

MoCha- π

An Exogenous Coordination Calculus
based on Mobile Channels.

Juan Guillen-Scholten

juan@cwil.nl

<http://homepages.cwi.nl/~juan>

ACG (Revised Slides)

Amsterdam, March 8, 2005.



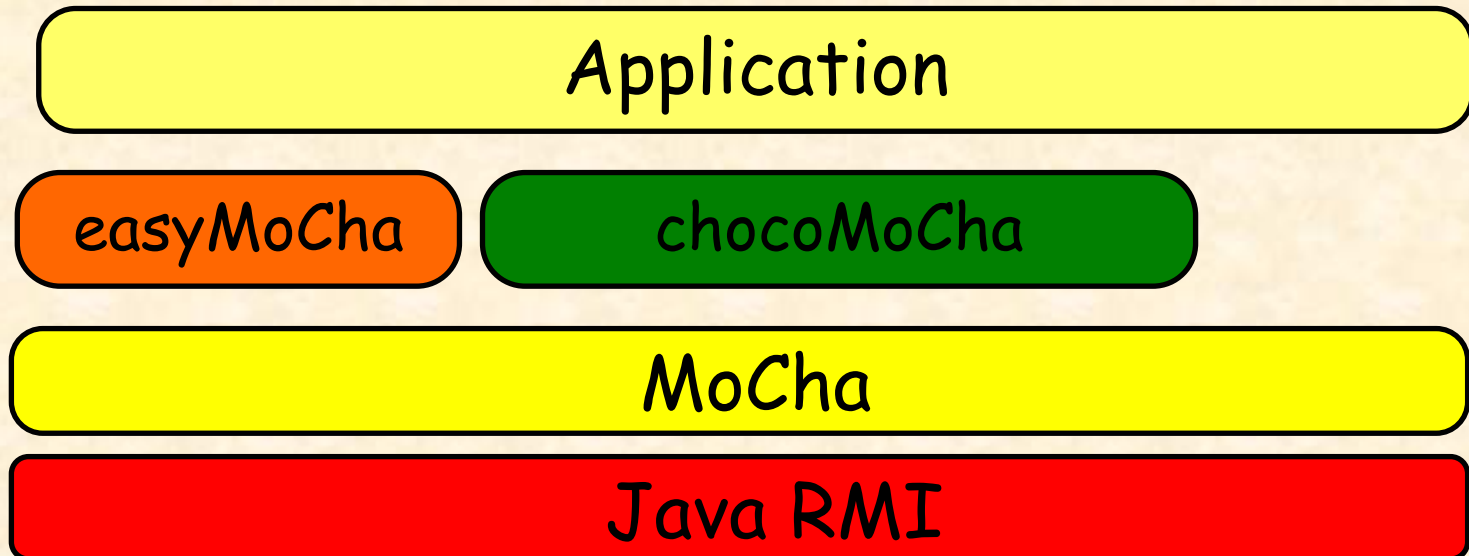
Overview



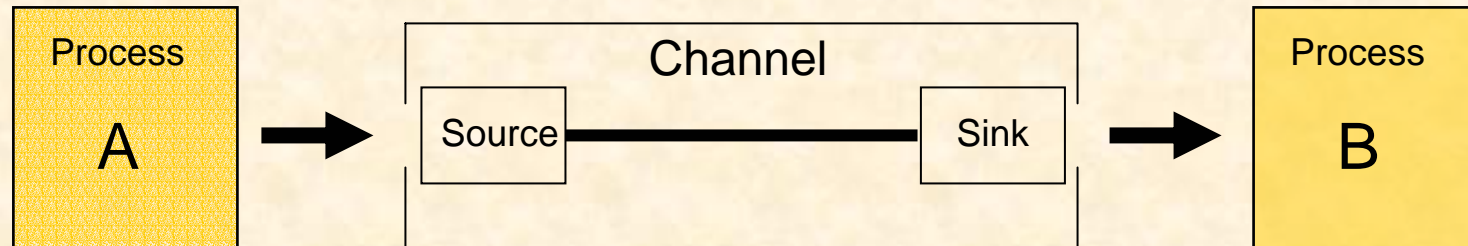
- ✓ The MoCha Middleware
- ✓ Our Goal
- ✓ Extending the π -calculus
- ✓ MoCha- π
 - Definitions
 - Actions
 - Structural Congruence
 - Rules
- ✓ Channel Examples
- ✓ A Producer/Consumer Example
- ✓ Conclusions
- ✓ Questions

The MoCha Middleware

- *MoCha*, a coordination middleware based on mobile channels.
- Implemented in Java RMI.
- Comes in different flavors: *MoCha*, *easyMoCha*, and *chocoMoCha*.
- In this talk we concentrate on the *chocoMoCha* version.



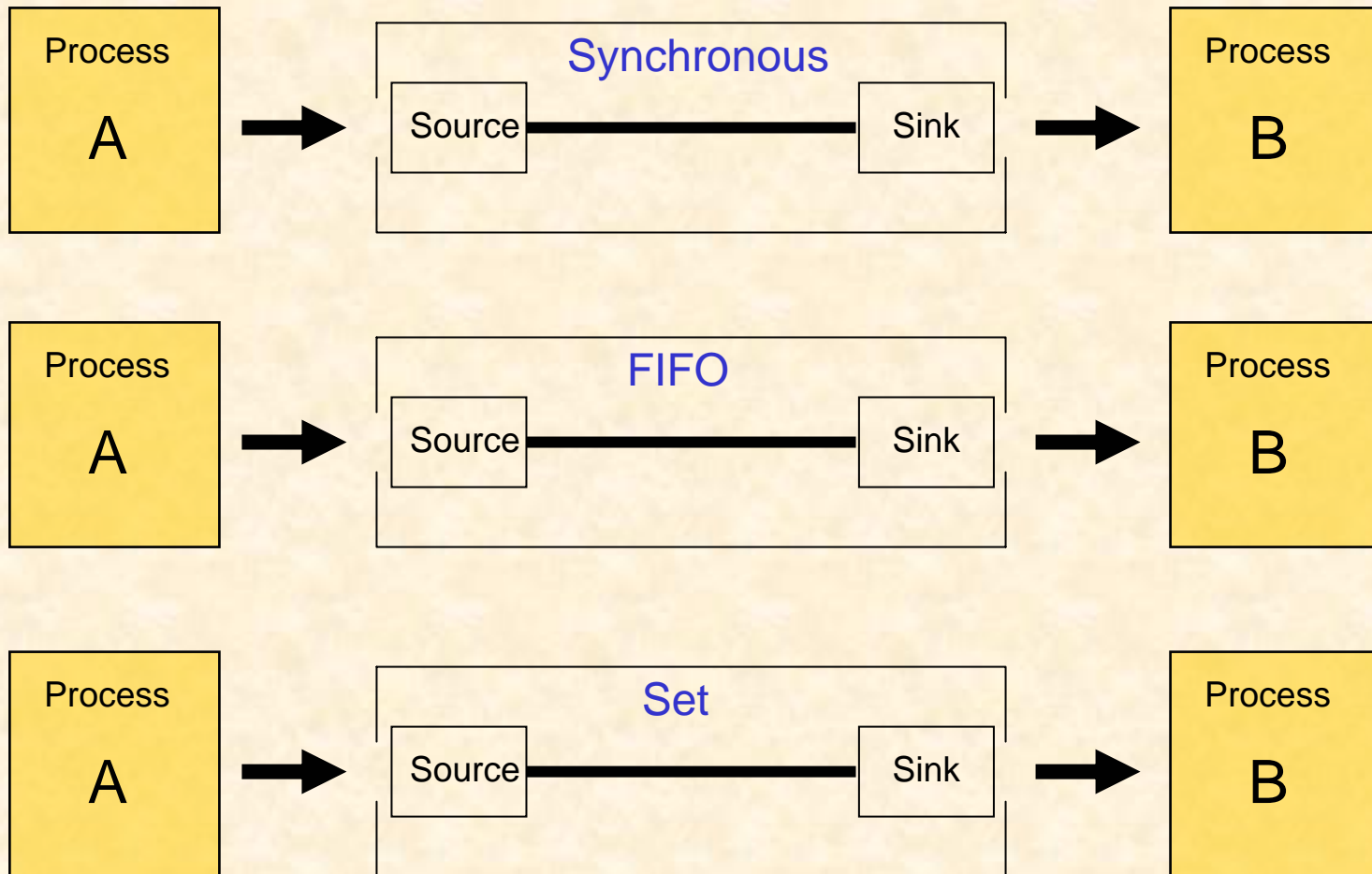
The MoCha middleware



- A channel has two distinct ends: Usually (*source, sink*), but also (*source, source*) and (*sink, sink*).
- Components write to the source-end, and take from the sink-end.
 - ✓ The dataflow is locally *one-way*.
 - ✓ The I/O operations performed on channel-ends are synchronous.
- Channels are point-to-point. They allow both client/server as well as peer-to-peer architectures.
- The communication is *anonymous*.
- Channels have different types (synchronous and asynchronous).
- Channels are mobile **➡** Their channel-ends are mobile.

The MoCha Middleware

✓ *Transparent exogenous coordination.*



Our Goal

To provide a logical model of the MoCha middleware for:

- Specification
- Verification
- Modeling of interactions between processes and channels
- Giving formal semantics for mobile channels

Extending the π -calculus

The π -calculus [Milner] is a basic mathematical model that focuses on the interaction between processes.

Points in favor for using the π -calculus:

- ✓ The calculus focuses only on process *communication* and *coordination*, not computation.
- ✓ The calculus implicitly models the notion of location. We can *abstract away from distribution*.
- ✓ Processes already communicate through channels.

Extending the π -calculus

- ✓ Like with mobile channels, π -calculus channels can be sent through channels themselves.
- ✓ The calculus is easy, but powerful enough, to use.
- ✓ There is a lot of support for the π -calculus . Both theory and tools.

Extending the π -calculus

Points against using the π -calculus:

- ✓ The calculus only supports one channel type: the *synchronous* type.
- ✓ MoCha works with channel-ends. The π -calculus works with channels as primitive entities.
- ✓ In MoCha channels are seen as resources, where processes have to connect to them. In the π -calculus any process can use a channel that it knows, at any time.
- ✓ There is no exogenous coordination.

Extending the π -calculus

Possible solution (1):

Build an architecture in the calculus that implements all of the requirements presented in the previous slide.

- *Implement channel-ends* as processes.
- Somehow, *relate* the ends of a channel.
- Make sure that other processes can communicate with channel-ends.
- Somehow, keep an *administration* regarding the *connect* and *disconnect* operations. And enforce this administration!
- Implement *different* channel-end processes for each *channel type*.

Extending the π -calculus

Possible solution (1):

➡ This architecture has to be present with every model.

Drawbacks,

- The specification is complex ➡ makes the models complex.
- The models become unnecessarily big.
- We get to see all kind of irrelevant details.

Extending the π -calculus

Possible solution (2):

Extend the π -calculus with *high-level constructs* for channel-ends, resources, and mobile channel actions.

Then, *dynamically* translate the high-level mobile channel actions into traditional π -calculus ones when needed.

Benefits:

- ✓ no complex architecture needed,
- ✓ easy specifications, and,
- ✓ in the end, everything is (dynamically) translated into the π -calculus.

We call this extension: the *MoCha- π* calculus.

MoCha- π calculus, Definitions

A system in MoCha- π consists of four kinds of processes: *threads, channels, runtime processes and resources.*

Definition 2.3.1 A Thread is a user-defined process specification with grammar L^φ that has the following syntax:

$$T ::= \sum_{i \in I} \varphi_i . T_i \mid T_1 \mid T_2 \mid \text{new } x \ T \mid A(y_1, \dots, y_n)$$

where I is any finite indexing set. The actions φ of threads are:

$e \downarrow$ connect to channel-end e

$e \uparrow$ disconnect from channel-end e

$e! \langle x \rangle$ write x to channel-end e

$e?(x)$ take x from channel-end e

τ unobservable action

MoCha- π calculus, Definitions

Definition 2.3.2 *A Channel is a user-defined process specification with grammar L^ϑ that has the following syntax:*

$$K ::= \sum_{i \in I} \vartheta_i . K_i \mid K_1 | K_2 \mid \text{new } x \ K \mid A_K(y_1, \dots, y_n)$$

where I is any finite indexing set. The actions ϑ of channels are:

$\bar{c}(x)$ *send x along link c*

$c(x)$ *receive x along link c*

τ *unobservable action*

MoCha- π calculus, Definitions

Definition 2.3.3 *A runtime process is an operational semantic process for either a thread or a channel. Its definition is given by $L^\pi = L^{\varphi \cup \vartheta}$. The runtime process expressions are defined by the following syntax:*

$$P ::= \sum_{i \in I} \pi_i.P_i \mid P_1 | P_2 \mid \text{new } x P \mid e[P] \mid A(y_1, \dots, y_n)$$

where I is any finite indexing set, and the actions $\pi = \varphi \cup \vartheta$.

MoCha- π calculus, Definitions

Definition 2.3.4 *A resource is a process without a body that always runs in parallel with the processes of a system. There is a set of resources associated with every channel-end. Therefore, we define a relation between the name of a channel-end and its resource names. We denote by \mathcal{R}^e a resource that belongs to channel-end e .*

Each end e of a channel process K has a user defined number of resources $\mathcal{R}_1^e, \mathcal{R}_2^e, \mathcal{R}_3^e, \dots$. We use $\mathcal{R}^e \in \{\mathcal{R}_1^e, \mathcal{R}_2^e, \mathcal{R}_3^e, \dots\}$ to refer to any resource of a particular channel-end e .

MoCha- π calculus, Actions

We now give the actions of our calculus:

send: $\bar{c}\langle x \rangle$

A name x is sent through link c

receive: $c(x)$

A name x is received through link c . Complementary action of send.

MoCha- π calculus, Actions

$$\textit{connect}: e \downarrow.P + Q \mid \mathcal{R}^e \longrightarrow e[P]$$

- For a successful channel-end connection one of the resources of the end e must be available.
- After the action, this resource is removed from the expression (hidden).
- Processes that try to connect are blocked until a resource becomes available.

There is a version of this action where process P is already connected to the channel-end. This action always succeeds.

MoCha- π calculus, Actions

$$\textit{disconnect}: e[e \uparrow P + Q] \longrightarrow P \mid \mathcal{R}^e$$

- Process P is connected to a channel-end e .
- After disconnecting, a resource becomes available; is set “free” for other processes to use.

There is a version of this action where P is not connected to e in the first place. The disconnect action still succeeds but nothing happens.

MoCha- π calculus, Actions

Write: $e[e!\langle a \rangle.P + Q] \longrightarrow e[\bar{e}\langle a \rangle.e(\lambda).P]$

- A write action on a channel-end e is translated into a communication pattern.
- This pattern consists of π -calculus actions only.
- First a value a is sent to a channel process.
- Afterwards, we wait for an acknowledgment λ .
- The channel process is going to match this pattern.

MoCha- π calculus, Actions

Take: $e[e?(b).P + Q] \longrightarrow e[\bar{e}\langle\lambda\rangle.e(b).P]$

- A take action on a channel-end e is also translated into a communication pattern of π -calculus actions .
- First a request λ is sent to a channel process.
- Afterwards, we wait for our value b .
- The channel process is going to match this pattern.

MoCha- π calculus, Actions

$$\text{Tau: } \tau.P + Q \longrightarrow P$$

- Finally, τ represents the unobservable action.

MoCha- π calculus, Structural Congruence

Definition 2.3.5 *Two process expressions P and Q in the MoCha- π calculus are structurally congruent, written $P \equiv Q$, if we can transform one into the other by using the following equations (in either direction):*

1. *Systematic change of bound names (alpha-conversion)*
2. *Reordering of terms in a summation*
3. $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
4. $\text{new } x (P \mid Q) \equiv P \mid \text{new } x Q$ if $x \notin \text{fn}(P)$
 $\text{new } x \mathbf{0} \equiv \mathbf{0}$, $\text{new } x y P \equiv \text{new } y x P$
5. $e[P \mid Q] \equiv P \mid e[Q]$ if $e \notin n(P)$
 $e[g[P]] \equiv g[e[P]]$
6. $A(\vec{y}) \equiv \{y/x\}P$ if $A(\vec{x}) \stackrel{\text{def}}{=} P$

where $n(P)$ are all the names in process P , with $n(\mathcal{R}^e) = e$. $\text{fn}(P)$ are all the free names in process P .

MoCha- π calculus, Rules

$$\frac{\textit{Parallel}}{P \longrightarrow P'} \\ \hline P|Q \longrightarrow P'|Q$$

$$\frac{\textit{Restriction(1)}}{P \longrightarrow P'} \\ \hline \textit{new } x P \longrightarrow \textit{new } x P'$$

$$\frac{\textit{Restriction(2)}}{P \longrightarrow P'} \\ \hline e[P] \longrightarrow e[P']$$

$$\frac{\textit{Restriction(3)}}{P \xrightarrow{e\uparrow} P'} \\ \hline e[P] \longrightarrow P' \mid \mathcal{R}^e$$

Structural Rule

$$\frac{P \longrightarrow P'}{Q \longrightarrow Q'} \quad \textit{if } P \equiv Q \textit{ and } P' \equiv Q'$$

Reaction:

$$\begin{aligned} (c(a).P + M) \mid (\bar{c}\langle b \rangle.Q + N) &\longrightarrow \{^b/a\}P \mid Q \\ e[(c(a).P + M)] + S \mid (\bar{c}\langle b \rangle.Q + N) &\longrightarrow \{^b/a\}e[P] \mid Q \\ (c(a).P + M) \mid e[(\bar{c}\langle b \rangle.Q + N)] + S &\longrightarrow \{^b/a\}P \mid e[Q] \end{aligned}$$

Channel Examples, Synchronous Type

The synchronous channel type in MoCha- π :

$$K(l, r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l, r)$$
$$K'(l, r) \stackrel{def}{=} l(x).r(\lambda).(\bar{l}\langle\lambda\rangle \mid \bar{r}\langle x\rangle).K'(l, r)$$

This channel process has a source-end $CE(l)$, and a sink-end $CE(r)$.

Initially the process first receives a name, x , from its source-end, then *sequentially* it receives a request from its sink-end.

Finally, it sends in *parallel* an acknowledgment to both channel-ends.

Channel Examples, FIFO Type

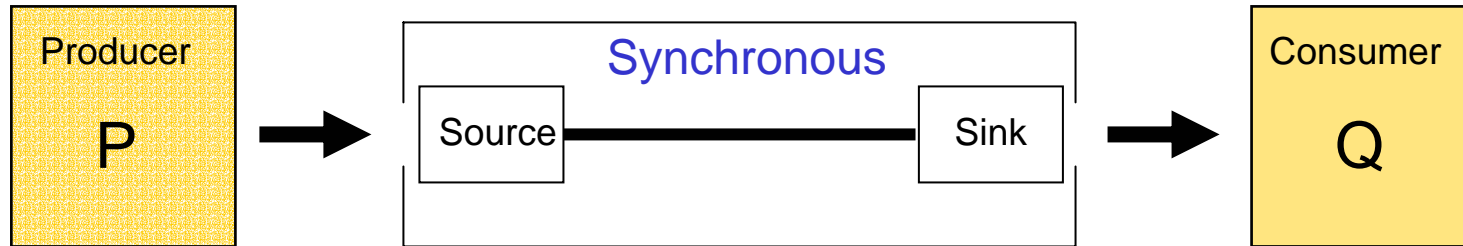
The FIFO channel type in MoCha- π :

$$\begin{aligned} K(l, r) &\stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l, r, 0) \\ K'(l, r, \vec{v})_{\{|\vec{v}|=0\}} &\stackrel{def}{=} l(v).\bar{l}\langle\lambda\rangle.K'(l, r, \langle v \rangle) \\ K'(l, r, \vec{v})_{\{|\vec{v}|\geq 1\}} &\stackrel{def}{=} (l(v).\bar{l}\langle\lambda\rangle.K'(l, r, \langle v_1, \dots, v_{|\vec{v}|}, v \rangle)) + \\ &\quad (r(\lambda).\bar{r}.\langle v_1 \in \vec{v} \rangle.K'(l, r, \langle v_2, \dots, v_{|\vec{v}|} \rangle)) \end{aligned}$$

This channel process has a source-end $CE(l)$, and a sink-end $CE(r)$.

The buffer is represented by the vector v .

A Producer/Consumer Example



$$S \stackrel{def}{=} \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNCHRONOUS}(l, r))$$

$$P(e) \stackrel{def}{=} \text{new } d (e \downarrow . e! \langle d \rangle . e \uparrow) . P(e)$$

$$Q(e) \stackrel{def}{=} e \downarrow . e?(x) . e \uparrow . Q(e)$$

Next, we model the transfer of one element from P to Q.

A Producer/Consumer Example

Initialization:

$$S = \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNC}(l, r))$$

$$P(l) = \text{new}(d) (l \downarrow .l!\langle d \rangle.l \uparrow).P(l)$$

$$Q(r) = r \downarrow .r?(x).r \uparrow .Q(r)$$

$$\text{SYNC}(l, r) = \mathcal{R}^l \mid \mathcal{R}^r \mid \text{SYNC}'(l, r)$$

$$\text{SYNC}'(l, r) = l(x).r(\lambda).(\bar{l}\langle \lambda \rangle \mid \bar{r}\langle x \rangle).\text{SYNC}'(l, r)$$

A Producer/Consumer Example

Rewrite into a more convenient form:

$$\begin{aligned} S &= \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNC}'(l, r) \mid \mathcal{R}^l \mid \mathcal{R}^r) \\ P(l) &= \text{new}(d) (l \downarrow .l!\langle d \rangle .l \uparrow).P(l) \\ Q(r) &= r \downarrow .r?(x).r \uparrow .Q(r) \\ \text{SYNC}'(l, r) &= l(x).r(\lambda).(\bar{l}\langle \lambda \rangle \mid \bar{r}\langle x \rangle).\text{SYNC}'(l, r) \end{aligned}$$

A Producer/Consumer Example

$$\xrightarrow{l \downarrow}_1$$

$$S = \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNC}'(l, r) \mid \mathcal{R}^r)$$

$$P(l) = \text{new}(d) (\underline{l[l! \langle d \rangle . l \uparrow]} . P(l))$$

$$Q(r) = r \downarrow . r?(x) . r \uparrow . Q(r)$$

$$\text{SYNC}'(l, r) = l(x) . r(\lambda) . (\bar{l} \langle \lambda \rangle \mid \bar{r} \langle x \rangle) . \text{SYNC}'(l, r)$$

$$\xrightarrow{l! \langle d \rangle}_2$$

$$S = \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNC}'(l, r) \mid \mathcal{R}^r)$$

$$P(l) = \text{new}(d) (\underline{l[\bar{l} \langle d \rangle . l(\lambda) . l \uparrow]} . P(l))$$

$$Q(r) = r \downarrow . r?(x) . r \uparrow . Q(r)$$

$$\text{SYNC}'(l, r) = l(x) . r(\lambda) . (\bar{l} \langle \lambda \rangle \mid \bar{r} \langle x \rangle) . \text{SYNC}'(l, r)$$

A Producer/Consumer Example

$Reaction(\bar{l}\langle d \rangle, l(x))$
 $\xrightarrow{3}$

$S = new(l, r) (P(l) \mid Q(r) \mid SYNC'(l, r) \mid \mathcal{R}^r)$

$P(l) = new(d) (l[l(\lambda).l \uparrow].P(l)]$

$Q(r) = r \downarrow .r?(x).r \uparrow .Q(r)$

$SYNC'(l, r) = \underline{r}(\lambda).(\bar{l}\langle \lambda \rangle \mid \bar{r}\langle d \rangle).SYNC'(l, r)$

$\xrightarrow{r \downarrow}_4$

$S = new(l, r) (P(l) \mid Q(r) \mid \underline{SYNC'(l, r)})$

$P(l) = new(d) (l[l(\lambda).l \uparrow].P(l)]$

$Q(r) = r[\underline{r?(x).r \uparrow .Q(r)}]$

$SYNC'(l, r) = r(\lambda).(\bar{l}\langle \lambda \rangle \mid \bar{r}\langle d \rangle).SYNC'(l, r)$

A Producer/Consumer Example

$$\xrightarrow[r?(x)]{5}$$

$$S = \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNC}'(l, r))$$

$$P(l) = \text{new}(d) (l[l(\lambda).l \uparrow].P(l)]$$

$$Q(r) = r[\underline{\bar{r}\langle\lambda\rangle}.r(x).r \uparrow].Q(r)]$$

$$\text{SYNC}'(l, r) = r(\lambda).(\bar{l}\langle\lambda\rangle \mid \bar{r}\langle d\rangle).\text{SYNC}'(l, r)$$

$$\text{Reaction:}(\bar{r}\langle\lambda\rangle, r(\lambda))$$
$$\xrightarrow{6}$$

$$S = \text{new}(l, r) (P(l) \mid Q(r) \mid \text{SYNC}'(l, r))$$

$$P(l) = \text{new}(d) (l[l(\lambda).l \uparrow].P(l)]$$

$$Q(r) = r[\underline{r(x)}.r \uparrow].Q(r)]$$

$$\text{SYNC}'(l, r) = \underline{\bar{l}\langle\lambda\rangle} \mid \bar{r}\langle d\rangle).\text{SYNC}'(l, r)$$

A Producer/Consumer Example

Reaction: $(\bar{l}\langle\lambda\rangle, l(\lambda)) ; (\bar{r}\langle d\rangle, r(x))$
 $\xrightarrow{7,8}$

$S = new(l, r) (P(l) \mid Q(r) \mid SYNC'(l, r))$

$P(l) = l[\underline{l \uparrow}] . P(l)$

$Q(r) = r[\underline{r \uparrow}] . Q(r)$

$SYNC'(l, r) = \underline{SYNC'}(l, r)$

$\xrightarrow{l \uparrow; r \uparrow}$
 $\xrightarrow{9,10}$

$S = new(l, r) (P(l) \mid Q(r) \mid SYNC'(l, r) \mid \underline{\mathcal{R}^l} \mid \underline{\mathcal{R}^r})$

$P(l) = \underline{P}(l)$

$Q(r) = \underline{Q}(r)$

$SYNC'(l, r) = SYNC'(l, r)$

Conclusions

In this talk we presented MoCha- π , an exogenous coordination calculus based on mobile channels.

The new and major features of our calculus are:

- *User-defined channel types*, without having to change the rules of the calculus itself.
- *Exogenous coordination*.
- *Mobile channels* are viewed as *resources*. Processes must compete to get exclusive access to a particular channel-end.

Questions?

Thank you for listening!
Are there any questions?

You can also ask me by e-mail:

Juan Guillen-Scholten

`juan@cwi.nl`

`http://homepages.cwi.nl/~juan`

Acknowledgements

Many thanks to:

- Dave Clarke,
- Nikolay Diakov,
- Helle Hansen,
- Clemens Kupke,
- and, the rest of the ACG members present during this talk.

for their comments on these slides.