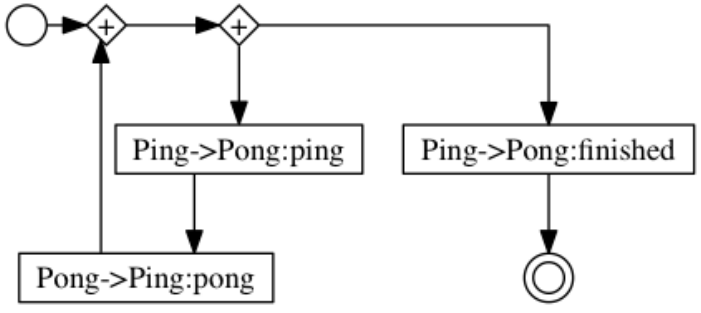


ChoSyn

...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 not familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

What does this program do? Does it work?

Exercise:

find the bug

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 not familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

What does this program do? Does it work?

Exercise:

find the bug

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 not familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

What does this program do? Does it work?

Exercise:

find the bug

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 not familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

What does this program do? Does it work?

Exercise:

find the bug

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 not familiar with Erlang?

Q:

Is this program correct?

A:

No!

Exercise:

What does this program do? Does it work?

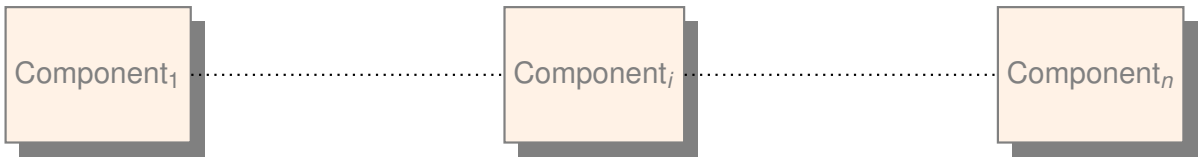
Exercise:

find the bug

Tools help...

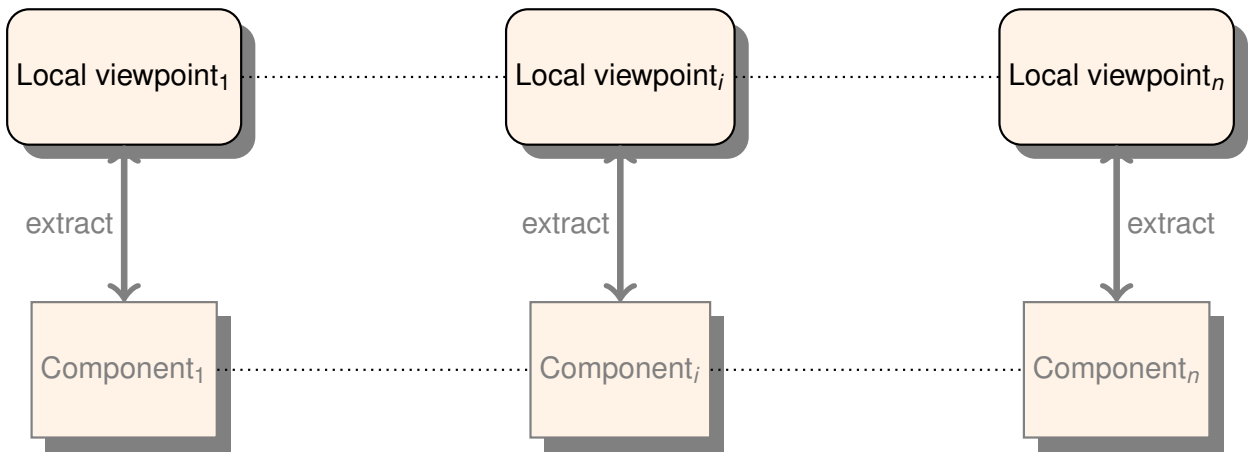
go to shell!

From programs to designs



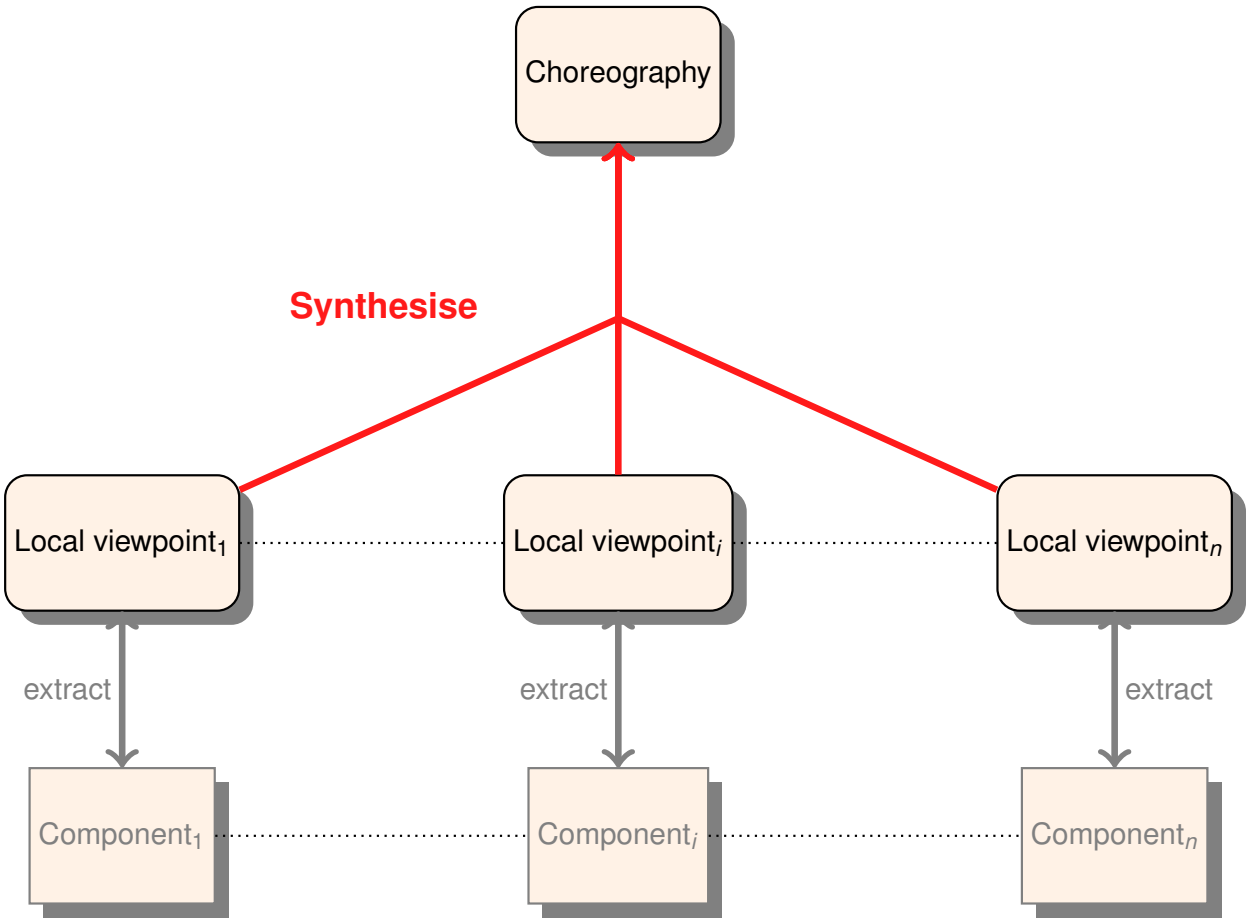
[Lange, Tuosto: CONCUR 2012]

From programs to designs



[Lange, Tuosto: CONCUR 2012]

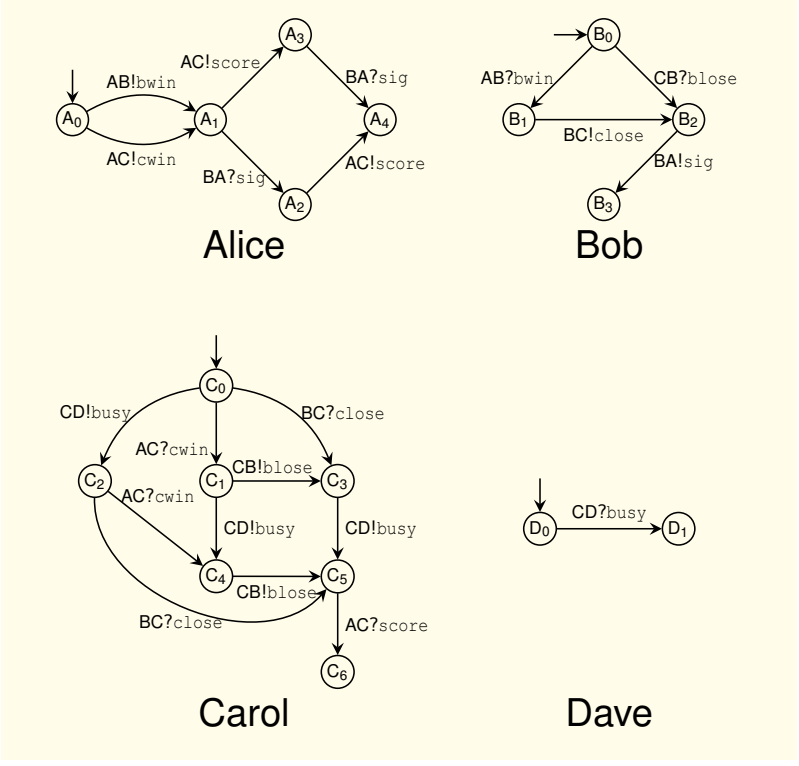
From programs to designs



[Lange, Tuosto: CONCUR 2012]

Synthesis: problem statement

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



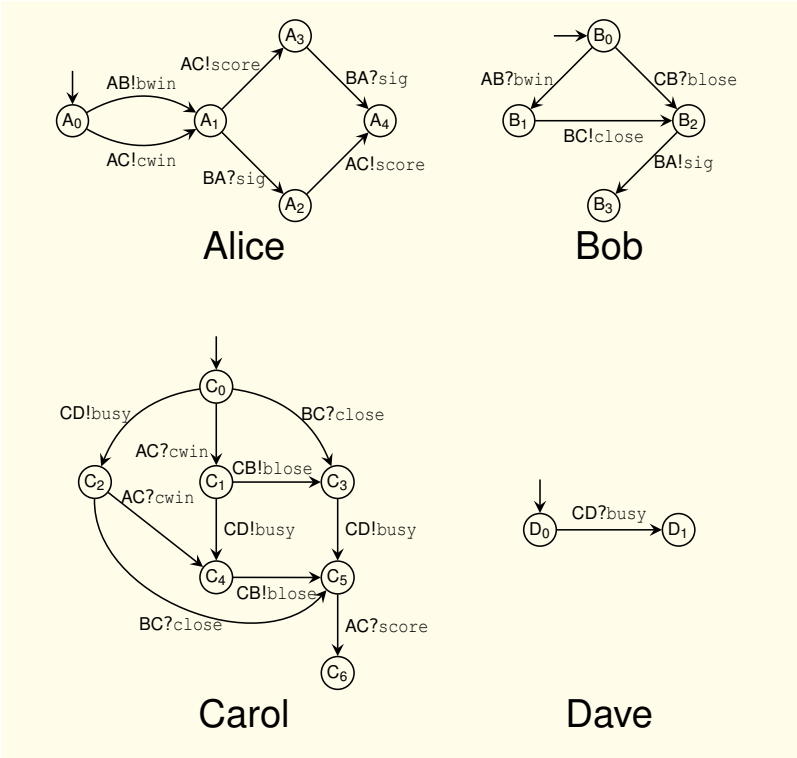
A: Not always...so let’s refine the statement

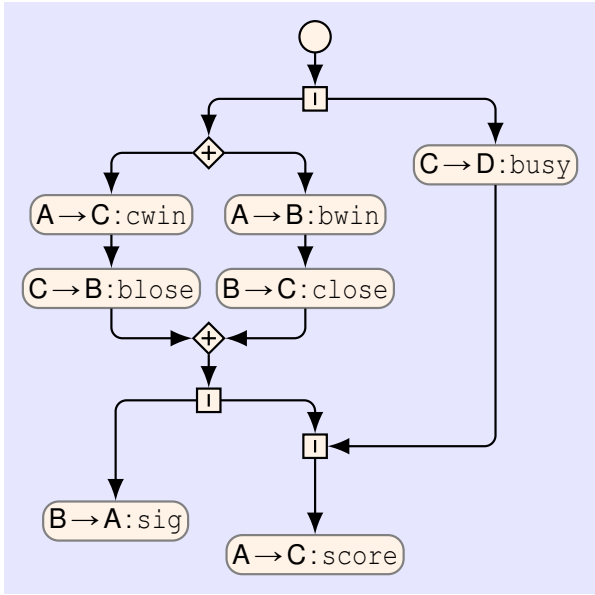
Q: Is there a class of (finite subsets of) CFSMs that “form” choreographies?

[Lange, Tuosto, Yoshida: POPL 2015]

Synthesis: problem statement

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





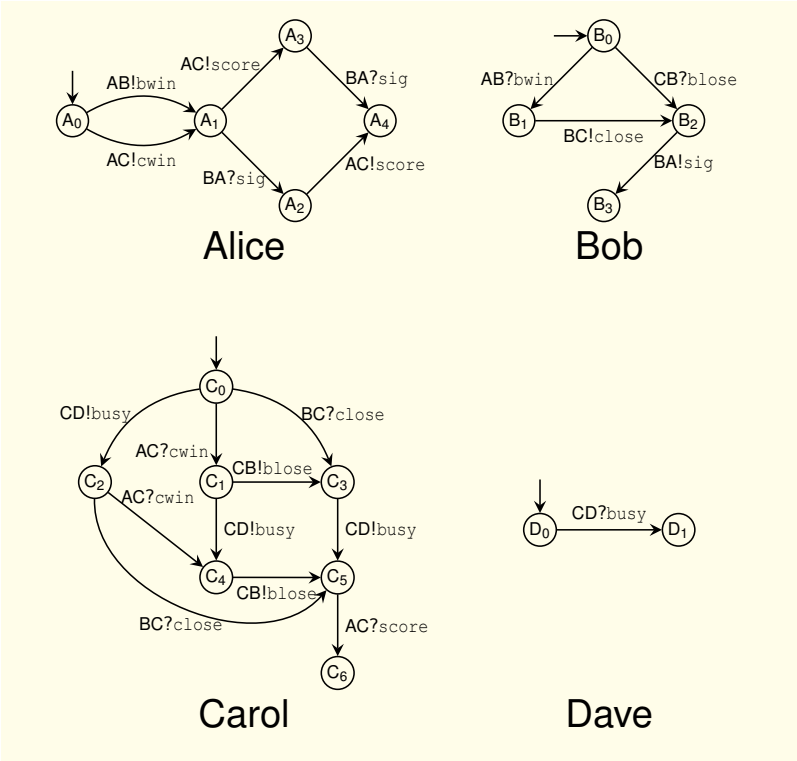
A: Not always...so let’s refine the statement

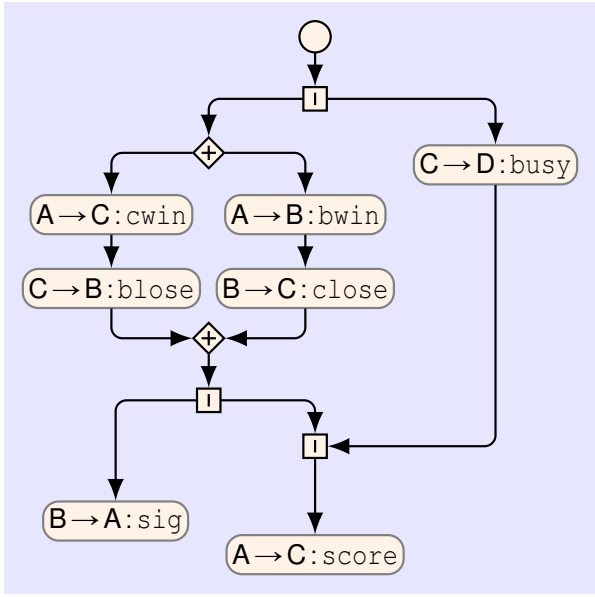
Q: Is there a class of (finite subsets of) CFSMs that “form” choreographies?

[Lange, Tuosto, Yoshida: POPL 2015]

Synthesis: problem statement

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





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

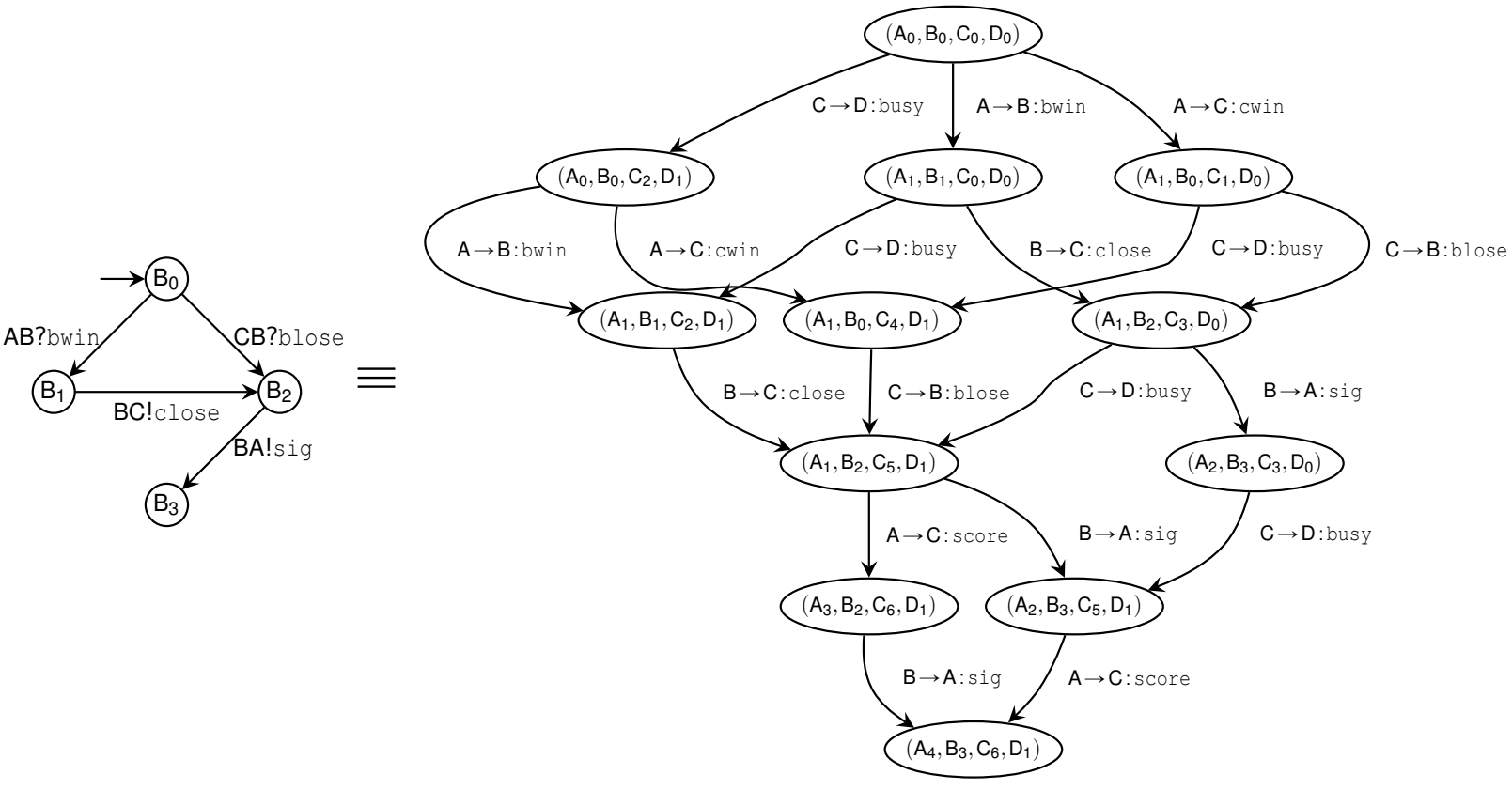
Q: Is there a class of (finite subsets of) CFSMs that “form” choreographies?

[Lange, Tuosto, Yoshida: POPL 2015]

Checking Compatibility: Representability

Representability

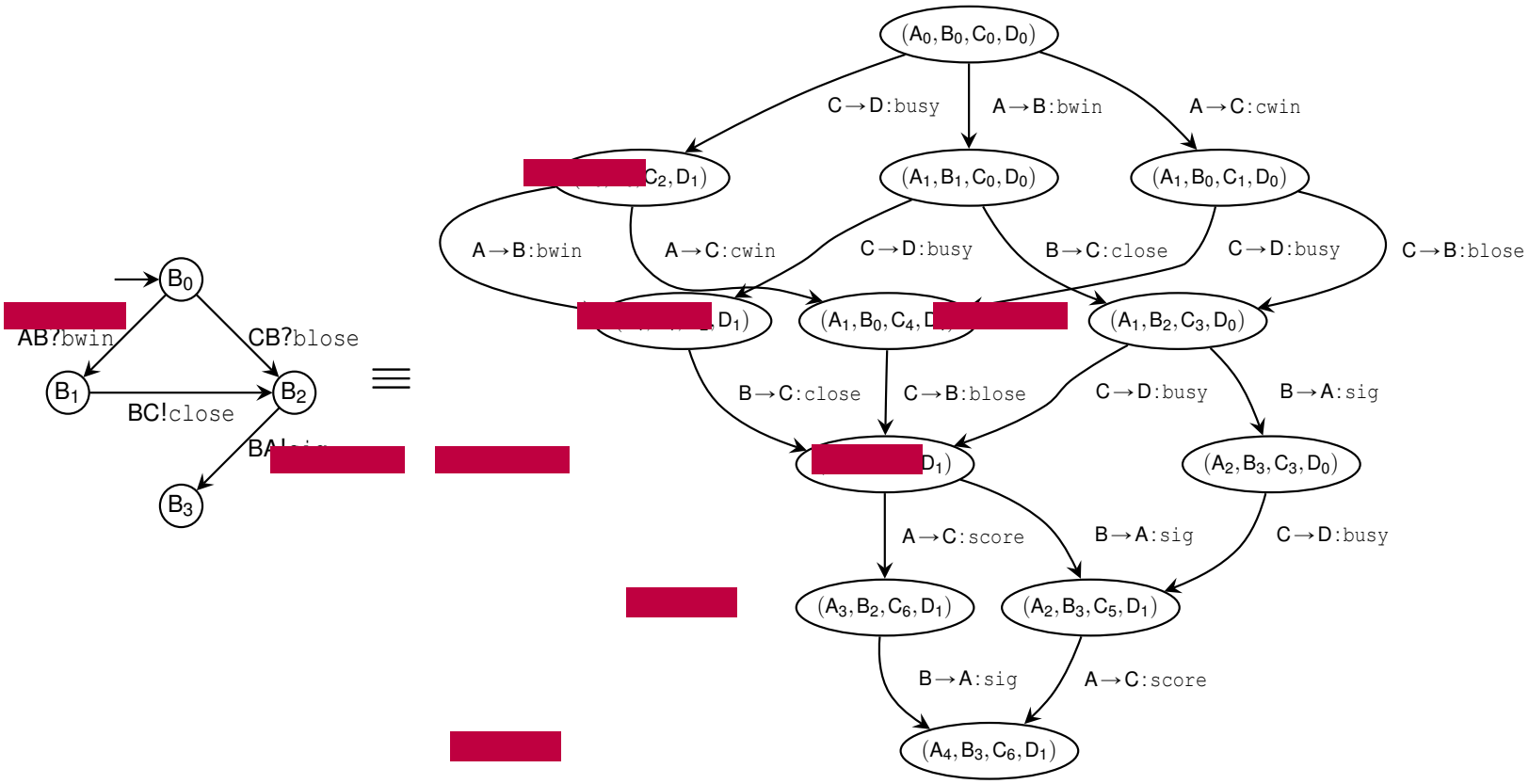
- The projected TS ≡ original machine
- Each branching in each machine must be represented in TS



Checking Compatibility: Representability

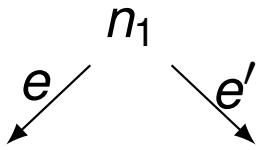
Representability

- The projected TS ≡ original machine
- Each branching in each machine must be represented in TS



Checking Compatibility: Branching Property

Branching Property:

each branching 

in TS must be either

- commuting:

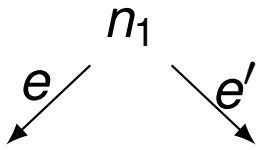
- or, each *last node* n_k

must be a “well-formed” choice, i.e.,

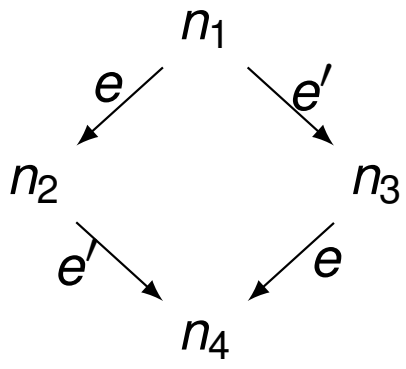
- each participant
  - receives a different message in each branch, or
  - is not involved in the choice
- there is a unique sender

Checking Compatibility: Branching Property

Branching Property:

each branching 

in TS must be either

- commuting: 

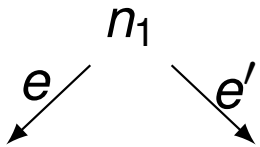
- or, each *last node* n_k

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

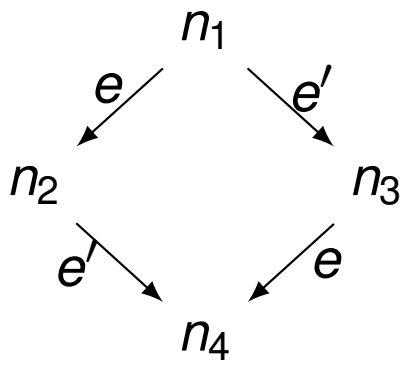
- each participant
  - receives a different message in each branch, or
  - is not involved in the choice
- there is a unique sender

Checking Compatibility: Branching Property

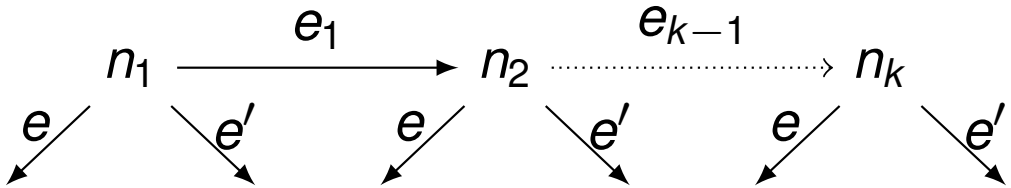
Branching Property:

each branching 

in TS must be either

- commuting: 

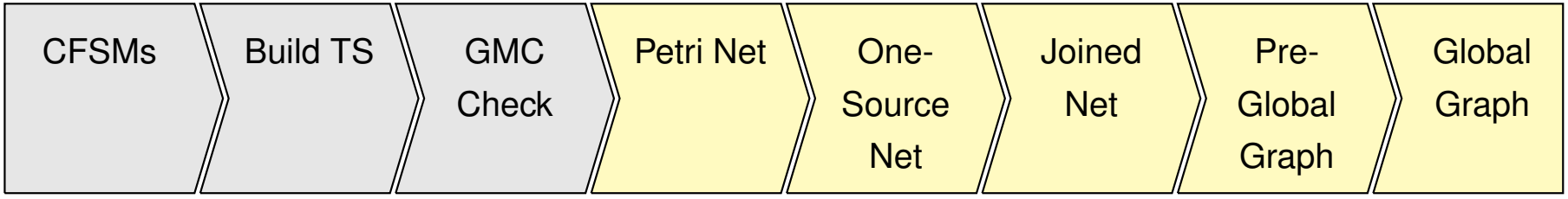
- or, each *last node* n_k



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

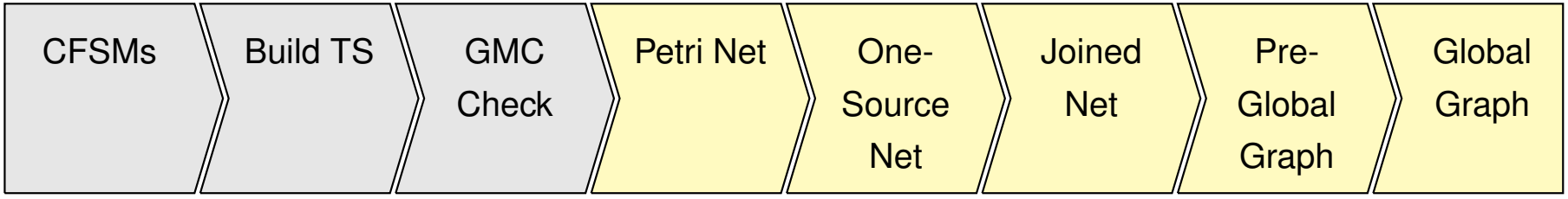
Last node reachable

Transformation Workflow

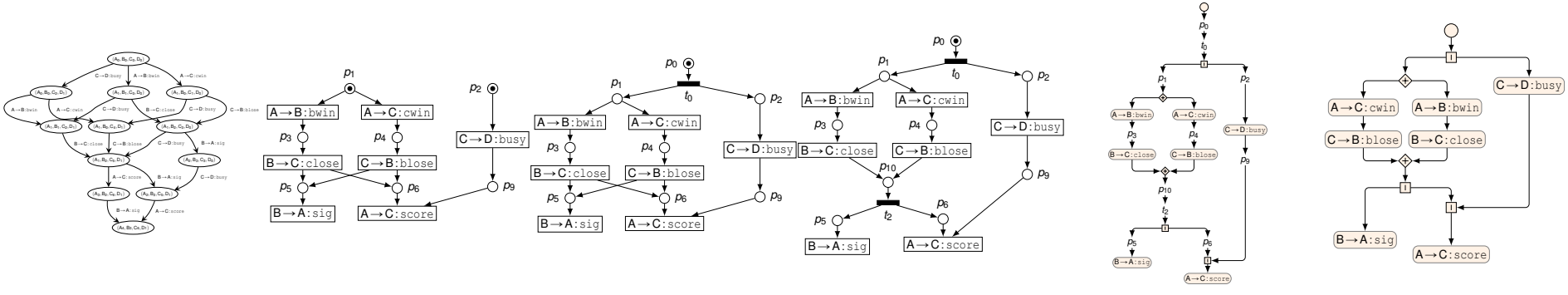


We use the work of Cortadella et al. (IEEE TC'98), based on the theory of regions, to synthesise a **safe** and **extended free-choice** Petri net from the Synchronous Transition System.

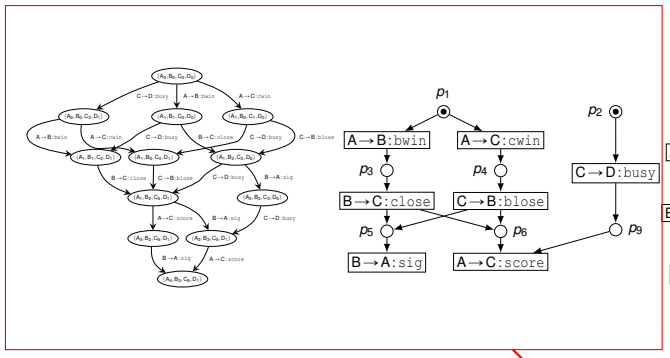
Transformation Workflow

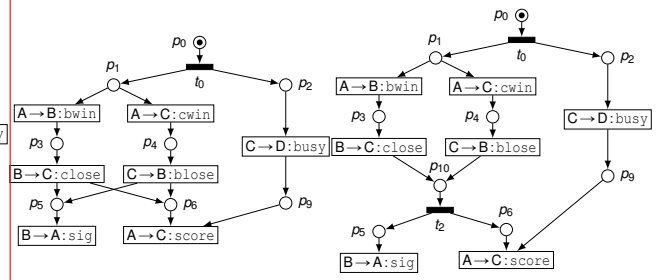


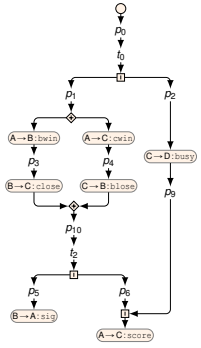
We use the work of Cortadella et al. (IEEE TC'98), based on the theory of regions, to synthesise a **safe** and **extended free-choice** Petri net from the Synchronous Transition System.

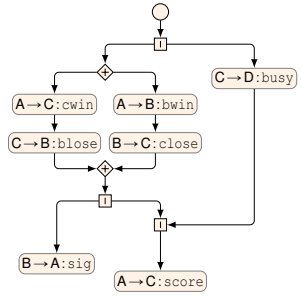


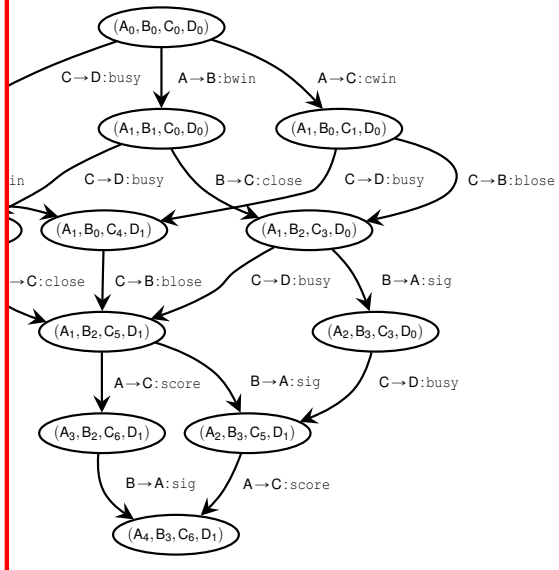
Step 1: TS ⇝ PN

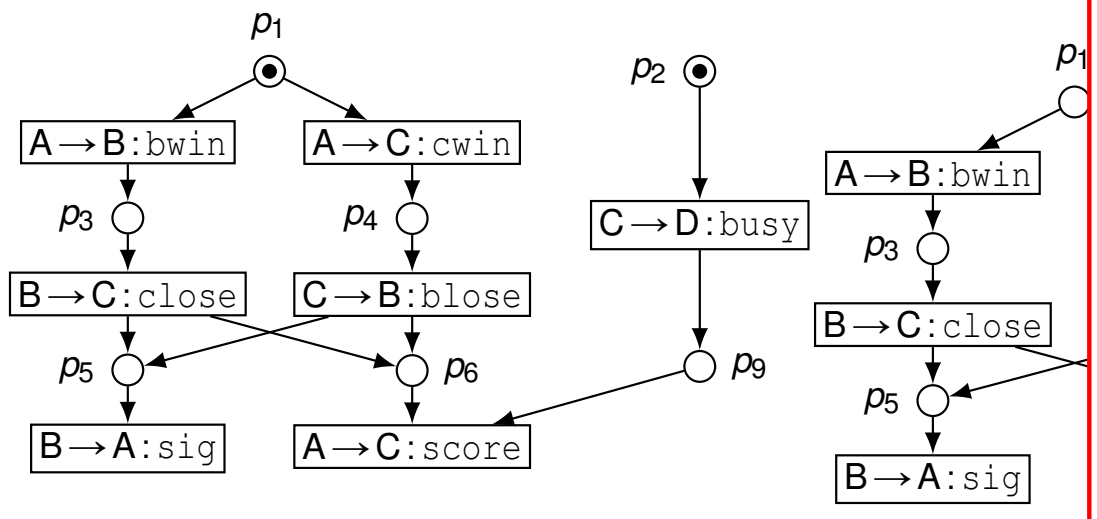






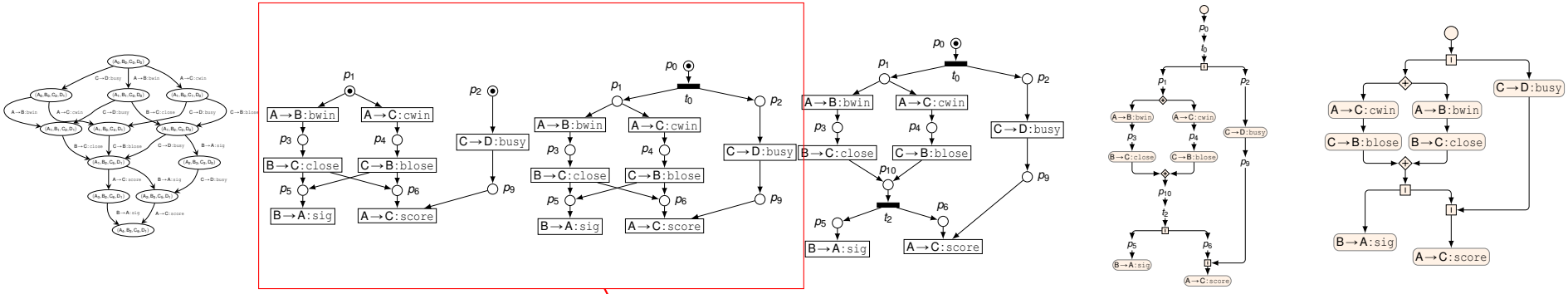


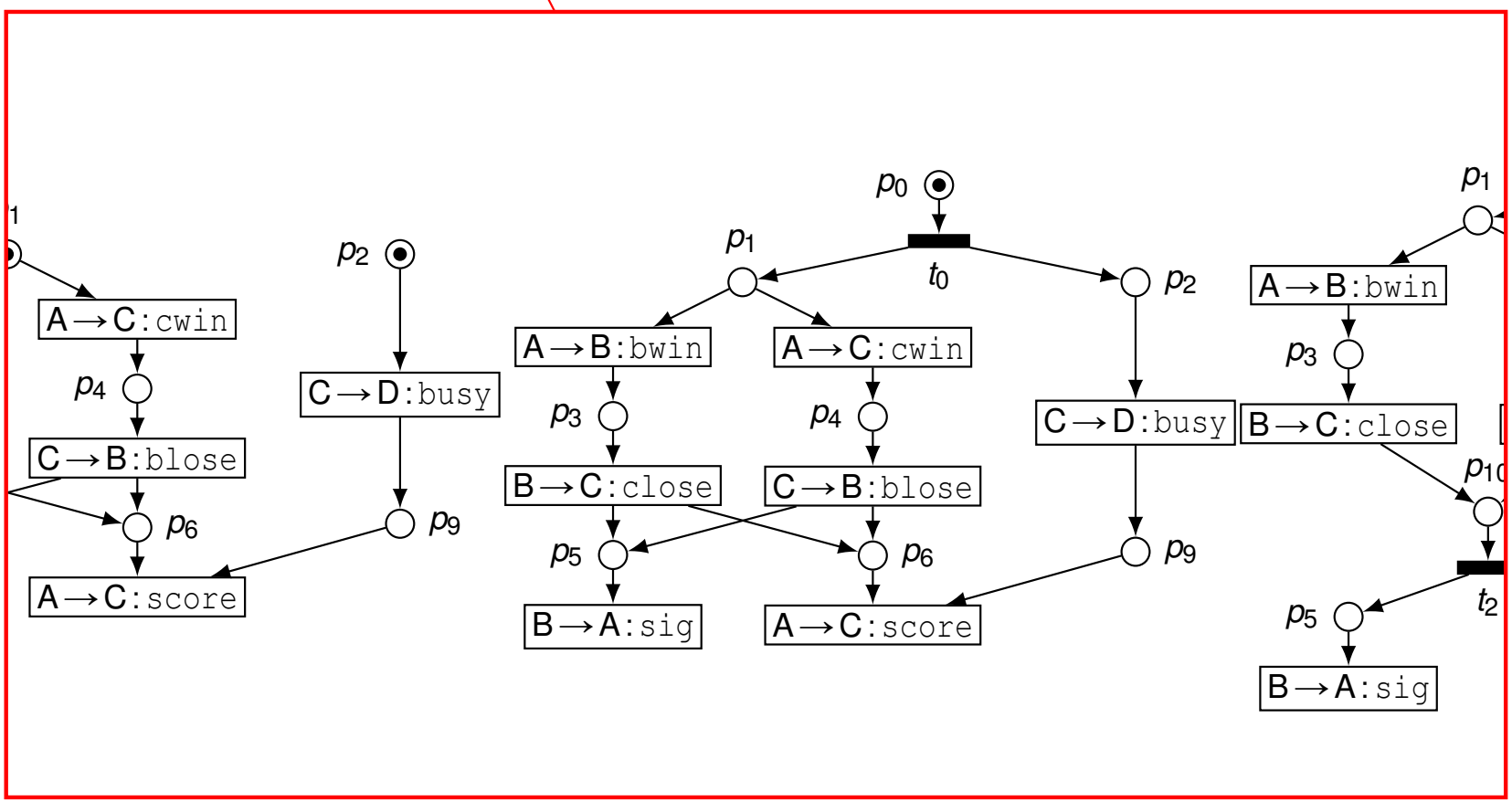




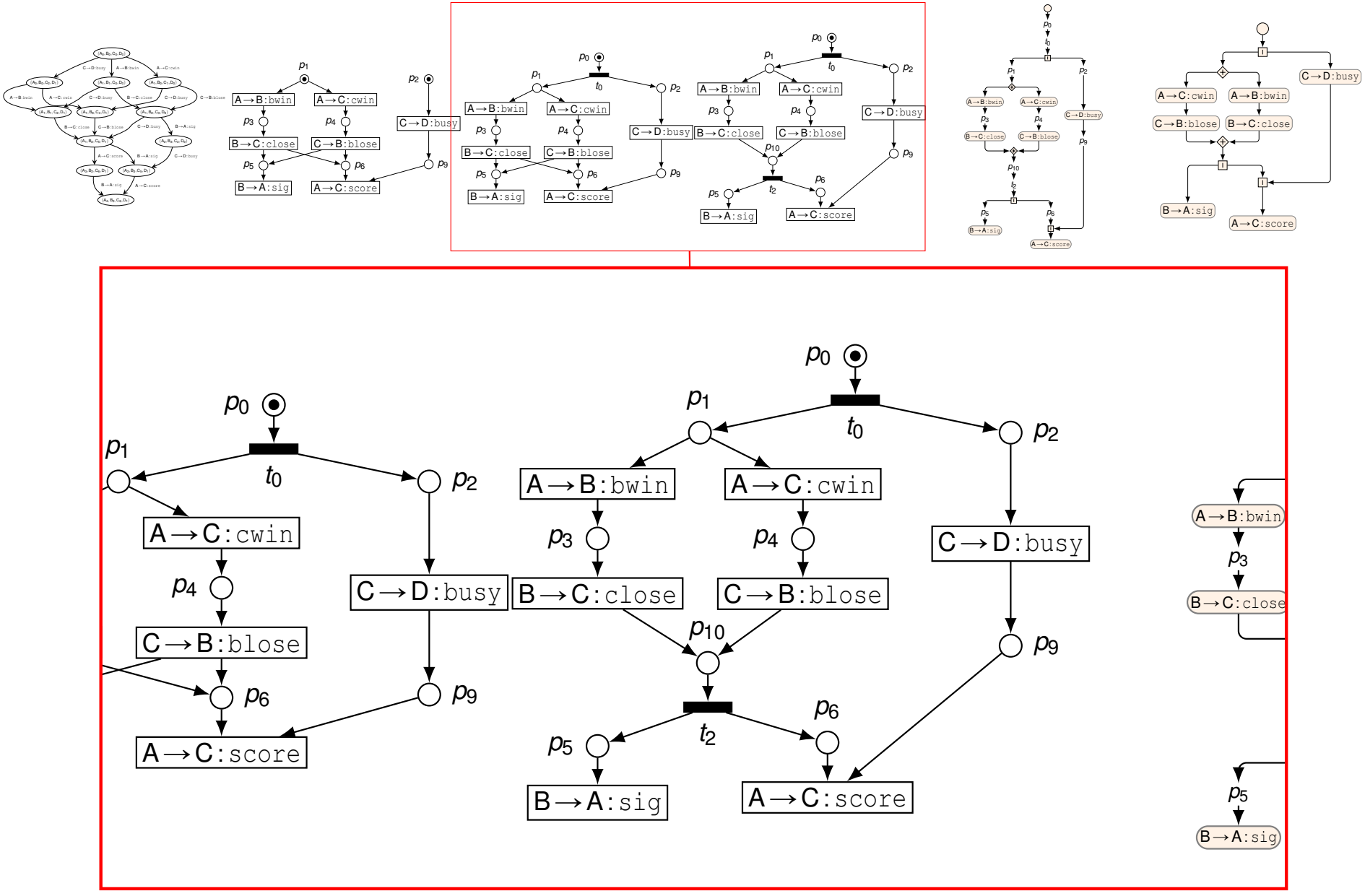
(cf. Algorithm of Cortadella et al.)

Step 2: PN ⇝ 1-source PN

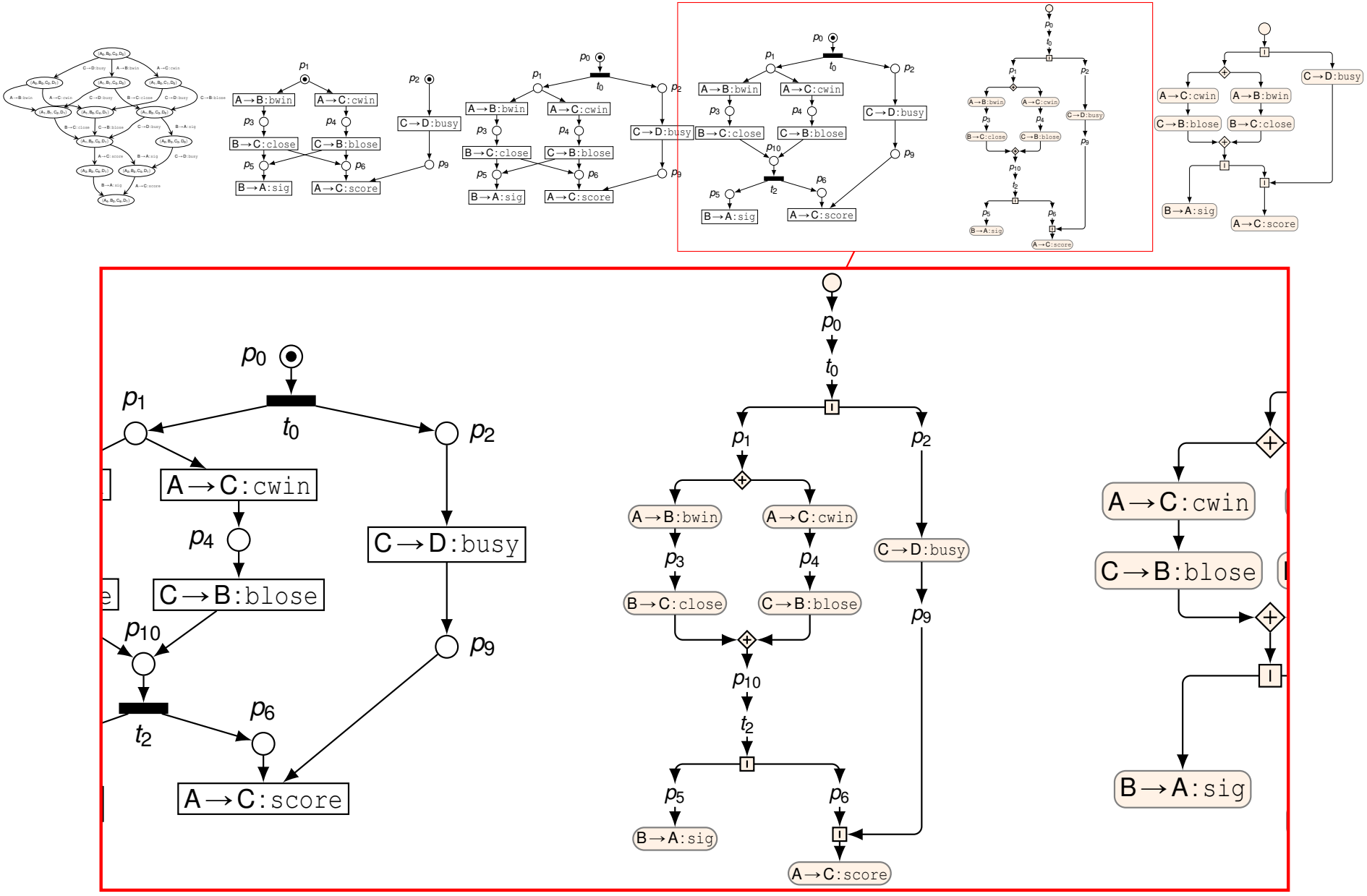




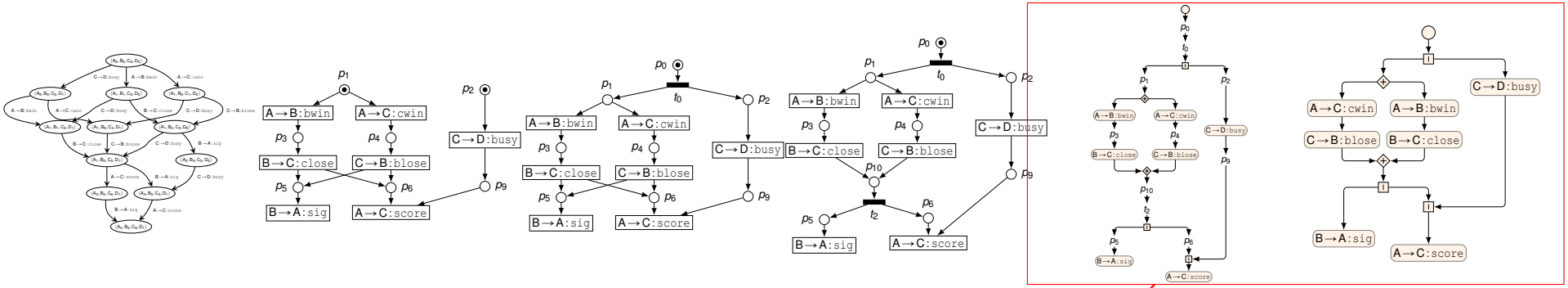
Step 3: 1-source PN ⇝ Joined PN

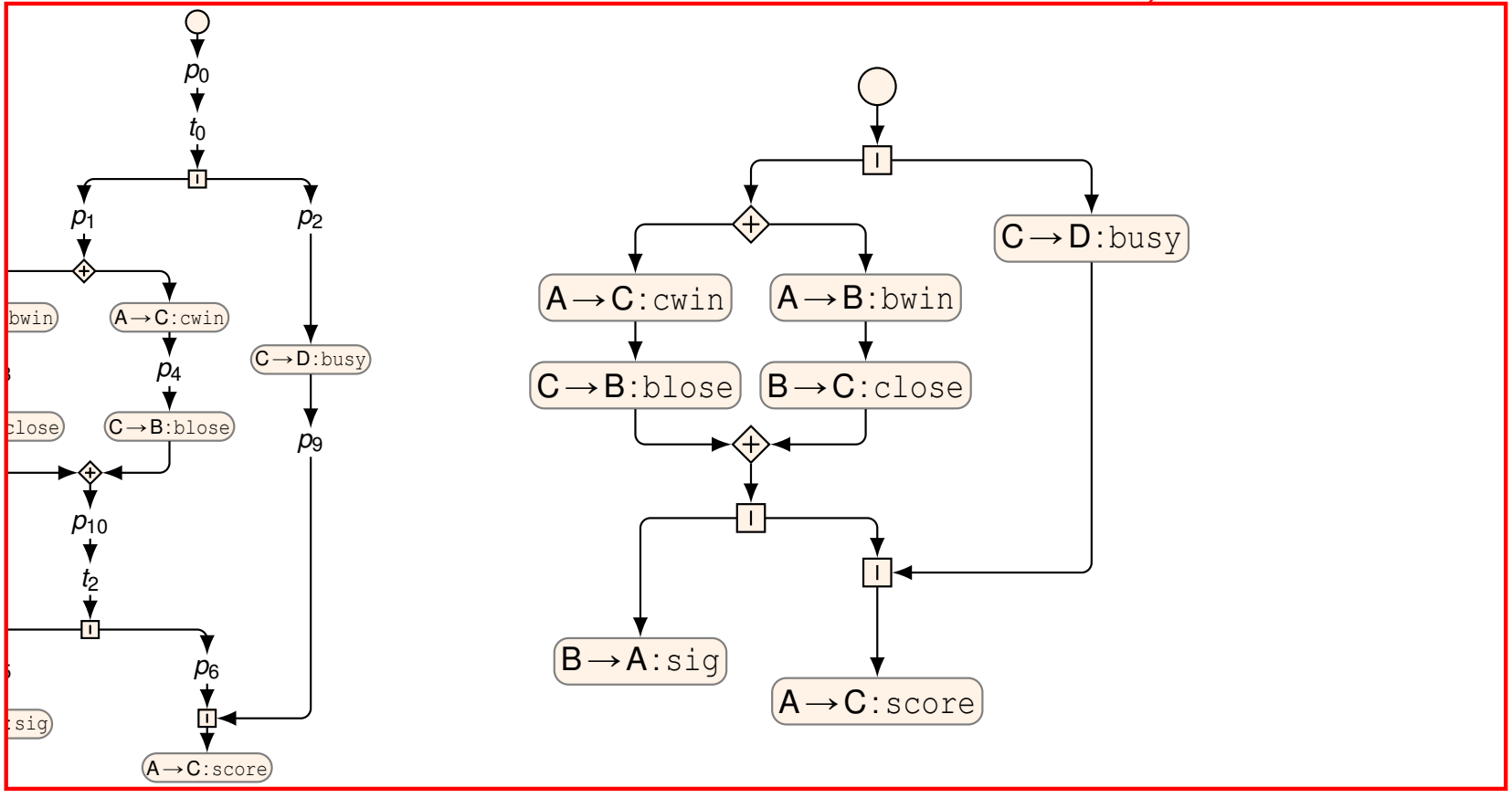


Step 4: Joined PN ⇝ Pre-Interaction Graph



Step 5: Pre-Interaction Graph ⇝ Interaction Graph





Distilling models

/emtalks/images/flow.png \protect \unhbox \voidb@x \penalty \@M

Why is this not so simple?

Decisions have to be taken for good abstractions:

on granularity

- every atomic thread is a participant: “too refined”...perhaps
- what about a participant when one of its threads “hiddenly” forks?
- 1 participant = several threads, usually

⊢ but a too coarse model could be not so good either

on state transitions

- state transition of machines depend on which are the participants
- states are determined by the invocations they make
- but some invocations are “internal”

⊢ care is necessary about what is observable

An application: the gen_server

From www.erlang.org

/emtalks/images/gen$_s$*erver*$_m$*an.png* \let =\let =\beamer@thcuse@ \bea

$\geqslant$11k Erlang's repos with $\geqslant$31k use of gen_server on github only

Step 1: Identifying participants on gen_server

For ping-pong

this step is trivial

The obvious

- let: Participants = Client (C) + Server (S)
  - $\perp$: From docs: C does not interact directly with S
- $\vdash$: Something from the API must be exposed...so
- let: Particants = C + Library (L)
  - $\perp$: This also fails: too coarse we tried
- $\vdash$ Participants = C + L + S
  (... may be; but let's use this working hypothesis)

Step 1: Identifying participants on gen_server

For ping-pong

this step is trivial

The obvious fails...obviously

- let: Participants = Client (C) + Server (S)
  - $\perp$: From docs: C does not interact directly with S
- $\vdash$: Something from the API must be exposed...so
- let: Particants = C + Library (L)
  - $\perp$: This also fails: too coarse we tried
- $\vdash$ Participants = C + L + S
  (... may be; but let's use this working hypothesis)

Step 1: Identifying participants on gen_server

For ping-pong

this step is trivial

The obvious fails...obviously

- let: Participants = Client (C) + Server (S)
  - $\perp$: From docs: C does not interact directly with S
- $\vdash$: Something from the API must be exposed...so
- let: Particants = C + Library (L)
  - $\perp$: This also fails: too coarse we tried
- $\vdash$ Participants = C + L + S
  (... may be; but let's use this working hypothesis)

Step 1: Identifying participants on gen_server

For ping-pong

this step is trivial

The obvious fails...obviously

- let: Participants = Client (C) + Server (S)
  - $\perp$: From docs: C does not interact directly with S
- $\vdash$: Something from the API must be exposed...so
- let: Particants = C + Library (L)
  - $\perp$: This also fails: too coarse we tried
- $\vdash$ Participants = C + L + S
  (... may be; but let's use this working hypothesis)

Step 1: Identifying participants on gen_server

For ping-pong

this step is trivial

The obvious fails...obviously

- let: Participants = Client (C) + Server (S)
  - $\perp$: From docs: C does not interact directly with S
- $\vdash$: Something from the API must be exposed...so
- let: Particants = C + Library (L)
  - $\perp$: This also fails: too coarse we tried
- $\vdash$ Participants = C + L + S
  (... may be; but let's use this working hypothesis)

Step 1: Identifying participants on gen_server

For ping-pong

this step is trivial

The obvious fails...obviously

- let: Participants = Client (C) + Server (S)
  - $\perp$: From docs: C does not interact directly with S
- $\vdash$: Something from the API must be exposed...so
- let: Particants = C + Library (L)
  - $\perp$: This also fails: too coarse we tried
- $\vdash$ Participants = C + L + S
  (... may be; but let's use this working hypothesis)

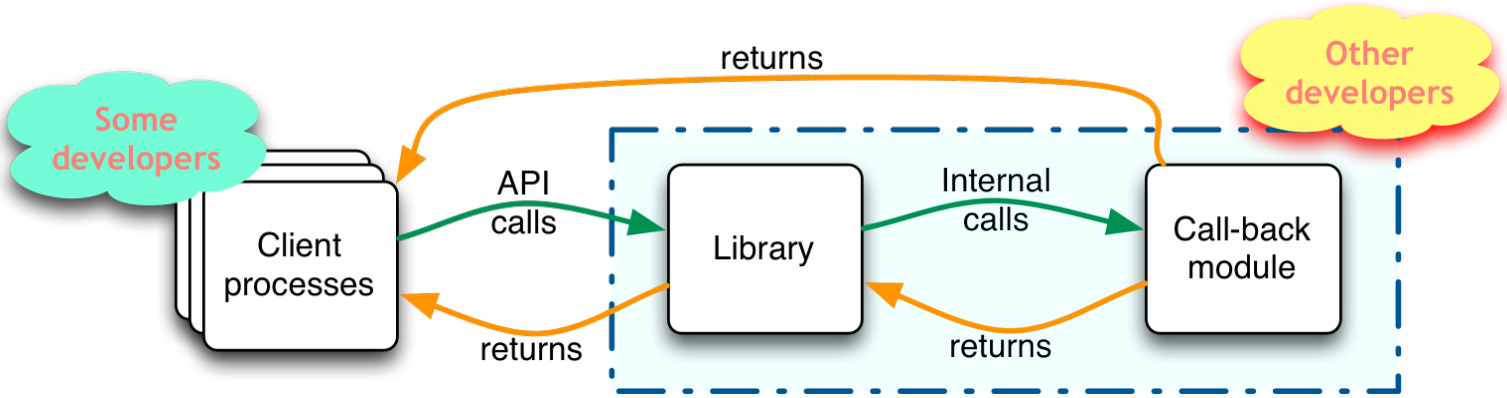
Step 2: Identifying events on gen_server

From gen_server API

Clients can

- start new instances of gen_server
- stop a gen_server
- request to handle C's calls

gen_server communication events



Step 3: extracting CFSMs on gen_server

Aren't they cute?

/emtalks/images/gen$_s$*erver*$_m$*achines.png* \let =\let =\beamer@thcuse@