procedural choreographic programming

luís cruz-filipe
(joint work with fabrizio montesi)

department of mathematics and computer science
university of southern denmark

forte 2017, neuchâtel, switzerland
june 20th, 2017

outline

choreographies

choreographies

a model for distributed computation based on “common practice”

- used for modeling interactions between web services
- high-level languages, alice-and-bob notation
- good properties: message pairing, deadlock-freedom
- projectable to adequate process calculi

different usages

- choreographies as specifications (types)
- choreographies as programs (our approach)

examples (i/ii)

a simple example

$$\mathsf{alice}.\,\text{“hi”} \rightarrow \mathsf{bob};\ \mathsf{bob}.\,\text{“hello”} \rightarrow \mathsf{alice}$$

- all messages are correctly paired
- synthetizable process implementation

$$\underbrace{!\mathsf{bob}.\,\text{“hi”}\,;\ ?\mathsf{bob}}_{\mathsf{alice}} \mid \underbrace{?\mathsf{alice};\ !\mathsf{alice}.\,\text{“hello”}}_{\mathsf{bob}}$$

examples (ii/ii)

non-interfering communications can swap

$$\text{alice}.\text{“hi”} \to \text{bob};\ \text{carol}.\text{“bye”} \to \text{dan};\ \text{bob}.\text{“hello”} \to \text{alice}$$
$$\equiv$$
$$\text{carol}.\text{“bye”} \to \text{dan};\ \text{alice}.\text{“hi”} \to \text{bob};\ \text{bob}.\text{“hello”} \to \text{alice}$$
$$\equiv$$
$$\text{alice}.\text{“hi”} \to \text{bob};\ \text{bob}.\text{“hello”} \to \text{alice};\ \text{carol}.\text{“bye”} \to \text{dan}$$

is implemented as

$$\underbrace{!\text{bob}.\text{“hi”};\ ?\text{bob}}_{\text{alice}} \mid \underbrace{?\text{alice};\ !\text{alice}.\text{“hello”}}_{\text{bob}} \mid \underbrace{!\text{dan}.\text{“bye”}}_{\text{carol}} \mid \underbrace{?\text{carol}}_{\text{dan}}$$

the world of choreographies

common features (present in most languages)

- message passing/method selection
- conditional and (tail) recursion

the world of choreographies

common features (present in most languages)

- message passing/method selection
- conditional and (tail) recursion

additional features (only in particular languages)

- channel passing
- process spawning
- asynchrony
- web services
- ...

$\rightsquigarrow$ the target process calculi reflect these design choices

our motivation

goal

- study foundational aspects of choreographies
- identify minimal primitives required for particular constructions
  - computational completeness and universality
  - asynchronous communication

$\rightsquigarrow$ "bottom-up" approach, rather than "top-down"

our motivation

goal

- study foundational aspects of choreographies
- identify minimal primitives required for particular constructions
  - computational completeness and universality
  - asynchronous communication

$\rightsquigarrow$ "bottom-up" approach, rather than "top-down"

this work

- procedures in choreographies
- arbitrary composition and recursion
- runtime process spawning and name mobility

outline

motivation

$\rightsquigarrow$ we want to be able to write intuitive parallel algorithms

merge sort

given a list ℓ

1. split ℓ into ℓ_1 and ℓ_2
2. compute mergesort(ℓ_1) and mergesort(ℓ_2)
3. merge mergesort(ℓ_1) and mergesort(ℓ_2)

motivation

⇝ we want to be able to write intuitive parallel algorithms

merge sort

given a list ℓ

1. split ℓ into ℓ_1 and ℓ_2
2. compute mergesort(ℓ_1) and mergesort(ℓ_2)
3. merge mergesort(ℓ_1) and mergesort(ℓ_2)

- step 2 should be done in two parallel computations
- it is not clear how to do this with only tail recursion...

procedural choreographies

design options

- typed processes
- each process holds only one value (but might be a record)
- communication allows for computation by both parties
- parameterized global procedures
- process spawning
- name mobility (three-way communication)

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, C \rangle$

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \to \mathtt{q}.f \mid \mathtt{p} \to \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q}^{T} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \mathbf{0}$$

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, C \rangle$

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \to \mathtt{q}.f \mid \mathtt{p} \to \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q}^{T} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \mathbf{0}$$

- $\mathcal{D}$ is a set of procedure definitions (choreographies)
- C is a distinguished ("main") choreography

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, C \rangle$

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \to \mathtt{q}.f \mid \mathtt{p} \to \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q}^{T} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \mathbf{0}$$

- η are communication actions (values, label selection, spawning, name communication)
- I are instructions (conditionals, procedure calls) composed sequentially

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, C \rangle$

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \to \mathtt{q}.f \mid \mathtt{p} \to \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q}^{T} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \mathbf{0}$$

- in $\mathtt{p}.e \to \mathtt{q}.f$, p evaluates expression e and sends its result to q
- e may refer to the value stored at p
- q applies function f to the value received and stores the result
- f may refer to the value stored at q

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, C \rangle$

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q}^{T} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \mathbf{0}$$

- in $\mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$, $\mathtt{p}$ *introduces* processes $\mathtt{q}$ and $\mathtt{r}$
- afterwards, $\mathtt{q}$ and $\mathtt{r}$ can communicate directly
- these actions are required for distributed implementations

semantics

components

transition semantics over triples $\langle \mathcal{G}, C, \sigma \rangle$, parameterized by $\mathcal{D}$

- $\mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

semantics

components

transition semantics over triples $\langle \mathcal{G}, C, \sigma \rangle$, parameterized by $\mathcal{D}$

- $\mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \qquad e \downarrow_{\mathsf{p}} v \qquad f(v) \downarrow_{\mathsf{q}} w}{G, \mathsf{p}.e \to \mathsf{q}.f; C, \sigma \;\; \to_{\mathcal{D}} \;\; G, C, \sigma[\mathsf{q} \mapsto w]} \;\lfloor \mathsf{C}|\mathsf{Com} \rceil$$

possible errors

- p and q do not know each other
- e or f is not well-typed
- v does not have the type expected by f

semantics

components

transition semantics over triples $\langle \mathcal{G}, C, \sigma \rangle$, parameterized by $\mathcal{D}$

- $\mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{}{G, \mathsf{p}\ \mathtt{start}\ \mathsf{q}^{T}; C, \sigma \ \rightarrow_{\mathcal{D}}\ G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}, C, \sigma[\mathsf{q} \mapsto \bot_{T}]}\ \lfloor \mathsf{C}|\mathsf{Start} \rceil$$

semantics

components

transition semantics over triples $\langle \mathcal{G}, C, \sigma \rangle$, parameterized by $\mathcal{D}$

- $\mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{\mathtt{p} \overset{G}{\longleftrightarrow} \mathtt{q} \qquad \mathtt{p} \overset{G}{\longleftrightarrow} \mathtt{r}}{G, \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}; C, \sigma \ \rightarrow_{\mathcal{D}} \ G \cup \{\mathtt{q} \leftrightarrow \mathtt{r}\}, C, \sigma} \ \lfloor \mathsf{C|Tell} \rceil$$

possible errors

- $\mathtt{p}$ does not know $\mathtt{q}$ or $\mathtt{r}$

semantics

components

transition semantics over triples $\langle \mathcal{G}, C, \sigma \rangle$, parameterized by $\mathcal{D}$

- $\mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{X(\widetilde{\mathrm{q}^T}) = C_X \in \mathcal{D}}{\mathtt{X}\left\langle \widetilde{\mathrm{p}^{\mathrm{T}}} \right\rangle; C \preceq_{\mathcal{D}} C_X[\tilde{\mathrm{p}}/\tilde{\mathrm{q}}] \fatsemi C} \quad \lfloor \mathrm{C}|\mathrm{Unfold} \rceil$$

possible errors

- procedure X is not defined
- the types of the ps and qs do not match

semantics

components

transition semantics over triples $\langle \mathcal{G}, C, \sigma \rangle$, parameterized by $\mathcal{D}$

- $\mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

implicit parallelism

swapping relation extended to instructions (including procedure calls)

- examples to come

typing system

avoiding errors

well-typed choreographies are guaranteed never to encounter an error (deadlock-freedom by design)

- symbolic execution
- over-approximation (analyzes potentially unreachable code)
- judgements include connection "requirements" and "guarantees" for procedure calls

typing system

avoiding errors

well-typed choreographies are guaranteed never to encounter an error (deadlock-freedom by design)

- symbolic execution
- over-approximation (analyzes potentially unreachable code)
- judgements include connection “requirements” and “guarantees” for procedure calls

- type checking is decidable
- type inference is decidable
- typable choreographies may be unprojectable

outline

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
         else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
               MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
MS(p) = if is_small then 0
         else start (q1 ▷ p?id; MS<q1>; p!*);
               start (q2 ▷ p?id; MS<q2>; p!*);
               q1!split1; q2!split2; q1?id; q2?merge
```

merge sort revisited

choreography

```
MS<p>
```

projection

```
p ▷ MS<p>
```

merge sort revisited

choreography

```
if p.is_small then 0
else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
     MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
p ▷ if is_small then 0
    else start (q1 ▷ p?id; MS<q1>; p!*);
         start (q2 ▷ p?id; MS<q2>; p!*);
         q1!split1; q2!split2; q1?id; q2?merge
```

merge sort revisited

choreography

```
p start q1,q2; p.split1 -> q1; p.split2 -> q2;
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
p ▷ start (q1 ▷ p?id; MS<q1>; p!*);
    start (q2 ▷ p?id; MS<q2>; p!*);
    q1!split1; q2!split2; q1?id; q2?merge
```

merge sort revisited

choreography

```
p.split1 -> q1; p.split2 -> q2;
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q1!split1; q2!split2; q1?id; q2?merge
q1 ▷ p?id; MS<q1>; p!*
q2 ▷ p?id; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split1 -> q1 ; p.split2 -> q2;
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q1!split1 ; q2!split2; q1?id; q2?merge
q1 ▷ p?id ; MS<q1>; p!*
q2 ▷ p?id; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split2 -> q2;
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q2!split2; q1?id; q2?merge
q1 ▷ MS<q1>; p!*
q2 ▷ p?id; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split2 -> q2 ;
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q2!split2 ; q1?id; q2?merge
q1 ▷ MS<q1>; p!*
q2 ▷ p?id ; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split2 -> q2;
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q2!split2; q1?id; q2?merge
q1 ▷ MS<q1>; p!*
q2 ▷ p?id; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split2 -> q2 ; if q1.is_small then 0
else q1 start q11,q12; q1.split1 -> q11; q1.split2 -> q12;
     MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge
MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q2!split2 ; q1?id; q2?merge
q1 ▷ if is_small then 0
     else start (q11 ▷ ...); start (q12 ▷ ...); ...
     p!*
q2 ▷ p?id ; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split2 -> q2;
q1 start q11, q12; q1.split1 -> q11; q1.split2 -> q12;
MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge
MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q2!split2; q1?id; q2?merge
q1 ▷ start (q11 ▷ ...); start (q12 ▷ ...);
     q11!split1; q12!split2; q11?id; q12?merge; p!*


q2 ▷ p?id; MS<q2>; p!*
```

merge sort revisited

choreography

```
p.split2 -> q2 ;
q1.split1 -> q11 ; q1.split2 -> q12 ;
MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge
MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p  ▷ q2!split2 ; q1?id; q2?merge
q1  ▷ q11!split1 ; q12!split2
q11 ▷ q1?id ; MS<q11>; q1!*
q12 ▷ q1?id ; MS<q12>; q1!*
q2  ▷ p?id ; MS<q2>; p!*
```

merge sort revisited

choreography

```
MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge
MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
 p ▷ q1?id; q2?merge
q1 ▷ q11?id; q12?merge; p!*
q11 ▷ MS<q11>; q1!*
q12 ▷ MS<q12>; q1!*
q2 ▷ MS<q2>; p!*
```

merge sort revisited

choreography

```
MS<q11> ; MS<q12> ; q11.* -> q1; q12.* -> q1.merge
MS<q2> ; q1.* -> p; q2.* -> p.merge
```

projection

```
 p  ▷ q1?id; q2?merge
q1  ▷ q11?id; q12?merge; p!*
q11 ▷ MS<q11> ; q1!*
q12 ▷ MS<q12> ; q1!*
q2  ▷ MS<q2> ; p!*
```

other examples

see our forte'16 paper

- gaussian elimination
  - naive implementation gives pipelined communication and computation
- fast fourier transform
  - naive implementation using an orchestrator

$\rightsquigarrow$ these examples use simple language extensions

outline

conclusions

- a minimalist choreography language including key primitives
  - procedure definition
  - runtime process spawning
  - name mobility
- type system (also) keeping track of connections between processes
  - decidable type checking
  - decidable type inference
- easy to extend for practical applications
- not included: asynchronous semantics (see our ice'17 paper)

thank you!