

Smooth Orchestrators

Cosimo Laneve, Luca Padovani

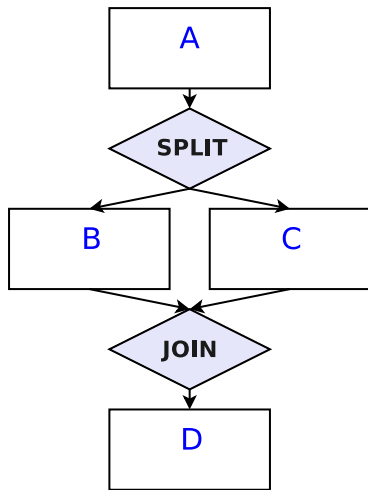
University of Bologna, University of Urbino

29 march 2006

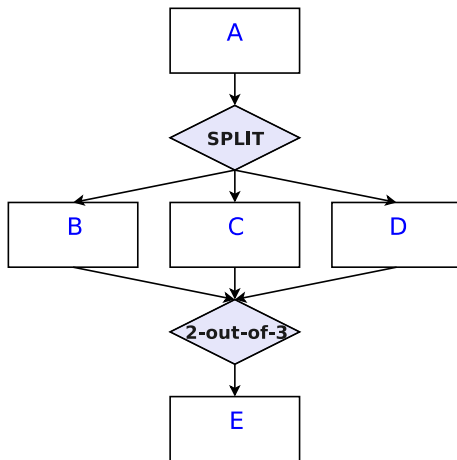
Summary

- Background
- Formal development
- Implementation
- Extensions and concluding remarks

Well-known workflow patterns: synchronization



Well-known workflow patterns: n -out-of- m



Synchronization patterns

Define a primitive *construct* that models synchronization patterns

Join-patterns in JoCaml:

```
let state(y) | get() = P
and state(y) | put(new_y) = Q
```

Remark: atomic input operation from **multiple** channels

Similar construct, different context

A synchronization pattern implemented in the join calculus is

- *permanent*
- *closed*

Orchestration of services is more dynamic:

- ephemeral synchronization
- extending existing services

Similar construct, different context

A synchronization pattern implemented in the join calculus is

- *permanent*
- *closed*

Orchestration of services is more dynamic:

- ephemeral synchronization
- extending existing services

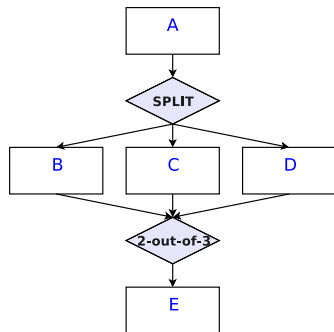
π with orchestrators

Asynchronous π -calculus with *orchestrators*:

$P ::=$		processes
	0	(nil)
	$\bar{x}[\tilde{u}]$	(output)
	$\sum_{i \in I} J_i \triangleright P_i$	(orchestrator)
	$(x)P$	(new)
	$P \mid P$	(parallel)
	$!P$	(replication)

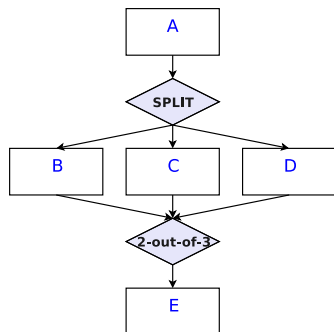
$J ::=$		join patterns
	$x(\tilde{u})$	(input)
	$J \& J$	(join)

Example: n -out-of- m



$$\begin{aligned} a_end(v) \triangleright & \\ & \overline{b_start[v]} \mid \overline{c_start[v]} \mid \overline{d_start[v]} \\ & \mid b_end(x) \& c_end(y) \triangleright \overline{e_start[x, y]} \\ + & b_end(x) \& d_end(z) \triangleright \overline{e_start[x, z]} \\ + & c_end(y) \& d_end(z) \triangleright \overline{e_start[y, z]} \end{aligned}$$

Example: n -out-of- m



$$\begin{aligned} a_end(v) \triangleright & \\ & \overline{b_start[v]} \mid \overline{c_start[v]} \mid \overline{d_start[v]} \\ & \mid b_end(x) \& c_end(y) \triangleright \overline{e_start[x, y]} \\ + & b_end(x) \& d_end(z) \triangleright \overline{e_start[x, z]} \\ + & c_end(y) \& d_end(z) \triangleright \overline{e_start[y, z]} \end{aligned}$$

Linearity is hard to enforce statically:

$$\bar{x}[a, a] \mid x(u, v) \triangleright u(y) \& v(z) \triangleright P \rightarrow a(y) \& a(z) \triangleright P\{a/u, a/v\}$$

Global consensus:

Take $x(u) \& y(v) \triangleright P$ located at ℓ

- If x and y are *not* located at ℓ this reduction requires *non-local* – global – information
- Migrating the process does not help either (and who likes mobile agents anyway?)

These are non-issues in the join-calculus:

- joined channels are *fresh*...
- ... hence they are co-located

Linearity is hard to enforce statically:

$$\bar{x}[a, a] \mid x(u, v) \triangleright u(y) \& v(z) \triangleright P \rightarrow a(y) \& a(z) \triangleright P\{a/u, a/v\}$$

Global consensus:

Take $x(u) \& y(v) \triangleright P$ located at ℓ

- If x and y are *not* located at ℓ this reduction requires *non-local* – global – information
- Migrating the process does not help either (and who likes mobile agents anyway?)

These are non-issues in the join-calculus:

- joined channels are *fresh*...
- ... hence they are co-located

Linearity is hard to enforce statically:

$$\bar{x}[a, a] \mid x(u, v) \triangleright u(y) \& v(z) \triangleright P \rightarrow a(y) \& a(z) \triangleright P\{a/u, a/v\}$$

Global consensus:

Take $x(u) \& y(v) \triangleright P$ located at ℓ

- If x and y are *not* located at ℓ this reduction requires *non-local* – global – information
- Migrating the process does not help either (and who likes mobile agents anyway?)

These are non-issues in the join-calculus:

- joined channels are *fresh* . . .
- . . . hence they are co-located

Our solution: co-location constraints

$P ::=$ **processes**

\dots
| $(x@y)P$ (new)

$J ::=$ **join patterns**

| $x(\tilde{u}@ \tilde{v})$ (input)

| $J \& J$ (join)

- $(x@y)P$ means “create x at the same location as y ”
- $(x@x)P$ means “create x at whatever location”
- $x(u@u, v@v) \triangleright P$ means “receive u and v no matter what their location is”
- $x(u@u, v@u) \triangleright P$ means “receive u and v if they are co-located”

Our solution: co-location constraints

$P ::=$ **processes**

...
| $(x@y)P$ (new)

$J ::=$ **join patterns**

| $x(\tilde{u}@ \tilde{v})$ (input)
| $J \& J$ (join)

- $(x@y)P$ means “create x at the same location as y ”
- $(x@x)P$ means “create x at whatever location”
- $x(u@u, v@v) \triangleright P$ means “receive u and v no matter what their location is”
- $x(u@u, v@u) \triangleright P$ means “receive u and v if they are co-located”

Our solution: co-location constraints

$P ::=$ **processes**

...
| $(x@y)P$ (new)

$J ::=$ **join patterns**

| $x(\tilde{u}@ \tilde{v})$ (input)
| $J \& J$ (join)

- $(x@y)P$ means “create x at the same location as y ”
- $(x@x)P$ means “create x at whatever location”
- $x(u@u, v@v) \triangleright P$ means “receive u and v no matter what their location is”
- $x(u@u, v@u) \triangleright P$ means “receive u and v if they are co-located”

Our solution: co-location constraints

$P ::=$ **processes**

...
| $(x@y)P$ (new)

$J ::=$ **join patterns**

| $x(\tilde{u}@ \tilde{v})$ (input)
| $J \& J$ (join)

- $(x@y)P$ means “create x at the same location as y ”
- $(x@x)P$ means “create x at whatever location”
- $x(u@u, v@v) \triangleright P$ means “receive u and v no matter what their location is”
- $x(u@u, v@u) \triangleright P$ means “receive u and v if they are co-located”

Co-location constraints: reduction semantics

Co-location relation:

$$\begin{array}{l} \text{(BASE)} \\ (\tilde{x} @ \tilde{y})(u @ v) \vdash u \frown v \end{array}$$

$$\begin{array}{l} \text{(LIFT)} \\ \frac{\tilde{x} @ \tilde{y} \vdash u \frown v \quad u, v \neq z}{(\tilde{x} @ \tilde{y})(z @ z') \vdash u \frown v} \end{array}$$

Reduction:

$$\tilde{z} @ \tilde{y} \vdash a \frown b$$

$$(\tilde{z} @ \tilde{y}) \left(\bar{x}[a] \mid \bar{y}[b] \mid x(u @ u) \& y(v @ u) \triangleright P \right) \rightarrow (\tilde{z} @ \tilde{y}) P\{a, b / u, v\}$$

Co-location constraints: reduction semantics

Co-location relation:

$$\begin{array}{c} \text{(BASE)} \\ (\tilde{x} @ \tilde{y})(u @ v) \vdash u \frown v \end{array}$$

$$\begin{array}{c} \text{(LIFT)} \\ \frac{\tilde{x} @ \tilde{y} \vdash u \frown v \quad u, v \neq z}{(\tilde{x} @ \tilde{y})(z @ z') \vdash u \frown v} \end{array}$$

Reduction:

$$\tilde{z} @ \tilde{y} \vdash a \frown b$$

$$(\tilde{z} @ \tilde{y}) \left(\bar{x}[a] \mid \bar{y}[b] \mid x(u @ u) \& y(v @ u) \triangleright P \right) \rightarrow (\tilde{z} @ \tilde{y}) P\{a, b / u, v\}$$

Checking co-location

The process

$$w(x@x, y@y) \triangleright x(u@u) \& y(v@u) \triangleright P$$

raises a runtime error if provided with a message

$$\bar{w}[c, d]$$

where c and d are not co-located

- the message on w is lost forever
- we want to check co-location *statically* through a type system

Checking co-location

$$\begin{array}{c} \text{(NIL)} \\ \Lambda \vdash 0 \end{array} \quad \begin{array}{c} \text{(OUTPUT)} \\ \Lambda \vdash \bar{x}[\tilde{u}] \end{array} \quad \begin{array}{c} \text{(PAR)} \\ \frac{\Lambda \vdash P \quad \Lambda \vdash Q}{\Lambda \vdash P \mid Q} \end{array} \quad \begin{array}{c} \text{(BANG)} \\ \frac{\Lambda \vdash P}{\Lambda \vdash !P} \end{array}$$

$$\begin{array}{c} \text{(ORCH)} \\ \frac{\Lambda(u@u)(v@u) \vdash P \quad \Lambda \vdash x \frown y}{\Lambda \vdash x(u@u) \& y(v@u) \triangleright P} \end{array} \quad \begin{array}{c} \text{(NEW)} \\ \frac{\Lambda(x@y) \vdash P}{\Lambda \vdash (x@y)P} \end{array}$$

A process P is *distributable* if $\varepsilon \vdash P$

Checking co-location

$$\begin{array}{c} \text{(NIL)} \\ \Lambda \vdash 0 \end{array} \quad \begin{array}{c} \text{(OUTPUT)} \\ \Lambda \vdash \bar{x}[\tilde{u}] \end{array} \quad \begin{array}{c} \text{(PAR)} \\ \frac{\Lambda \vdash P \quad \Lambda \vdash Q}{\Lambda \vdash P \mid Q} \end{array} \quad \begin{array}{c} \text{(BANG)} \\ \frac{\Lambda \vdash P}{\Lambda \vdash !P} \end{array}$$

$$\begin{array}{c} \text{(ORCH)} \\ \frac{\Lambda(u@u)(v@u) \vdash P \quad \Lambda \vdash x \frown y}{\Lambda \vdash x(u@u) \& y(v@u) \triangleright P} \end{array} \quad \begin{array}{c} \text{(NEW)} \\ \frac{\Lambda(x@y) \vdash P}{\Lambda \vdash (x@y)P} \end{array}$$

A process P is *distributable* if $\varepsilon \vdash P$

Subject reduction

Distributable processes reduce to distributable processes

Theorem (subject reduction): If

- $(\tilde{x} @ \tilde{y}) \vdash P$, and
- $(\tilde{x} @ \tilde{y})P \rightarrow (\tilde{x} @ \tilde{y})Q$

then

- $(\tilde{x} @ \tilde{y}) \vdash Q$

Lemma (substitution): Substitution cannot be defined for single names only

Consider

$$(a @ a)(u @ u)(v @ u) \vdash u \& v \triangleright 0 \quad \{a/v\}$$

leads to

$$(a @ a)(u @ u) \not\vdash u \& a \triangleright 0$$

Substitutions must preserve co-location

Subject reduction

Distributable processes reduce to distributable processes

Theorem (subject reduction): If

- $(\tilde{x} @ \tilde{y}) \vdash P$, and
- $(\tilde{x} @ \tilde{y})P \rightarrow (\tilde{x} @ \tilde{y})Q$

then

- $(\tilde{x} @ \tilde{y}) \vdash Q$

Lemma (substitution): Substitution cannot be defined for single names only

Consider

$$(a @ a)(u @ u)(v @ u) \vdash u \& v \triangleright 0 \quad \{a/v\}$$

leads to

$$(a @ a)(u @ u) \not\vdash u \& a \triangleright 0$$

Substitutions must preserve co-location

The smoothness restriction

We decouple the orchestrator from the continuation through the $\llbracket \cdot \rrbracket$ encoding

$$\llbracket \sum_{i \in I} J_i \triangleright P_i \rrbracket = (z_i^{i \in I}) \left(\overbrace{\sum_{i \in I} J_i \triangleright \bar{z}_i[\tilde{u}_i]}^{\text{smooth orchestrator}} \mid \underbrace{\prod_{i \in I} z_i(\tilde{u}_i) \triangleright \llbracket P_i \rrbracket}_{\text{continuation}} \right)$$

The smooth orchestrator is now free to migrate

Proposition (encoding correctness): P is barbed congruent to $\llbracket P \rrbracket$

The smoothness restriction

We decouple the orchestrator from the continuation through the $\llbracket \cdot \rrbracket$ encoding

$$\llbracket \sum_{i \in I} J_i \triangleright P_i \rrbracket = (z_i^{i \in I}) \left(\overbrace{\sum_{i \in I} J_i \triangleright \bar{z}_i[\tilde{u}_i]}^{\text{smooth orchestrator}} \mid \underbrace{\prod_{i \in I} z_i(\tilde{u}_i) \triangleright \llbracket P_i \rrbracket}_{\text{continuation}} \right)$$

The smooth orchestrator is now free to migrate

Proposition (encoding correctness): P is barbed congruent to $\llbracket P \rrbracket$

Implementation

Size of an orchestrator:

$$x_1(\tilde{u}_1 @ \tilde{v}_1) \& \cdots \& x_n(\tilde{u}_n @ \tilde{v}_n) \triangleright \bar{z}[\tilde{u}_1 \cdots \tilde{u}_n]$$

It may be encoded as

- a vector with $n + 1$ names x_1, \dots, x_n, z
- a vector of $k_1 + \dots + k_n$ values, where $k_i = |\tilde{u}_i|$:
 - ▶ the integer value j at position h indicates that the j -th and h -th bound names must be co-located
 - ▶ the constant c at position h indicates that the h -th bound name must be co-located with c

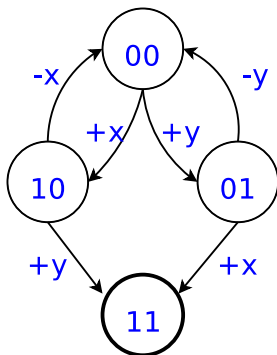
Co-location check: it basically amounts to comparing IP addresses

Compiling simple orchestrators

Turn an orchestrator into a finite-state automaton (c.f. *Le Fessant, Maranget*):

$$x(u) \& y(v) \triangleright \bar{z}[uv]$$

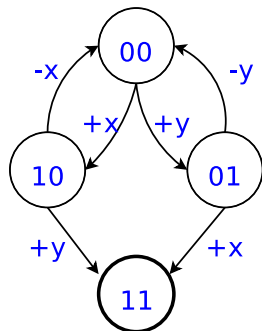
is compiled into



Compiling orchestrators with co-location constraints I

$$x(u@u, v@u) \& y(w@w) \triangleright \bar{z}[uvw]$$

is compiled into



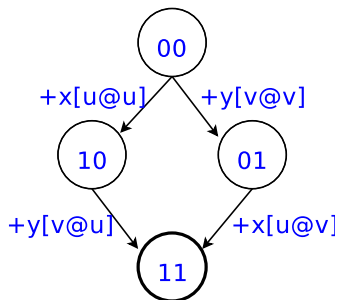
- $+x$ = “there is a message $\bar{x}[a, b]$ such that a and b are co-located”
- $-x$ = “there is **no** message $\bar{x}[a, b]$ such that a and b are co-located”

Compiling orchestrators with co-location constraints II

Consider

$$x(u@u) \& y(v@u) \triangleright P$$

Reception of a message on y depends on the message received on x .



What if the message on y arrives first?
We rewrite the orchestrator thus:

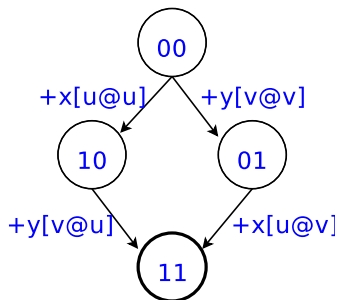
$$y(v@v) \& x(u@v) \triangleright P$$

Compiling orchestrators with co-location constraints II

Consider

$$x(u@u) \& y(v@u) \triangleright P$$

Reception of a message on y depends on the message received on x .



What if the message on y arrives first?

We rewrite the orchestrator thus:

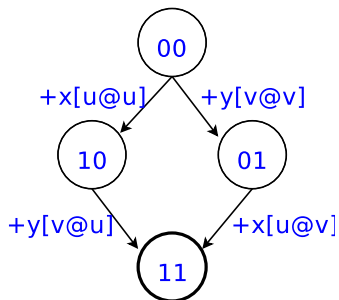
$$y(v@v) \& x(u@v) \triangleright P$$

Compiling orchestrators with co-location constraints II

Consider

$$x(u@u) \& y(v@u) \triangleright P$$

Reception of a message on y depends on the message received on x .



What if the message on y arrives first?

We rewrite the orchestrator thus:

$$y(v@v) \& x(u@v) \triangleright P$$

Compiling orchestrators: when and where?

- The smooth orchestrator is usually small in size
- The automaton is bigger, and it depends on the cardinality of name occurrences which are not known until runtime

Two strategies:

- eager compilation: bigger messages, needs patching
- delayed compilation: smaller messages, burden on the receiver

Compiling orchestrators: when and where?

- The smooth orchestrator is usually small in size
- The automaton is bigger, and it depends on the cardinality of name occurrences which are not known until runtime

Two strategies:

- eager compilation: bigger messages, needs patching
- delayed compilation: smaller messages, burden on the receiver

Extensions and conclusion

Further investigations:

- Stronger type system
 - ▶ eliminate runtime co-location checks
 - ▶ technically challenging (dangerous variables)

$$x(u@α) \& y(v@α) \triangleright P$$

x and y cannot be treated polymorphically w.r.t. $α$ because of their dependency

- Expressivity (workflow patterns)

PiDuce prototype available at

<http://www.cs.unibo.it/PiDuce/>

(C# implementation)