

# Process Composition



# Process Composition Hierarchies

---

- Control:
  - Centralized control: e.g., RPC/WSDL
  - Mediated or orchestrated control: e.g., BPEL
  - Fully distributed control: autonomous processes, e.g., WS-CDL (WSCI)
- Messaging:
  - Shared variables: tight coupling, service internal
  - Synchronous messaging: looser coupling; some service-service interactions
  - Asynchronous messaging: loose coupling; service-service interactions
- Synchronous and asynchronous both applicable to BPEL:
  - WSDL/SOAP, WSDL/JMS

# Communicating Sequential Processes

---

- Mathematical framework for the description and analysis of systems consisting of processes interacting via exchange of messages
- The evolution of processes is based on a sequence of **events** or **actions**
  - **Visible actions**  $\Sigma$
- Interaction with other processes, communication
  - **Invisible action**  $\tau$
- Internal computation steps

# The CSP Language: Syntax

---

- Termination: **Stop**
- Input: **in ?  $x \rightarrow P(x)$** 
  - Execute an input action on channel **in**, get message  $x$ , then continue as  $P(x)$
- Output: **out !  $x \rightarrow P(x)$** 
  - Execute an output action on channel **out**, send message  $x$ , then continue as  $P(x)$
- Recursion:  **$P(y_1, \dots, y_n) = Body(y_1, \dots, y_n)$** 
  - **Process definition.**  $P$  is a process name,  $y_1, \dots, y_n$  are the parameters,  $Body(y_1, \dots, y_n)$  is a process expression
- Example:  **$Copy = in ? x \rightarrow out ! m \rightarrow Copy$**

## CSP's Syntax (Cont'ed)

---

- External (aka guarded) choice:  $P \mid Q$
- Execute a choice between  $P$  and  $Q$ . Do not choose a process which cannot proceed
- Example:  $(a ? x \rightarrow P(x)) \mid (b ? x \rightarrow Q(x))$
- Execute one and only one input action. If only one is available then choose that one. If both are available than choose arbitrarily. If none are available then block. The unchosen branch is discarded
- Internal choice:  $P + Q$
- Execute an arbitrary choice between  $P$  and  $Q$ . It is possible to choose a process which cannot proceed

## CSP's Syntax (Cont'ed)

---

- Parallel operator w/synchronization:  $P \parallel Q$ 
  - $P$  and  $Q$  proceed in parallel and are obliged to synchronize on all the common actions
- Example:  $(c ? x \rightarrow P(x)) \parallel (c ! m \rightarrow Q)$
- Synchronization: the two processes can proceed only if their actions correspond
- Handshaking: sending and receiving is simultaneous (Buffered communication can anyway be modeled by implementing a buffer process)
- Communication:  $m$  is transmitted to the first process, which continues as  $P(m)$

# Laws of CSP

---

- Equivalence of expressions: useful in reasoning about CSP processes
- Many laws concerning different aspects
- Examples:
  - If  $Q$  does not involve  $e_1$ :  
$$(e_1 \rightarrow P) \parallel (e_2 \rightarrow Q)$$
$$= e_1 \rightarrow (P \parallel (e_2 \rightarrow Q))$$
  - Synchronization:  
$$(c ? b \rightarrow P(b)) \parallel (c ! a \rightarrow Q)$$
$$= c ! a \rightarrow (P(a) \parallel Q)$$

# Example

---

- ADD2 = ( $in_1 ?x \rightarrow in_2 ?y \rightarrow out !(x+y) \rightarrow ADD2$ )
- ADD2 = ( $in_1 ?x \rightarrow in_2 ?y \rightarrow out !(x+y) \rightarrow ADD2$ ) |  
    ( $in_2 ?y \rightarrow in_1 ?x \rightarrow out !(x+y) \rightarrow ADD2$ )
- SQ = ( $out ?z \rightarrow sqr !square(z) \rightarrow SQ$ )
- ADDSQ = ADD2 || SQ = ?  
  
= ( $in_1 ?x \rightarrow in_2 ?y \rightarrow out !(x+y) \rightarrow ADD2$ ) |  
    ( $in_2 ?y \rightarrow in_1 ?x \rightarrow out !(x+y) \rightarrow ADD2$ )  
|| ( $out ?z \rightarrow sqr !square(z) \rightarrow SQ$ )
- = ( $in_1 ?x \rightarrow in_2 ?y \rightarrow out !(x+y) \rightarrow ADD2$ )  
|| ( $out ?z \rightarrow sqr !square(z) \rightarrow SQ$ )  
|| ( $in_2 ?y \rightarrow in_1 ?x \rightarrow out !(x+y) \rightarrow ADD2$ )  
|| ( $out ?z \rightarrow sqr !square(z) \rightarrow SQ$ )

# Example

## ■ ADDSQ

```
= (in1?x → in2?y → out !(x+y) → ADD2)
  || (out ?z → sqr !square(z) → SQ)
  | (in2?y → in1?x → out !(x+y) → ADD2)
    || (out ?z → sqr !square(z) → SQ)
    = in1?x → in2?y → (out !(x+y) → ADD2)
      || (out ?z → sqr !square(z) → SQ)
      | in2?y → in1?x → (out !(x+y) → ADD2)
        || (out ?z → sqr !square(z) → SQ)
        = in1?x → in2?y → out !(x+y) → ADD2 || SQ
          |
          ...
          = in1?x → in2?y → out !(x+y) → sqr !square(x+y) → ADDSQ
            |
            ...
```

# BPEL : CSP Semantics

---

- BPEL interactions can be modeled in CSP
  - Three main activities: invoke, receive, reply

```
<invoke partner="..." portType="..." operation="..."  
      inputContainer=x outputContainer = y />
```

partner-port-in ! **x** → partner-port-out ? **y** → ...

```
<receive partner="..." portType="..." operation="..."  
      container=x [createInstance="..."] />
```

my-port-in ? **x** → ...

```
<reply partner="..." portType="..." operation="..."  
      container =y />
```

... my-port-out ? **y** → ...

# BPEL : CSP Semantics

---

- Synchronous communication
- Advantages:
  - Easier to analyze
  - Well studied
- Disadvantages:
  - Not completely autonomous

# The $\pi$ -Calculus

---

- The  $\pi$ -calculus is a process algebra
- Constructs for concurrency
- Communication on channels
  - channels are first-class
    - channel names can be sent on channels
  - access restrictions for channels
- In  $\pi$ -calculus everything is a process

# Communications in $\pi$ -Calculus

---

- Processes communicate on channels:

- $c < M >$  send message  $M$  on channel  $c$
- $c(x)$  receives  $x$  on channel  $c$

- Sequencing:

- $c < M >.p$  sends message  $M$  on  $c$ , then does  $p$
- $c(x).p$  receives  $x$  on  $c$ , then does  $p$  with  $x$

- Concurrency:

- $p \mid q$  is the parallel composition of  $p$  and  $q$

- Replication:

- $!p$  creates an infinite number of replicas of  $p$

# Examples

---

- For example we might define

*Speaker* =  $air < M >$

*Phone* =  $air(x).wire < x >$

*ATT* =  $wire(x).fiber < x >$

*System* = *Speaker* | *Phone* | *ATT*

- Communication between processes is modeled by
  - reduction:

*Speaker* | *Phone*  $\rightarrow$  *wire* < *M* >

*wire* < *M* > | *ATT*  $\rightarrow$  *fiber* < *M* >

- Composing these reductions we get:

*Speaker* | *Phone* | *ATT*  $\rightarrow$  *fiber* < *M* >

# Channel Visibility

---

- Anybody can monitor an unrestricted channel:
- Consider that we define
  - Copies the messages from the wire to NSA
  - Possible since the name "wire" is globally visible

$\text{WireTap} = \text{wire}(x).\text{wire}\langle x \rangle.\text{NSA}\langle x \rangle$

- Now
  - $\text{WireTap} \mid \text{wire}\langle M \rangle \mid \text{ATT} \rightarrow \text{wire}\langle M \rangle.\text{NSA}\langle M \rangle \mid \text{ATT} \rightarrow \text{NSA}\langle M \rangle \mid \text{fiber}\langle M \rangle$

# Restriction

---

- The restriction operator " $(\nu c) p'$ " makes a fresh channel  $c$  within process  $p$ 
  - $\nu$  is the Greek letter "nu"
  - The name " $c$ " is local (bound) in  $p$
- Restricted channels cannot be monitored
  - $wire(x) \dots | (\nu wire) (wire < M > \mid ATT) \rightarrow$
  - $wire(x) \dots | fiber < M >$
- The scope of the name "wire" is restricted
- There is no conflict with the global "wire"

# Restriction and Scope

---

## ■ Restriction

- is a binding construct
- is lexically scoped
- allocates a new object (a channel)

$(\nu c) p$     is like    "let  $c = \text{new Channel()}$  in  $p$ "

## ■ In particular, $c$ can be sent outside its scope

- But only if " $p$ " decides so

## First-Class Channels

---

- A channel  $c$  can leave its scope of declaration
  - via a message  $d_{<c>}$  from within  $p$
- Allowing channels to be sent as messages means communication topology is dynamic
  - If channels are not sent as messages (or stored in the heap) then the communication topology is static
  - This differentiates  $\pi$ -calculus from CSP

# Example of First-Class Channels

---

Consider:

$MobilePhone = air(x).cell < x >$

$ATT_1 = wire < cell >$

$ATT_2 = wire(y).y(x).fiber(x)$

in

$(v\ cell)\ (MobilePhone \mid ATT_1) \mid ATT_2$

- $ATT_1$  is trying to pass **cell** out of the static scope of the **restriction v cell**

# Scope Extrusion

---

- A channel is a name
  - First-class names must be usable even outside their original scope
- The  $\pi$ -calculus allows restrictions to move:
$$((v\ c)\ p) \mid q = (v\ c)(p \mid q) \quad \text{if } c \text{ not free in } q$$
- Renaming is needed in general:
$$((v\ c)\ p) \mid q = ((v\ d)\ [d/c]\ p) \mid q = (v\ d)\ ([d/c]\ p \mid q)$$
where "d" is fresh (does not appear in p or q)

## Example, Continued

---

$(\nu \text{ cell})(\text{ MobilePhone} \mid \text{ATT}_1) \mid \text{ATT}_2$

$= (\nu \text{ cell})(\text{ MobilePhone} \mid \text{ATT}_1) \mid \text{ATT}_2$

$\rightarrow (\nu \text{ cell})(\text{ MobilePhone} \mid \text{cell}(x).\text{fiber}\langle x \rangle)$

- Scope extrusion distinguishes the  $\pi$ -calculus from other process calculi

# Syntax of the $\pi$ -Calculus

---

- There are many versions of the  $\pi$ -calculus

A basic version:

$p, q ::=$

$nil$

$x < y >. p$

$x(y). p$

$p \parallel q$

$!p$

$(\nu x) p$

nil process (sometimes written 0)

sending

receiving

parallel composition

replication

restriction

- Note that only variables can be channels and messages

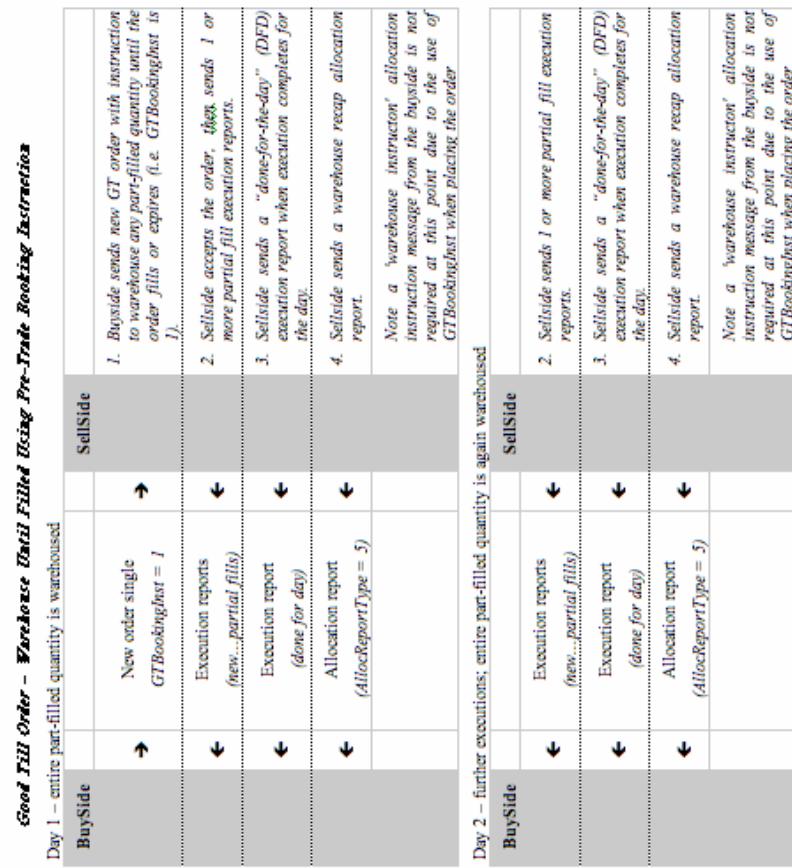
# Choreography Definition Language (WS-CDL)

---

- Global model
  - Ensured conformance
- Description language
  - Not executable
- Tools
  - Generators for end points
  - Advanced typing
- Status
  - Moving for last call end of 2004

# Global Models

Example flow for Pre-Trade Allocation (using Allocation Instruction message)

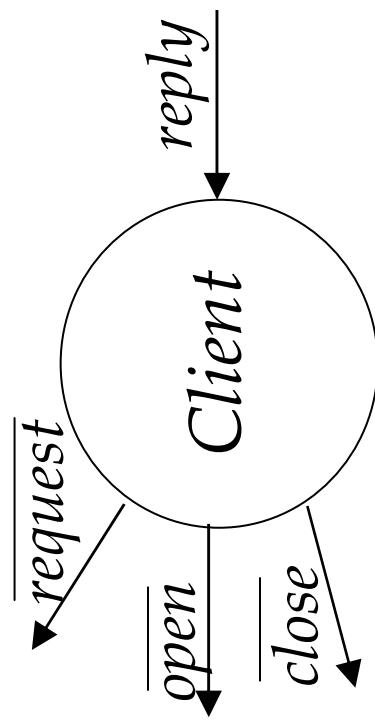


# WS-CDL Global Models

---

- A sequential process

$$\text{Client}(\text{open}, \text{close}, \text{request}, \text{reply}) = \overline{\text{open}.\text{request}_1.\text{reply}_1} \cdot \overline{\text{request}_2.\text{open}} \cdot \overline{\text{request}_2.\text{reply}_2} \cdot \overline{\text{close}.0}$$

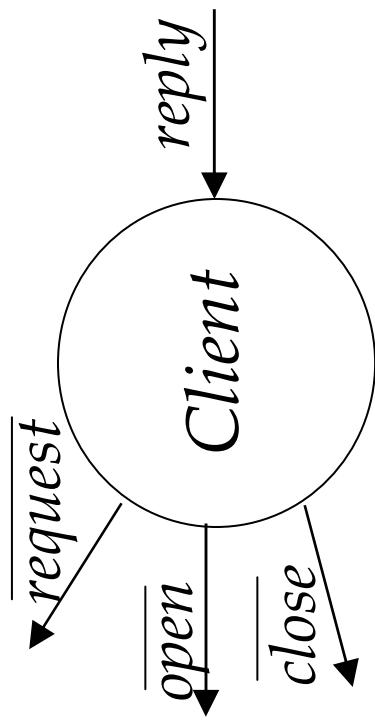


# WS-CDL Global Models

---

- A repetitive process

$\text{Client}(\text{open}, \text{close}, \text{request}, \text{reply}) =$   
 $\overline{\text{open}}.\overline{\text{request}_1.\text{reply}_1}.\overline{\text{request}_2.\text{reply}_2}.\overline{\text{close}}.\text{Client}(\text{open},$   
 $\text{close}, \text{request}, \text{reply})$



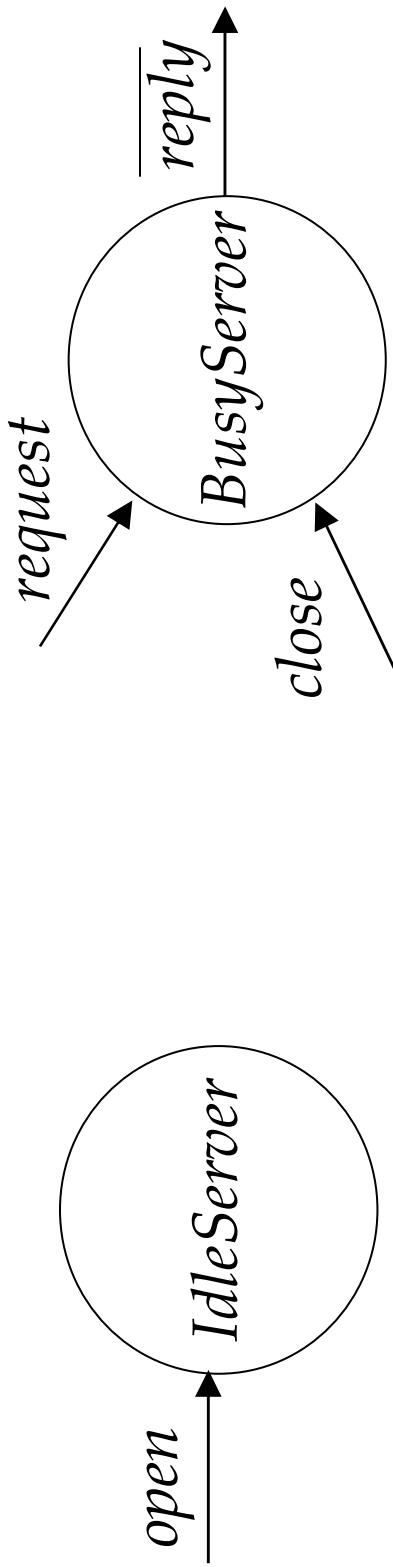
# WS-CDL Global Models

---

- A process with choices to make

```
IdleServer (o, req, rep, c) =  
o.BusyServer(o, req, rep, close)
```

```
BusyServer(o, req, rep, c) =  
req.rep.BusyServer(o, req, rep, c) +  
c.IdleServer(o, req, rep, c)
```



# WS-CDL Global Model

## ■ Communication, Concurrency and Replication

$$\text{SYSTEM} = (!\text{Client} \mid \text{IdleServer})$$

- When  $\text{Client}_i$  has started an exchange with  $\text{IdleServer}$
  - No other  $\text{Client}_j$  can then communicate with the server
  - Until  $\text{Client}_i$  has finished and the server is once again  $\text{IdleServer}$
- |  |  |  |  |       |
|--|--|--|--|-------|
| $\text{Client}_i \mid \text{IdleServer}$ | $\text{Client}_i \mid \text{BusyServer}$ | $\text{Client}_j \mid \text{IdleServer}$ | $\text{Client}_j \mid \text{BusyServer}$ | ..... |
|--|--|--|--|-------|

# WS-CDL and the $\pi$ -Calculus

Collapse send and receive  
into an  
interact on channels

Operation	Notation	Meaning
Prefix	$\pi.p$	Sequence
Action	$a(y), a < y >$	Communication
Summation	$a(y).p + b(x).q$ $\sum \pi_i.p_i$	Choice
Recursion	$p = \{ \dots \} . p$	Repetition
Replication	$!p$	Repetition
Composition	$p \mid q$	Concurrency
Restriction	$(\nu x) p$	Encapsulation

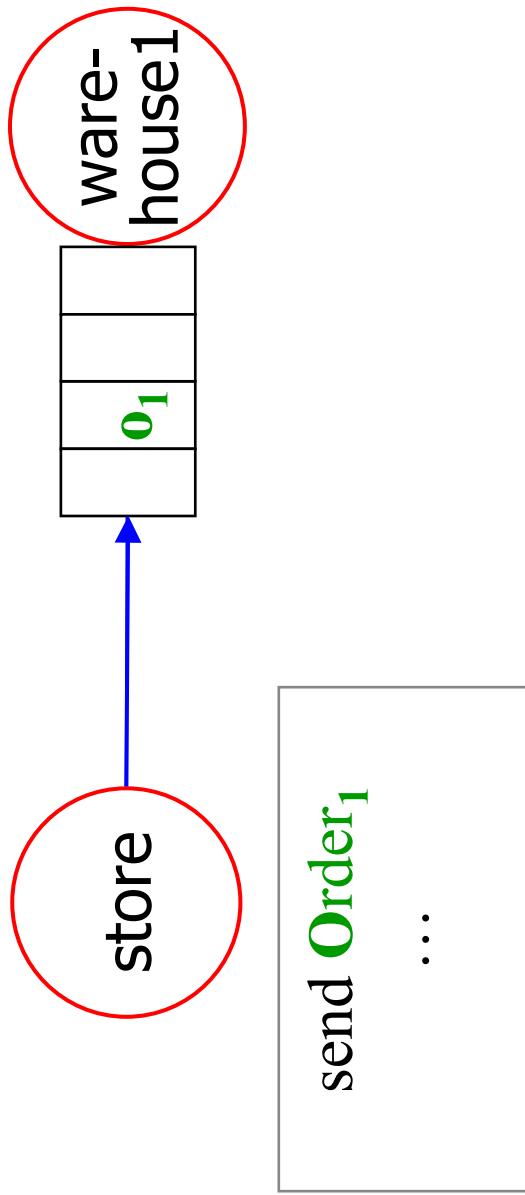
# Composition: Formal Semantics

---

- WSDL is fundamentally message based
  - So is everything based on it
- Process algebra approach to formal semantics:
  - BPEL : CSP
  - WS-CDL :  $\pi$ -calculus
- Alternatives to process algebra:
  - Automata theoretic
    - PSL

# BPEL and Asynchronous Communication

- Channels are assumed to be reliable
- **Asynchronous**, for example, the following channel:



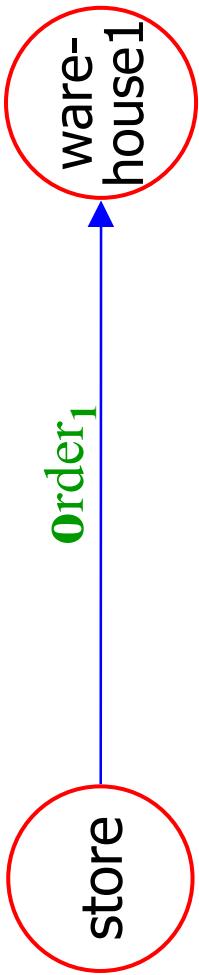
- Queues are **FIFO, unbounded length**
- Can simulate synchronous and also bounded queues

send Order<sub>1</sub>  
receive Receipt<sub>1</sub>  
...

# Messages

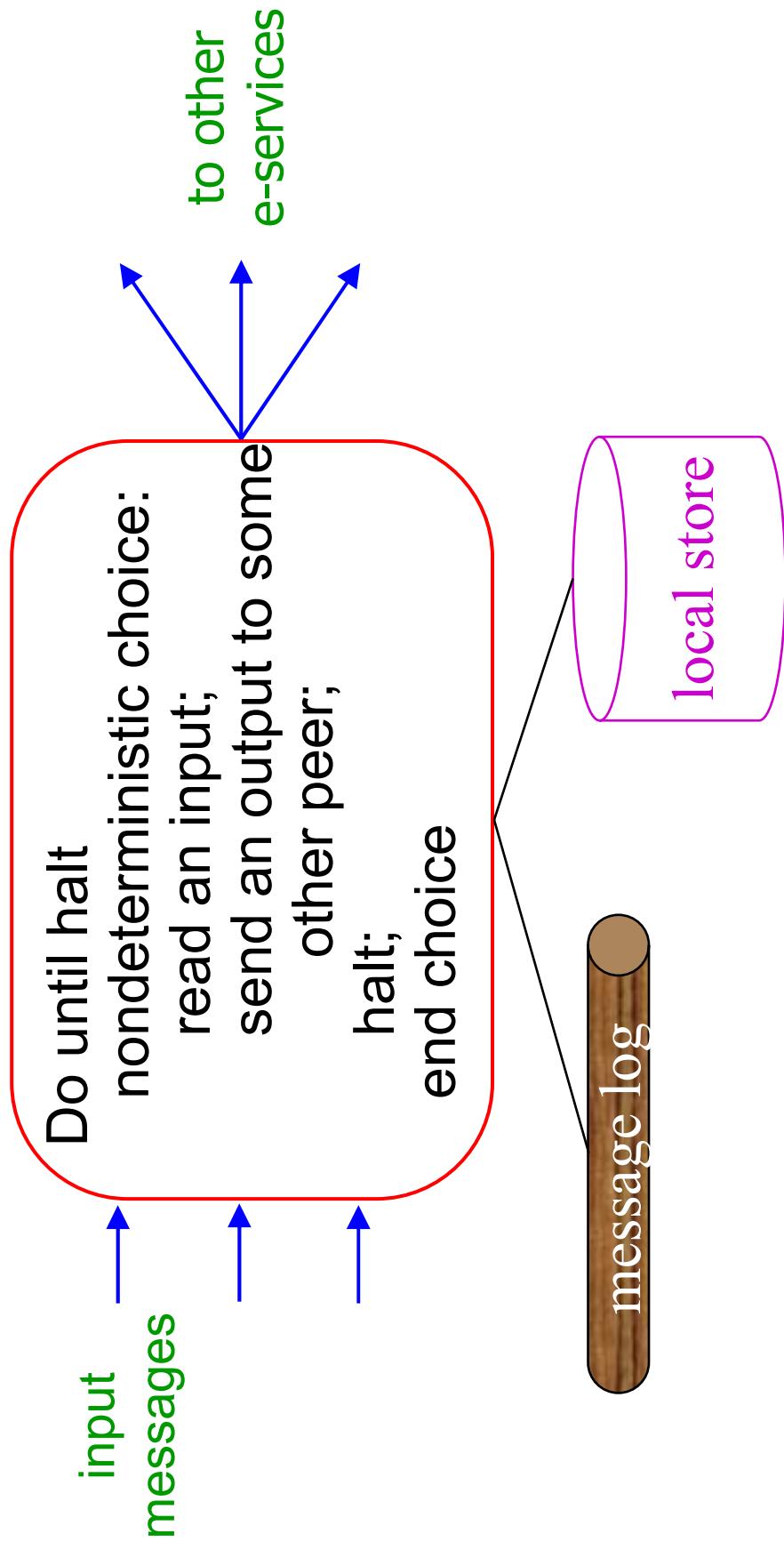
---

- Messages are classified into **classes**
- Each class is associated with **one channel**

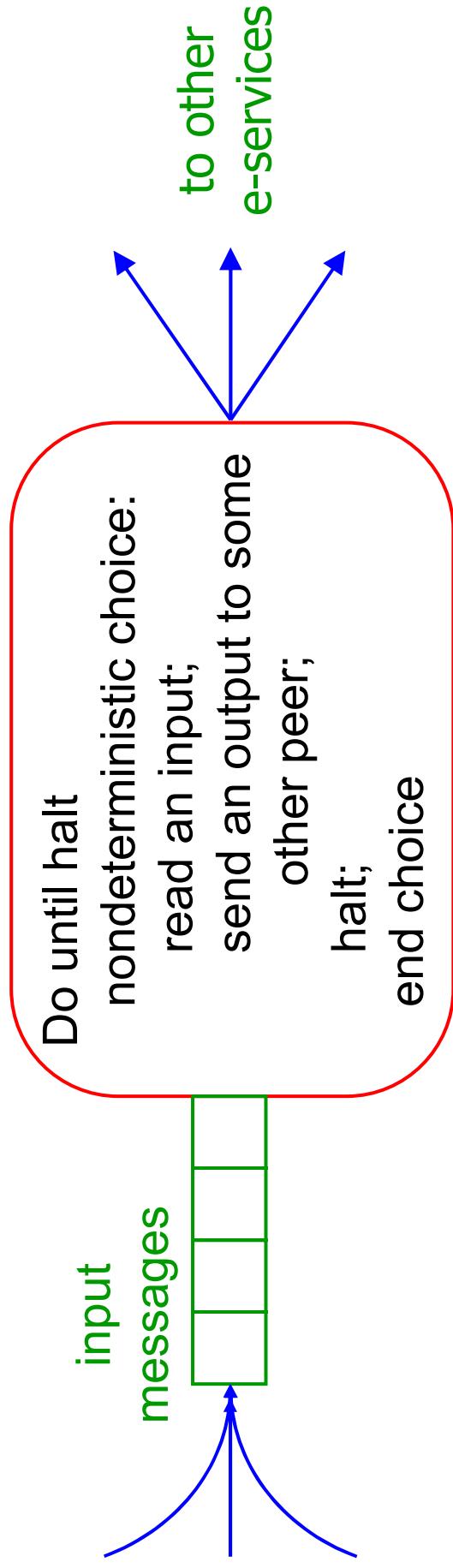


- Each message class may have **additional attributes** which can carry the contents of messages
  - For now, analysis involves no contents
  - Results immediately apply to "finite domain" contents

# Individual Web Services



# Individual Web Services

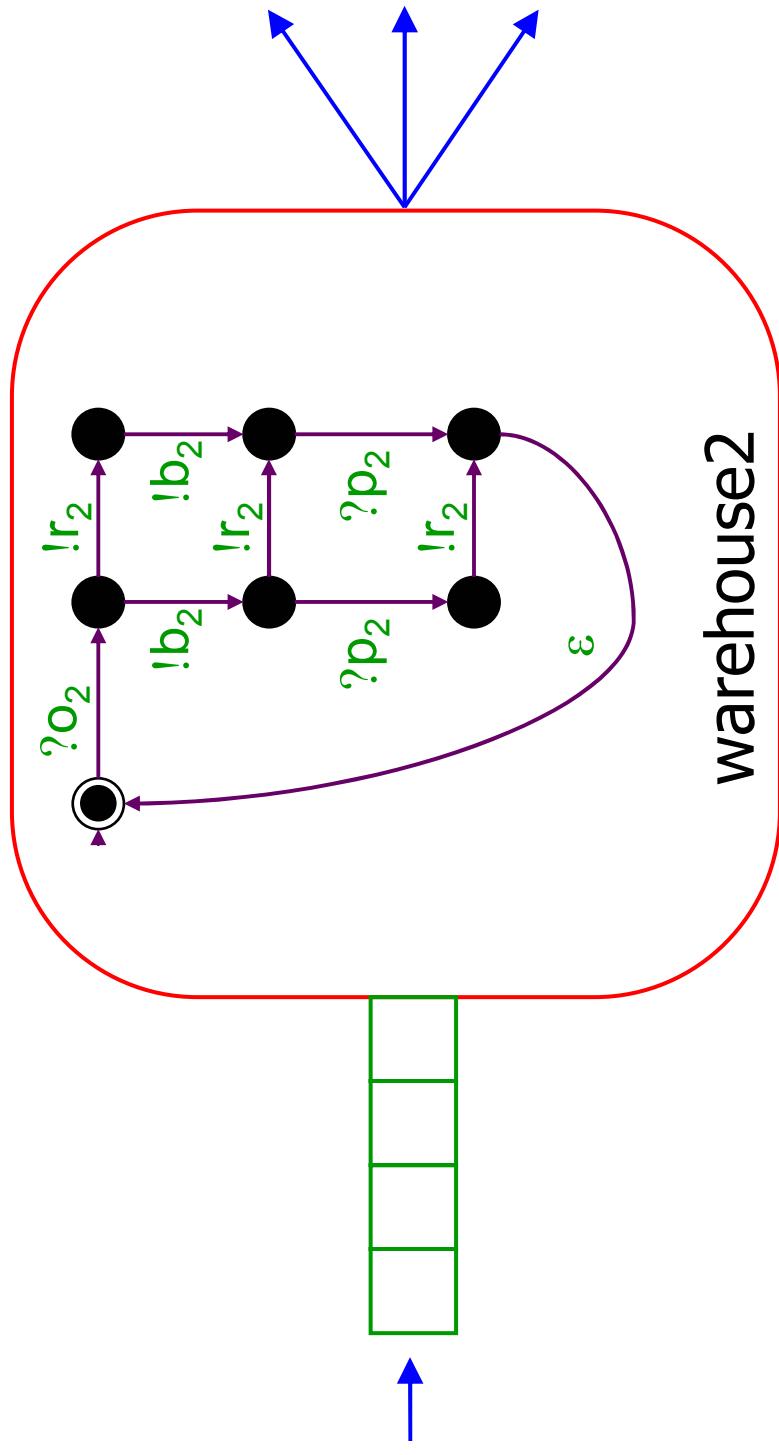


- Again, ports and storages are ignored
- Internal logic of peers : finite state control

# Mealy Web Services

---

- Mealy machines: Finite state machines with input (incoming messages) & output (outgoing messages)



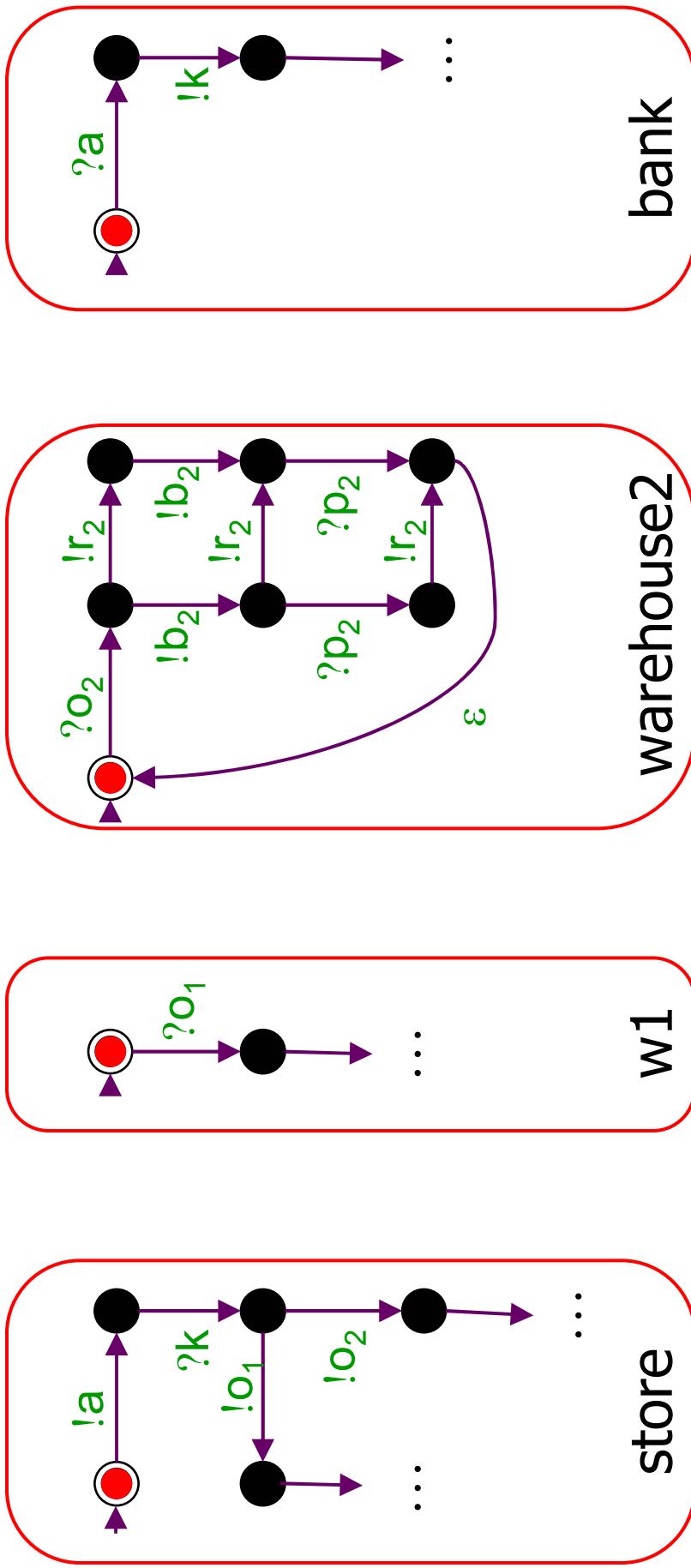
# Technical Definition

---

A **Mealy web service** is an FSA  $M = (T, s, F, \Sigma^{\text{in}}, \Sigma^{\text{out}}, \delta)$

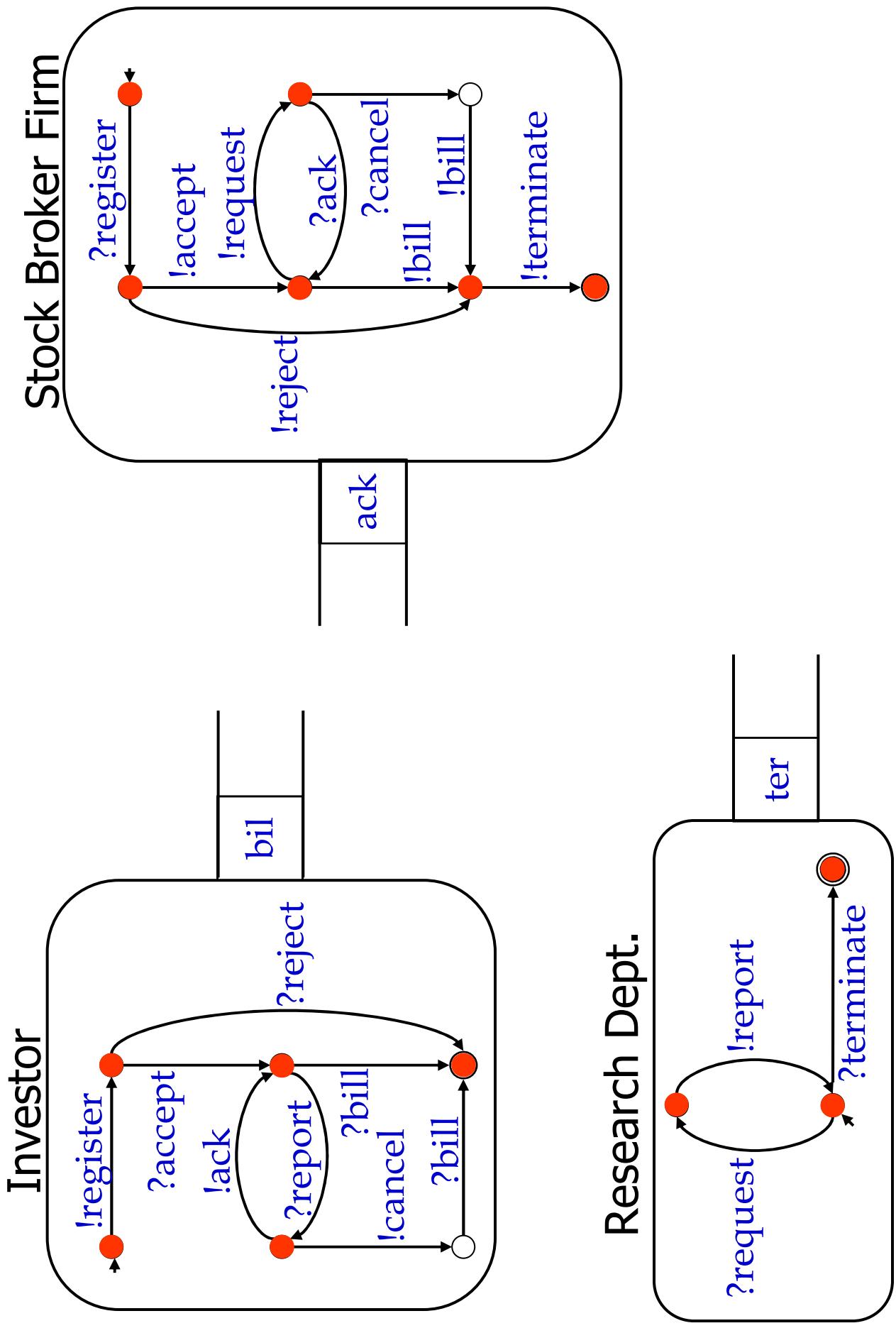
- $T$  : a set of states
- $s$  : the initial state
- $F$  : a set of final states
- $\Sigma^{\text{in}}$  : input message classes
- $\Sigma^{\text{out}}$  : output message classes
- $\delta$  : transition relation that either
  - **consume** an input,  $(s_1, ?m, s_2)$ , or
  - **produce** output,  $(s_1, !m, s_2)$ , or
  - make an empty (**internal**) move,  $(s_1, \varepsilon, s_2)$

# Executing a Mealy Composition

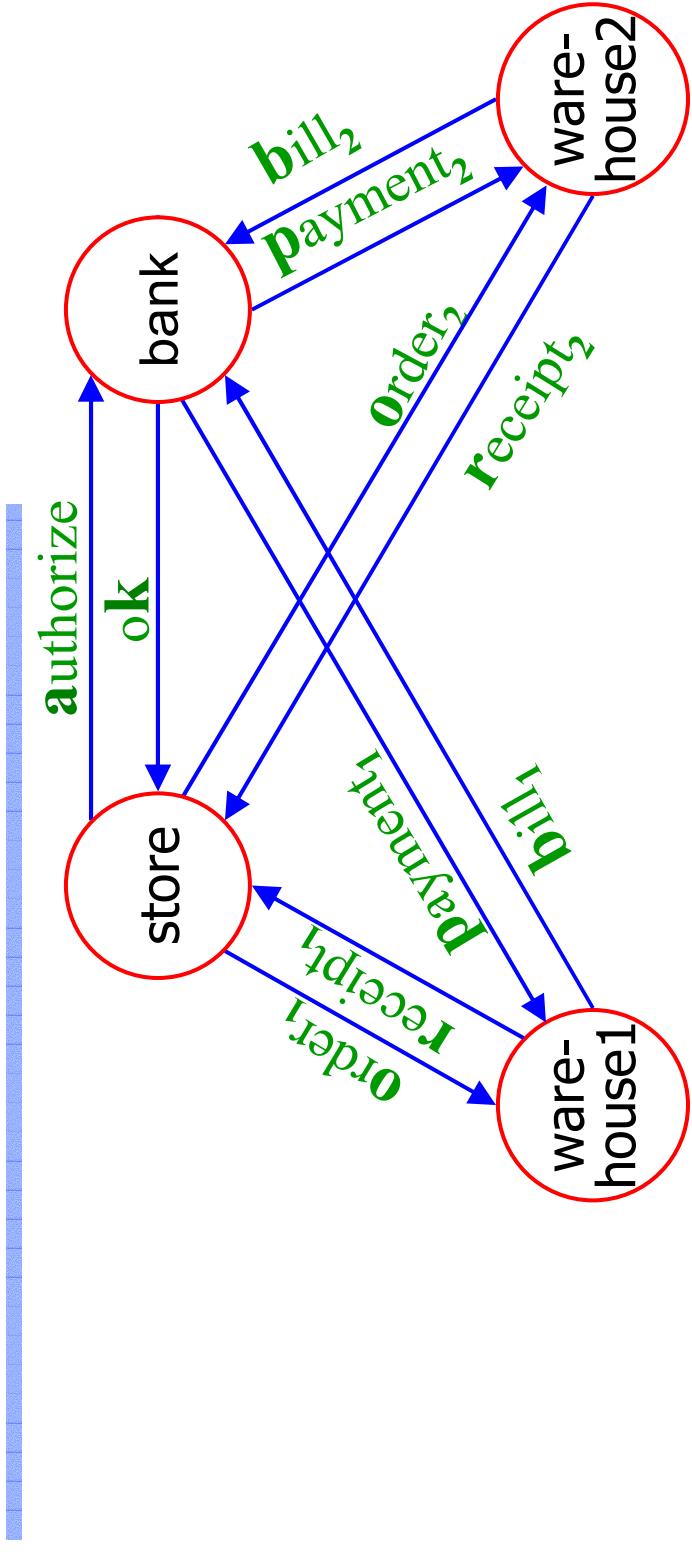


- Execution halts if
  - All mealy peers are in final states
  - All queues are empty

# Composite Web Service Execution



# Web Service Composition: What Now?



## ■ Execution:

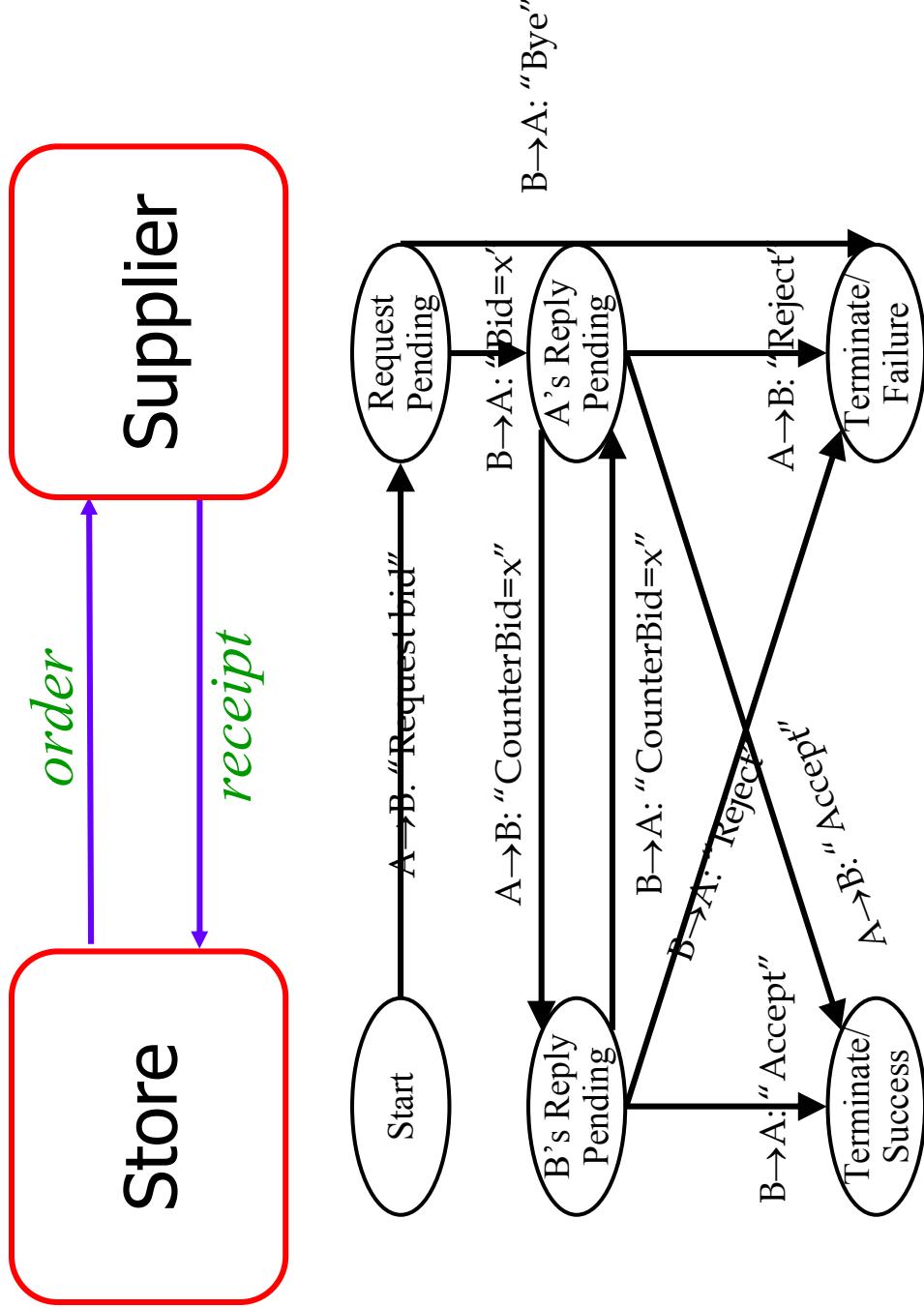
- Is there a deadlock?
- Always terminates in finite steps?
- ...

## ■ Functionality:

- Is it correct?
- Is there an unauthorized payment?
- ...

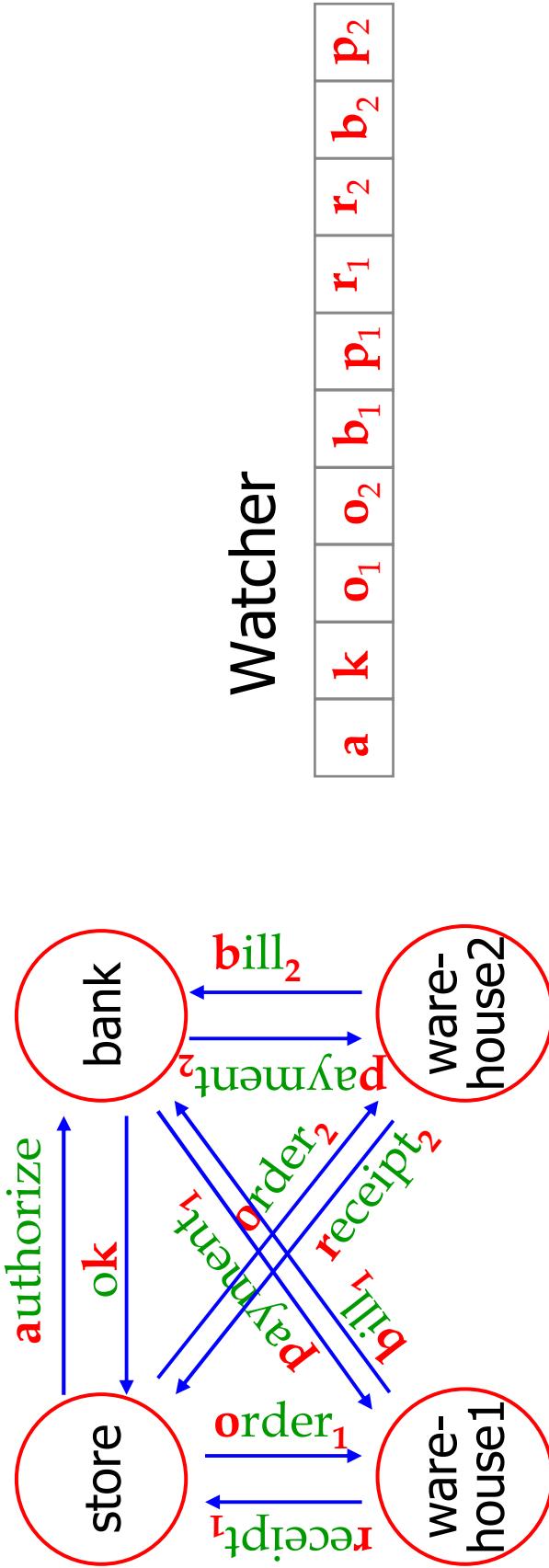
# Conversation Policies

- A conversation: a sequence of messages between two parties



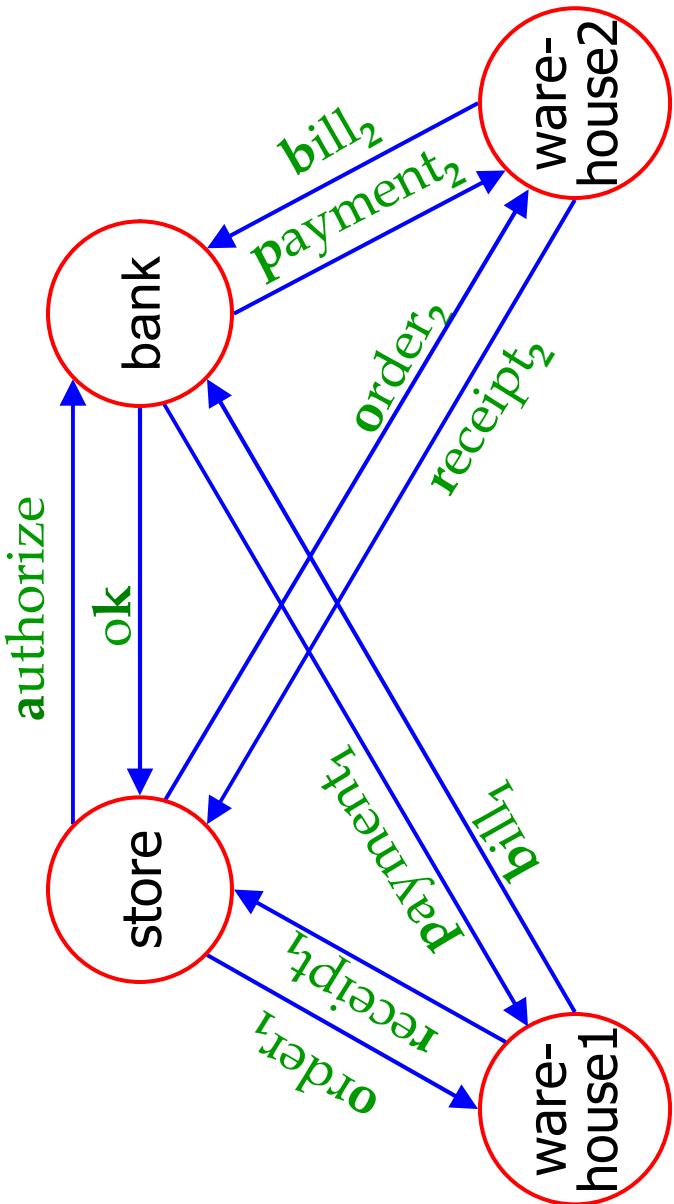
# Multi-Party Conversations

- **Watcher:** "records" the messages as they are sent



- A **conversation** is a sequence of messages the watcher sees in a successful run (or session)
- **composition language:** the set of all possible conversations
- What properties do composition languages have?

# Warehouse Example

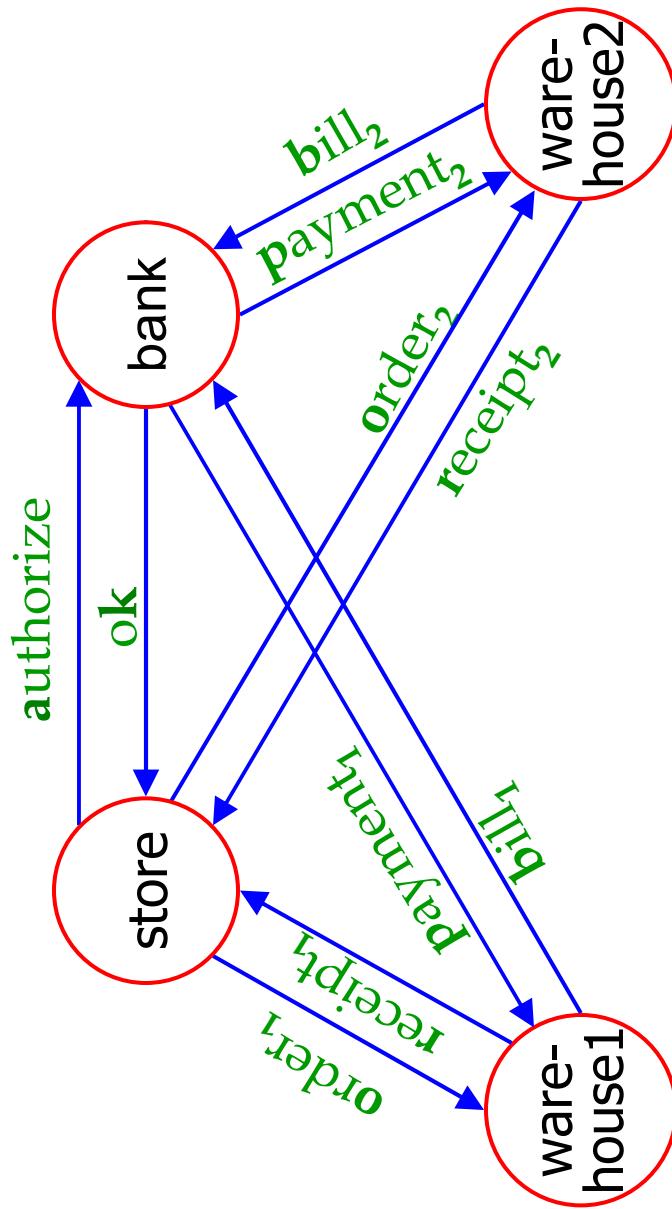


■ The composition language recognized

$a \ k \ shuff( ( \mathbf{o}_1(shuff( \mathbf{r}_1, \mathbf{b}_1 \mathbf{p}_1 ) )^*, ( \mathbf{o}_2(shuff( \mathbf{r}_2, \mathbf{b}_2 \mathbf{p}_2 ) )^* ) )$

# Web Service Composition Schema

- A composition schema is a triple  $(M, P, C)$  where
  - $M$  : finite set of message classes
  - $P$  : finite set of peers (**web services**)
  - $C$  : finite set of peer to peer channels



- Specifies the infrastructure of composition

# Global Configurations

---

- Given  $n$  Mealy implementations for composition schema  $(M, P, C)$
- **Global configuration:**  $(Q_1, t_1, \dots, Q_n, t_n, w)$  where
  - $Q_i$  : queue contents for peer  $i$
  - $t_i$  : state for peer  $i$
  - $w$  : watcher contents

# Derivation

---

$(Q_1, t_1, \dots, Q_n, t_n, w) \xrightarrow{\text{red}} (Q'_1, t'_1, \dots, Q'_n, t'_n, w')$  if

- peer  $p_i$  takes an  $\varepsilon$  move, or
- peer  $p_i$  reads an input, or
- peer  $p_i$  sends a message  $m$  to  $p_j$ 
  - $(t_i, !m, t'_i) \in \delta_i$
  - $Q'_j = Q_j m$  ( $m$  appended to  $p_j$ 's queue)
  - $\forall k \neq i, t'_k = t_k$
  - $\forall k \neq j, Q'_k = Q_k$
  - $w' = wm$  (watcher records the message  $m$ )

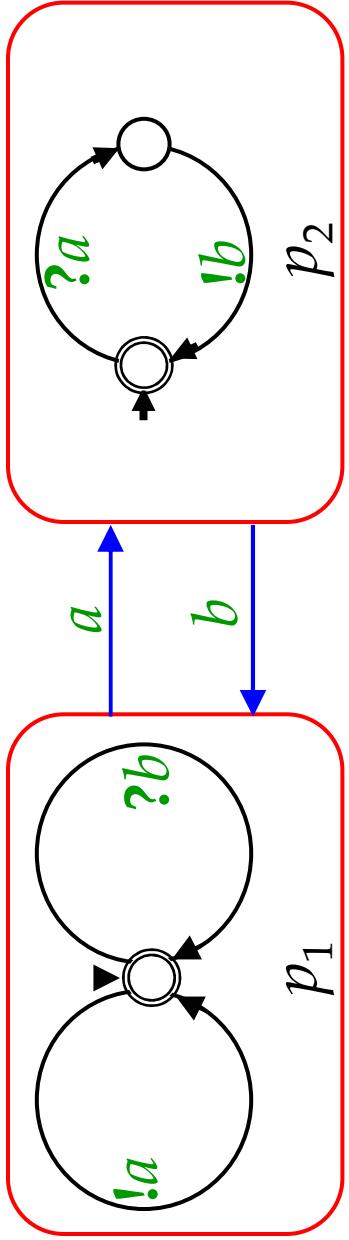
# Conversations

---

- A halting run:
  - $(\varepsilon, s_1, \dots, \varepsilon, s_n, \varepsilon) \rightarrow \dots \rightarrow (\varepsilon, f_1, \dots, \varepsilon, f_n, w)$
  - Starting from the initial configuration with empty queues and
  - Ending in final states with empty queues
- A word  $w$  is a **conversation** if
  - $(\varepsilon, s_1, \dots, \varepsilon, s_n, \varepsilon) \xrightarrow{*} (\varepsilon, f_1, \dots, \varepsilon, f_n, w)$
  - is a halting run
- Composition language (CL):
  - the set of all conversations

# Composition Languages Are Regular?

---



- $\text{CL} \cap a^*b^* = a^n b^n$
- Composition languages are not always regular
- Some may not even be context free
- Causes: **asynchronous communication & unbounded queue**
- **Bounded** queues or **synchronous**: CL always regular

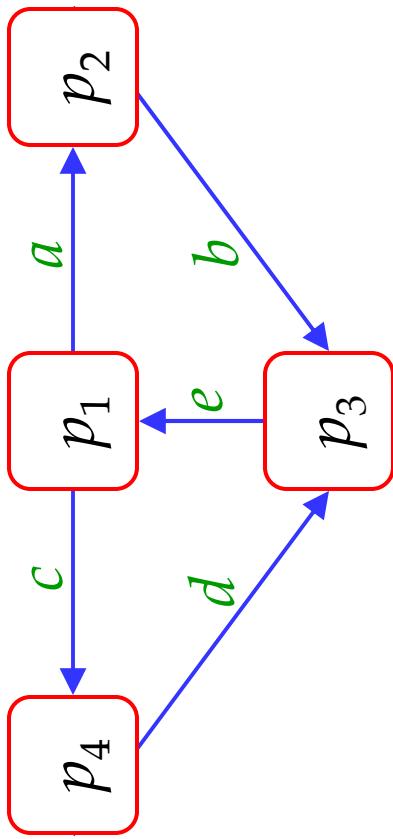
# Bounded Queues

---

- Synchronous messaging case: composition language can be recognized by the product machine
- If queues are bounded, composition languages are also regular
  - Production machines can be constructed

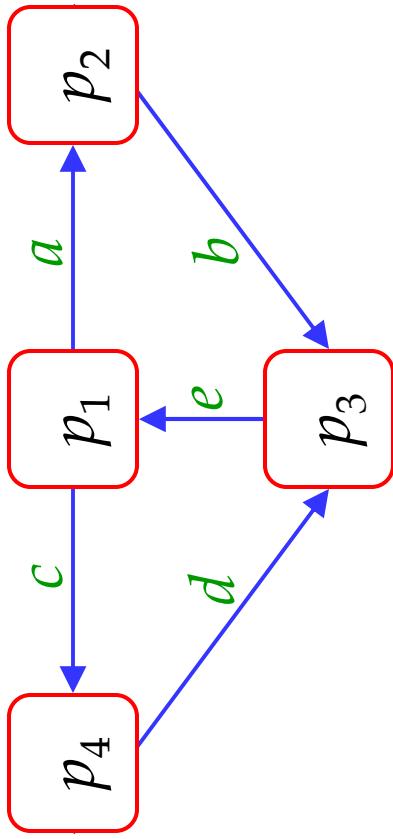
# Conversations as Orchestration

---



- “Programming” interactions
- Two questions:
  - Given Mealy peers, what conversations can they have?
  - Given a conversation language, can we “implement” it?

## Two Factors



- Local views: Conversations with the same local views are not distinguishable:
  - $abcde$  and  $acbde$
- Queuing effect: a peer can postpone sending a message

## Local Views

---

- Local view of a conversation for a peer: part of the execution that is related to the peer
  - Defined as projection:  $\pi_p(w)$  for a conversation  $w$
  - Two conversations cannot be distinguished if they have exactly the same set of local views

# Example

---



- If  $abc$  is a part of a conversation, so are  $bac$  and  $bca$
- $\pi_{p_i}(abc) = \pi_{p_i}(bac) = \pi_{p_i}(bca) = a$  for  $i = 1, 2$
- $\pi_{p_i}(abc) = \pi_{p_i}(bac) = \pi_{p_i}(bca) = bc$  for  $i = 3, 4$

## Join

---

- Given languages  $L_i$  over  $\Sigma_{\textcolor{blue}{i}}$ ,  $1 \leq i \leq n$

$$\bowtie_i L_i = \left\{ w \mid \forall 1 \leq i \leq n, \pi_{\Sigma_i}(w) \in L_i \right\} \subseteq (\cup_i \Sigma_i)^*$$

- Composition languages  $L$  are closed under “projection-join”:

$$\bowtie_{\text{peers}} \pi_{\text{peer}}(L) \subseteq L$$

# Local Prepone

---

- If the global watcher sees:



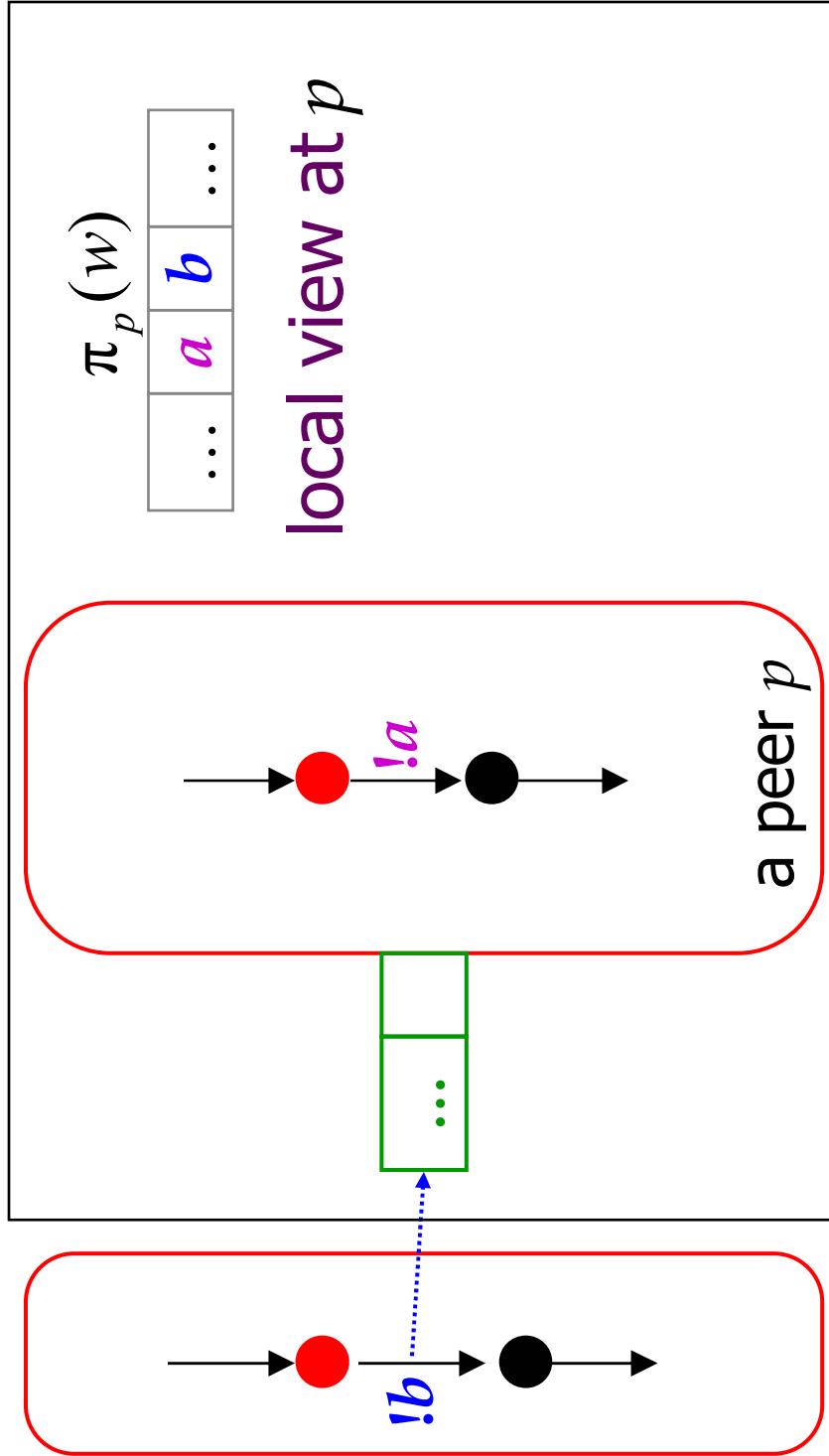
a peer

- What happens inside  $p$ ?

# Local Prepone

the global watcher:

...	<b>a</b>	<b>b</b>	...
-----	----------	----------	-----



$\pi_p(w)$  should also allow

...	<b>b</b>	<b>a</b>	...
-----	----------	----------	-----

# A Synthesis Result

---

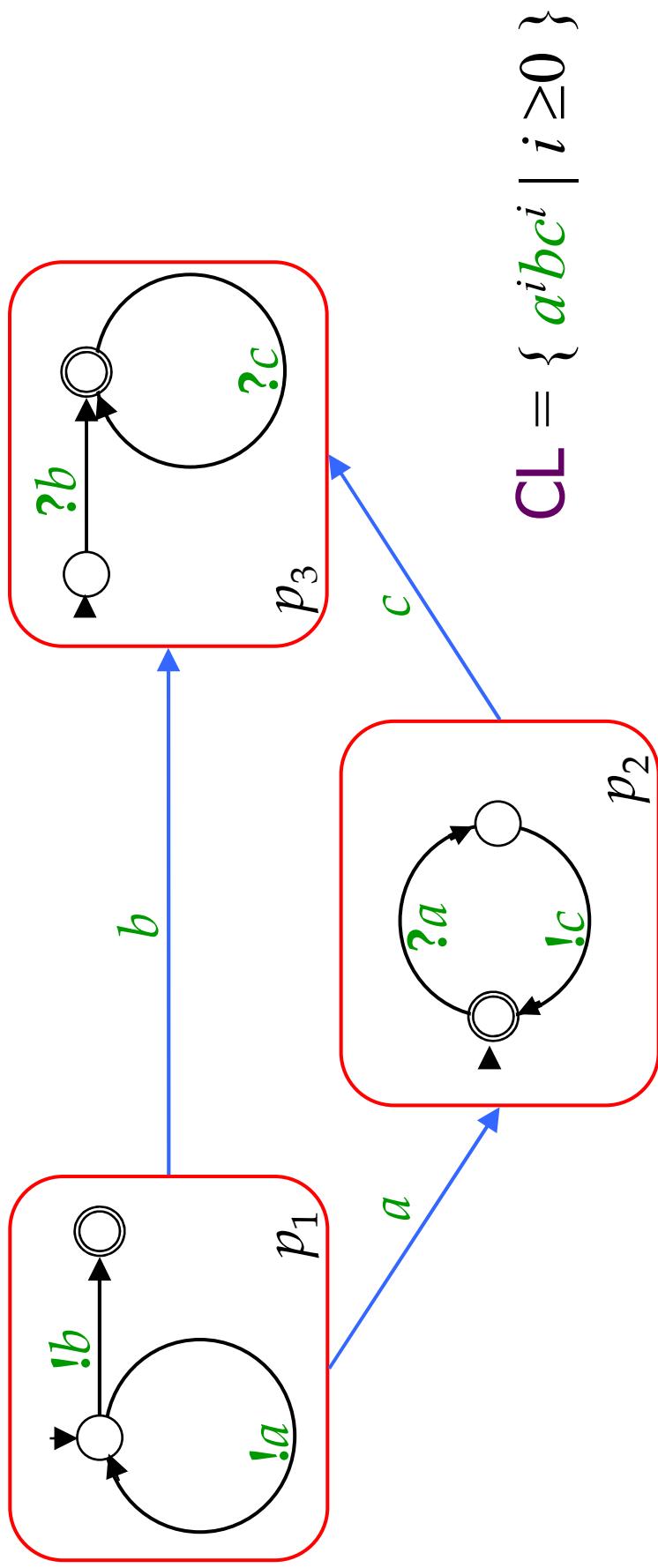
- Given a regular language  $L$ , we can find a Mealy composition such that its CL is the closure:

$$\bowtie_{\text{peers}} \text{LocalPrepone}^*(\pi_{\text{peer}}(L))$$

- Intuitively: given a regular  $L$  (e.g.,  $\text{ako}_1\dots$ ), we can find Mealy peers whose conversations are not arbitrary
  - Opportunity for automatic composition
- But some Mealy compositions do not relate to any regular languages in this way

# The Converse (General Case)

- There is an Mealy compositions whose CL is not  
 $\bowtie_{\text{peers}}^* (\pi_{\text{peer}}(L))$
- for every regular languages  $L$



# The Tree Case

---

- When the schema graph is a tree, then the Mealy composition has a composition language equal to

$$\bowtie_{\text{peers}} \text{LocalPrepone}^*(\pi_{\text{peer}}(L))$$

for some regular languages  $L$

- Intuitively: the global behavior of bottom-up composition is still predictable if the composition infrastructure is a tree
  - In particular, adding an mediator (hub-spoke) isn't a bad idea!

## Hub-and-spoke

---

- For every star-shaped composition schema, and every regular language  $L$ , we can construct an Mealy composition whose  $\text{CL} = L$
- Good news for hub-and-spoke!

# Results of Mealy Web Services

---

1. CLs of **some** Mealy compositions are not regular, **some** not context free
2. The “prepend” and “join” closure of **every** regular language = CL of **some** composite Mealy web services
3. The converse of 2. is not true in general, true in special cases
  - However: if bounded queue or synchronous:
    - CL of **every** Mealy composition is regular
    - Design time decision! Need to be explicit in specifications (BPEL4WS, BPLM, ...)

# Communicating FSAs

---

- Two communicating FSAs can simulate Turing machines

