

---

# Cours de ReactiveML

---

Louis Mandel

Laboratoire de Recherche en Informatique  
Université Paris-Sud 11  
INRIA Saclay – Ile-de-France  
ANR-08-EMER-010

## Les systèmes réactifs

elip

Caractéristiques des systèmes que nous voulons programmer :

- ▶ pas de contraintes temps réel
- ▶ beaucoup de **communications et de synchronisations**
- ▶ beaucoup de **concurrence**
- ▶ **création dynamique** de processus

# ReactiveML

---

Extension d'un langage généraliste (Ocaml\*)

- ▶ structures de données
- ▶ structures de contrôle

Modèle de concurrence simple et déterministe

- ▶ composition parallèle
- ▶ communications entre processus

Compilé vers du code Ocaml

- ▶ générateur de bytecode et de code natif
- ▶ exécutif efficace, glaneur de cellule (GC)

\* sans objets, foncteurs, labels, variants polymorphes, ...

## Plan

---

1. Programmer en ReactiveML
2. Programmer ReactiveML

```
let plateforme centre rayon alpha_init vitesse =  
  let alpha = ref alpha_init in  
  while true do  
    alpha := move !alpha;  
    draw centre rayon !alpha;  
  done
```

```
let plateforme centre rayon alpha_init vitesse =  
  let alpha = ref alpha_init in  
  while true do  
    alpha := move !alpha;  
    draw centre rayon !alpha;  
  done
```

```
let main =  
  Thread.create (plateforme c1 r a1) vitesse;  
  Thread.create (plateforme c2 r a2) vitesse
```

```
let plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    Thread.yield()
  done

let main =
  Thread.create (plateforme c1 r a1) vitesse;
  Thread.create (plateforme c2 r a2) vitesse
```

```
let plateforme centre rayon alpha_init vitesse m1 m2 =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    Mutex.unlock m2; Mutex.lock m1
  done

let main =
  let m1, m2 = Mutex.create (), Mutex.create () in
  Mutex.lock m1; Mutex.lock m2;
  Thread.create (plateforme c1 r a1) vitesse m1 m2;
  Thread.create (plateforme c2 r a2) vitesse m2 m1
```

## Synchrone/Asynchrone

---

```
let barriere n =
  let mutex, attente = Mutex.create (), Mutex.create () in
  Mutex.lock attente;
  let nb_att = ref 0 in
  fun () ->
    Mutex.lock mutex;
    incr nb_att;
    if !nb_att = n then begin
      for i = 1 to n-1 do Mutex.unlock attente done;
      nb_att := 0; Mutex.unlock mutex
    end else begin
      Mutex.unlock mutex; Mutex.lock attente
    end
end
```

## Synchrone/Asynchrone

plateforme5, plateforme5\_bis

---

```
let stop = barriere 3

let plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    stop ()
  done

let main =
  Thread.create (plateforme c1 r a1) vitesse;
  Thread.create (plateforme c2 r a2) vitesse;
  Thread.create (plateforme c3 r a3) vitesse
```

```
let process plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    pause
  done

let process main =
  run (plateforme c1 r a1 vitesse)
|| run (plateforme c2 r a2 vitesse)
|| run (plateforme c3 r a3 vitesse)
```

---

## Le modèle réactif synchrone

---

### Caractéristiques

- ▶ Instants logiques
- ▶ Composition parallèle synchrone
- ▶ Diffusion instantanée d'événements
- ▶ Création dynamique de processus

### Origines

- ▶ Esterel [G. Berry & *al.* 1983]
- ▶ ReactiveC [F. Boussinot 1991]
- ▶ SL [F. Boussinot & R. de Simone 1996]

### Autres langages :

- ▶ SugarCubes, Simple, Fair Threads, Loft, FunLoft, Lurc, S-pi, ...

Déclaration de processus :

- ▶ `let process <id> { <pattern> } = <expr>`

Expressions de base :

- ▶ coopération : `pause`
- ▶ exécution : `run <expr>`

Composition :

- ▶ séquentielle : `<expr> ; <expr>`
- ▶ parallèle : `<expr> || <expr>`

Déclaration d'un signal :

- ▶ `signal <id>`

Émission d'un signal :

- ▶ `emit <signal>`

Statut d'un signal :

- ▶ attente : `await [ immediate ] <signal>`
- ▶ test de présence : `present <signal> then <expr> else <expr>`

## Causalité à la Boussinot

---

Problème de causalité:

- ▶ incohérence logique sur le statut d'un signal :  
au cours d'un instant, un signal doit être : soit présent, soit absent !

- ▶ en Esterel :

```
signal s in
  present s then nothing else emit s end;
end
```

- ▶ en ReactiveML :

```
signal s in
  present s then () else emit s
```

le retard de la réaction à l'absence supprime les problèmes de causalité

## Signaux valués

---

Émission de valeurs sur les signaux :

- ▶ `emit <signal> <value>`

Déclaration de signaux :

- ▶ `signal <id> default <value> gather <function>`
- ▶ type des signaux : ('a, 'b) event
- ▶ type de la valeur par défaut : 'b
- ▶ type de la fonction de combinaison : 'a -> 'b -> 'b

Réception de valeurs sur les signaux :

- ▶ `await <signal> (patt) in <expr>`
- ▶ utilisation à l'instant suivant : absence de problèmes de causalité



Délai avant la récupération de la valeur d'un signal

► En Esterel :

```
signal s := 0 : combine integer with + in
  emit s(1);
  var x := ?s: integer in
    emit s(x)
  end
end
```

## Fonctions de combinaison

---

```
signal s1 default [] gather (fun x y -> x :: y);;
val s1 : ('_a, '_a list) event
```

```
signal s2 default 0 gather (+);;
val s2 : (int , int) event
```

```
signal s3 default 0 gather (fun x y -> x);;
val s3 : (int , int) event
```

Remarque :

► déterminisme si la fonction de combinaison est associative et commutative

## Cas particulier

---

Attendre une seule valeur :

- ▶ exemple : `await s (x :: _) in print_int x`
- ▶ `await [ immediate ] one <signal> (< Patt >) in <expr>`

Garantir l'émission unique :

- ▶ dynamiquement :

```
signal s5 default None gather
  (fun x y ->
    match y with
    | None -> Some x
    | Some _ -> assert false);;
val s5 : ('a, 'a option) event
```
- ▶ statiquement : [Amadio et Dogguy 08]

## Création dynamique de plates-formes

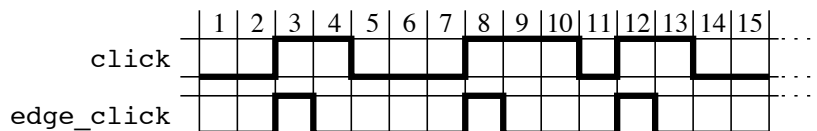
---

```
let process read_click click =
  loop
    if Graphics.button_down() then emit click (Graphics.mouse_pos());
  pause
end
val read_click : ((int * int) , 'a) event -> unit process
```

```
let rec process add new_bal =
  await one new_bal(x,y) in
  run (plateforme (float x, float y) 150. 0. vitesse)
  ||
  run (add new_bal)
val add : ((int * int), (int * int) list) event -> unit process
```

## Création dynamique de plates-formes

```
let process edge click edge_click =
  await immediate one click(pos) in
  emit edge_click pos;
  loop
    present click then pause
  else
    await immediate one click(pos) in
    emit edge_click pos
  end
val edge : ('a, 'a list) event -> ('a, 'b) event -> unit process
```



```
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre

let rec process iter_largeur f a =
  pause;
  match a with
  | Vide -> ()
  | Noeud (x, g, d) ->
    f x;
    (run (iter_largeur f g) || run (iter_largeur f d))
val iter_largeur : ('a -> 'b) -> 'a arbre -> unit process
```

## Parcours d'arbres

---

```
let rec process mem x a =
  pause;
  match a with
  | Vide -> false
  | Noeud (y, g, d) ->
    if x = y then true
    else
      let b1 = run (mem x g)
      and b2 = run (mem x d) in
      b1 or b2
val mem : 'a -> 'a arbre -> bool process
```

Préemption

- ▶ `do <expr> until <signal> done`
- ▶ `do <expr> until <signal> -> <expr> done`
- ▶ `do <expr> until <signal>(<patt>) -> <expr> done`

## Causalité à la Boussinot

---

Délai avant l'exécution de la continuation d'une préemption faible

- ▶ Esterel :

```
signal s1, s2, k in
  weak abort
    await s1;
    emit s2
  when k do emit s1; end weak abort;
end
```

## Parcours d'arbres

---

```
let rec process mem x a =
  pause;
  match a with
  | Vide -> false
  | Noeud (y, g, d) ->
    if x = y then true
    else
      let b1 = run (mem x g)
      and b2 = run (mem x d) in
      b1 or b2
val mem : 'a -> 'a arbre -> bool process
```

## Parcours d'arbres

---

```
let rec process mem_aux x a ok =
  pause;
  match a with
  | Vide -> ()
  | Noeud (y, g, d) ->
    if x = y then emit ok
    else
      let b1 = run (mem_aux x g ok)
      and b2 = run (mem_aux x d ok) in
      ()
val mem_aux : 'a -> 'a arbre -> (unit, 'b) event -> unit process
```

## Parcours d'arbres

---

```
let process mem x a =
  signal ok in
  do
    run (mem_aux x a ok);
    pause; false
  until ok -> true done
val mem : 'a -> 'a arbre -> bool process
```

Remarque :

```
let mem_aux x a ok =
  iter_largeur (fun y -> if x = y then emit ok) a
val mem_aux : 'a -> 'a arbre -> (unit , 'b) event -> unit process
```

## Parcours d'arbres

---

```
let assoc_aux x a ok =
  iter_largeur (fun (y,v) -> if x = y then emit ok v) a
val assoc_aux :
  'a -> ('a * 'b) arbre -> ('b, 'c) event -> unit process
```

```
let process assoc x a =
  signal ok in
  do
    run (assoc_aux x a ok);
    pause; []
  until ok (x) -> x done
val assoc : 'a -> ('a * 'b) arbre -> 'b list process
```

### Suspension

- ▶ condition d'activation : `do <expr> when <signal> done`
- ▶ interrupteur : `control <expr> with <signal> done`



## Glonemo (Ludovic Samper)

---

Étude de cas :

- ▶ Application : détection d'un nuage toxique
- ▶ Environnement : un nuage qui se déplace sous l'influence du vent
- ▶ Routage : diffusion dirigée
- ▶ Protocole MAC : un protocole à échantillonnage de préambule
- ▶ Matériel : processeur et radio basse consommation

Simulateur pour l'étude de la consommation d'énergie dans les réseaux de capteurs

- ▶ Modélisation des nœuds
- ▶ Modèle de l'environnement en Lucky [Jahier & Raymond]

## ReactiveML : outils

---

Compilateur :

- ▶ `rmlc`
- ▶ génération de code Ocaml

Toplevel :

- ▶ `rmltop`
- ▶ équivalent de la commande `ocaml`

Calcul des dépendances :

- ▶ `rmldep`
- ▶ équivalent de la commande `ocamldep`

<http://rml.lri.fr>

Journées Francophones des Langages Applicatifs 2010

---

# Cours de ReactiveML

---

Louis Mandel

Laboratoire de Recherche en Informatique  
Université Paris-Sud 11  
INRIA Saclay – Ile-de-France  
ANR-08-EMER-010

ReactiveML

---

Sémantique

# Le noyau ReactiveML

---

```
e ::= x | c | (e, e) | λx.e | e e | rec x=e | process e
    | let x = e and x = e in e | pause | run e
    | signal x default e gather e in e
    | present e then e else e | emit e e | pre e | pre ?e
    | do e until e(x) -> e done | do e when e
c ::= true | false | () | 0 | ... | + | - | ...
```

Opérateurs dérivés

$$e_1 || e_2 \stackrel{def}{=} \text{let } x_1 = e_1 \text{ and } x_2 = e_2 \text{ in } ()$$

...

## Sémantiques statiques

---

Analyse d'instantanéité

► exemple :

```
let f x =
  let y = x + 1 in
  pause;
  print_int y
incorrect
```

```
let process f x =
  let y = x + 1 in
  pause;
  print_int y
correct
```

Typage

► Extension conservative du typage de ML

$$\frac{H \vdash e_1 : (\tau_1, \tau_2) \text{ event} \quad H \vdash e_2 : \tau_1}{H \vdash \text{emit } e_1 e_2 : \text{unit}} \quad \dots$$

## Sémantiques dynamiques

---

Sémantique comportementale (“grands pas”)

- ▶ qu'est ce qu'une réaction valide ?
- ▶ abstraction de l'ordonnement à l'intérieur d'un instant

$$N \vdash e \xrightarrow[S]{E, b} e'$$

Sémantique opérationnelle (“petits pas”)

- ▶ comment obtenir une réaction valide ?
- ▶ description de tous les ordonnancements possibles

$$e/S_0 \rightarrow e_1/S_1 \rightarrow \dots \rightarrow e_n/S \xrightarrow{eoi} e'$$

## La sémantique comportementale

---

Forme des réductions

$$N \vdash e \xrightarrow[S]{E, b} e'$$

- ▶  $N$  ensemble des noms de signaux  $n$  créés par la réaction de  $e$
- ▶  $E$  signaux émit par la réaction de  $e$
- ▶  $S$  environnement de signaux dans lequel  $e$  doit réagir
- ▶  $b$  statut de terminaison

Comme pour Esterel, nous avons l'invariant  $E \sqsubseteq S$ .

## Sémantique Comportementale

---

Exemple de règles

$$\emptyset \vdash v \xrightarrow[S]{\emptyset, true} v \quad \emptyset \vdash \text{pause} \xrightarrow[S]{\emptyset, false} ()$$

$$\frac{N_1 \vdash e \xrightarrow[S]{E, true} n \quad n \in S \quad N_2 \vdash e_1 \xrightarrow[S]{E_1, b} e'_1}{N_1 \cdot N_2 \vdash \text{present } e \text{ then } e_1 \text{ else } e_2 \xrightarrow[S]{E \sqcup E_1, b} e'_1}$$

$$\frac{N \vdash e \xrightarrow[S]{E, true} n \quad n \notin S}{N \vdash \text{present } e \text{ then } e_1 \text{ else } e_2 \xrightarrow[S]{E, false} e_2}$$

## Sémantique Comportementale

---

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, b_1} e'_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, b_2} e'_2 \quad b_1 \wedge b_2 = false}{N_1 \cdot N_2 \vdash e_1 \parallel e_2 \xrightarrow[S]{E_1 \sqcup E_2, false} e'_1 \parallel e'_2}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, true} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, true} v_2}{N_1 \cdot N_2 \vdash e_1 \parallel e_2 \xrightarrow[S]{E_1 \sqcup E_2, true} ()}$$

⇒ L'environnement  $S$  est global.

## Sémantique opérationnelle

---

La sémantique opérationnelle se décompose en 3 étapes :

- ▶ réaction pendant l'instant

$$e/S \rightarrow^* e'/S'$$

- ▶ calcul des sorties

$$O = next(S)$$

- ▶ réaction de fin d'instant

$$O \vdash e' \rightarrow_{eoi} e''$$

## Sémantique opérationnelle

---

Réduction en tête de terme

$$(\lambda x.e) v / S \rightarrow_{\varepsilon} e[x \leftarrow v] / S \qquad \mathbf{emit} \ n \ v / S \rightarrow_{\varepsilon} () / S + [v/n]$$

$$\mathbf{present} \ n \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 / S \rightarrow_{\varepsilon} e_1 / S \ \text{si} \ n \in S \quad \dots$$

Contextes

$$\Gamma ::= [] \mid \Gamma; e \mid \mathbf{present} \ \Gamma \ \mathbf{then} \ e \ \mathbf{else} \ e \\ \mid \mathbf{let} \ x = \Gamma \ \mathbf{and} \ x = e \ \mathbf{in} \ e \mid \mathbf{let} \ x = e \ \mathbf{and} \ x = \Gamma \ \mathbf{in} \ e \mid \dots$$

$$\frac{e/S \rightarrow_{\varepsilon} e'/S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} \qquad \frac{n \in S \quad e/S \rightarrow e'/S'}{\Gamma(\mathbf{do} \ e \ \mathbf{when} \ n)/S \rightarrow \Gamma(\mathbf{do} \ e' \ \mathbf{when} \ n)/S'}$$

## Sémantique opérationnelle

---

Fin d'instant

$$\frac{n \notin O}{O \vdash \text{present } n \text{ then } e_1 \text{ else } e_2 \rightarrow_{eoi} e_2}$$

$$O \vdash \text{pause} \rightarrow_{eoi} ()$$

...

avec  $O = \text{next}(S)$

## Propriétés

---

Sémantique comportementale

- ▶ déterministe

Sémantique opérationnelle

- ▶ preuve de sûreté du typage

Sémantiques comportementale et opérationnelle

- ▶ équivalence entre les deux sémantiques
- ▶ absence de problème de causalité

---

## Implantation de ReactiveML

### **Les clés d'un interprète efficace : l'attente passive**

---

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```



## Les clés d'un interprète efficace : l'attente passive

---

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

→  
s0

```
await immediate s1 || await immediate s0; emit s1
```

## Les clés d'un interprète efficace : l'attente passive

---

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

→  
s0

```
await immediate s1 || await immediate s0; emit s1
```

→  
s0, s1

```
await immediate s1
```

## Les clés d'un interprète efficace : l'attente passive

---

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

→  
s0

```
await immediate s1 || await immediate s0; emit s1
```

→  
s0, s1

```
await immediate s1 → s0, s1 → ()
```

⇒ Il faut réactiver une instruction uniquement lorsque le signal dont elle dépend est émis : utilisation de files d'attente

## Les clés d'un interprète efficace

---

D'autres points clés :

- ▶ Exécution du code Ocaml sans surcoût
- ▶ Gestion efficace des signaux
  - ▷ accès en temps constant
  - ▷ allocation/désallocation automatique
- ▶ ...

## Sémantique et implantation sans suspension ni préemption

---

$L_k$  : un langage à base de continuations

- ▶ traduction de ReactiveML vers  $L_k$  :  $C_k[e_1; e_2] = C_{(C_k[e_2])}[e_1] \quad \dots$
- ▶ exemple :

```
let nat k =  
  fun _ ->  
    (let cpt = ref 0 in  
     Lk_record.rml_loop  
     (fun k' ->  
       Lk_record.rml_compute (fun () -> print_int !cpt; ...)  
       (Lk_record.rml_pause k'))  
     ()))
```

## Sémantique de $L_k$

---

Sémantique gloutonne

- ▶ toujours aller de l'avant
- ▶ représentation du programme
  - ▷  $\mathcal{C}$  ensemble des expressions à exécuter instantanément
  - ▷  $\mathcal{W}$  ensemble des expressions en attente d'un signal
  - ▷  $\mathcal{J}$  ensemble des points de synchronisation

Exécution d'une étape de réaction

$$S, J, \mathcal{W} \vdash \langle e, v \rangle \longrightarrow S', J', \mathcal{W}' \vdash \mathcal{C}$$

- ▶  $e$  expression à exécuter
- ▶  $v$  valeur précédente

## Implantation en OCaml

---

Les règles de la sémantique  $L_k$  peuvent se traduire quasiment directement en des fonctions de transition de type :

$$\begin{aligned} \text{step} &= \text{env} \times \text{value} \rightarrow \text{env} \\ \text{env} &= \text{signal\_env} \times \text{join} \times \text{waiting} \times \text{current} \end{aligned}$$

En implantant l'environnement directement dans le tas, les fonctions de transitions ont le type Ocaml suivant :

```
type 'a step = 'a -> unit
```

## Implantation en OCaml : compute

---

$$\frac{e/S \Downarrow v'/S'}{S, J, \mathcal{W} \vdash \langle e.k, v \rangle \longrightarrow S', J, \mathcal{W} \vdash \langle k, v' \rangle}$$

## Implantation en OCaml : compute

---

$$\frac{e/S \Downarrow v'/S'}{S, J, \mathcal{W} \vdash \langle e.k, v \rangle \longrightarrow S', J, \mathcal{W} \vdash \langle k, v' \rangle}$$

La fonction de transition compute est définie par :

```
let compute e k =  
  fun v ->  
    let v' = e() in  
    k v'  
val compute : (unit -> 'a) -> 'a step -> 'b step
```

## Implantation en OCaml : await/immediate

---

$$\frac{e/S \Downarrow n/S' \quad n \in S'}{S, J, \mathcal{W} \vdash \langle \text{await immediate } e.k, v \rangle \longrightarrow S', J, \mathcal{W} \vdash \langle k, () \rangle}$$
$$\frac{e/S \Downarrow n/S' \quad n \notin S' \quad \text{self} = \text{await immediate } n.k}{S, J, \mathcal{W} \vdash \langle \text{await immediate } e.k, v \rangle \longrightarrow S', J, \mathcal{W} + [\langle \text{self}, v \rangle / n] \vdash \emptyset}$$

## Implantation en OCaml : await/immediate

---

$$\frac{e/S \Downarrow n/S' \quad n \in S'}{S, J, \mathcal{W} \vdash \langle \text{await immediate } e.k, v \rangle \longrightarrow S', J, \mathcal{W} \vdash \langle k, () \rangle}$$
$$\frac{e/S \Downarrow n/S' \quad n \notin S' \quad \text{self} = \text{await immediate } n.k}{S, J, \mathcal{W} \vdash \langle \text{await immediate } e.k, v \rangle \longrightarrow S', J, \mathcal{W} + [\langle \text{self}, v \rangle / n] \vdash \emptyset}$$

```
let await_immediate e k =
  fun v ->
    let (n, w) = e() in
    let rec self () =
      if Event.status n then k ()
      else w := self :: !w
    in self ()
val await_immediate : (unit -> ('a, 'b) event) -> unit step -> 'c step
```

## Implantation en OCaml : emit

---

```
let emit e1 e2 k =
  fun v ->
    let (n, w) = e1() in
    let v' = e2() in
    Event.emit n v';
    current := !w @ !current;
    !w := [];
    k ()
val emit :
  (unit -> ('a, 'b) event) -> (unit -> 'a) -> unit step
  -> 'c step
```

## Sémantique de $L_k$

---

Les suspensions et préemptions ?

- ▶ on a perdu la structure du programme !
- ▶ utilisation d'un arbre de contrôle

## Bibliothèque pour la programmation réactive

---

```
val rml_compute: (unit -> 'a) -> 'a expr
val rml_seq: 'a expr -> 'b expr -> 'b expr
val rml_par: 'a expr -> 'b expr -> unit expr
...
```

L'expression ReactiveML :

```
(await s1 || await s2); emit s3
```

se traduit en Ocaml par :

```
rml_seq
  (rml_par
    (rml_await (fun () -> s1))
    (rml_await (fun () -> s2)))
  (rml_emit (fun () -> s3)))
```

## Toplevel

### rmltop : le toplevel ReactiveML

---

Basé sur l'idée des Reactive Scripts [Boussinot & Hazard 96]

Utile pour :

- ▶ comprendre le modèle réactif
- ▶ faire des expériences de reconfiguration dynamique
- ▶ concevoir des systèmes réactifs



## Démo

---

Glonemo

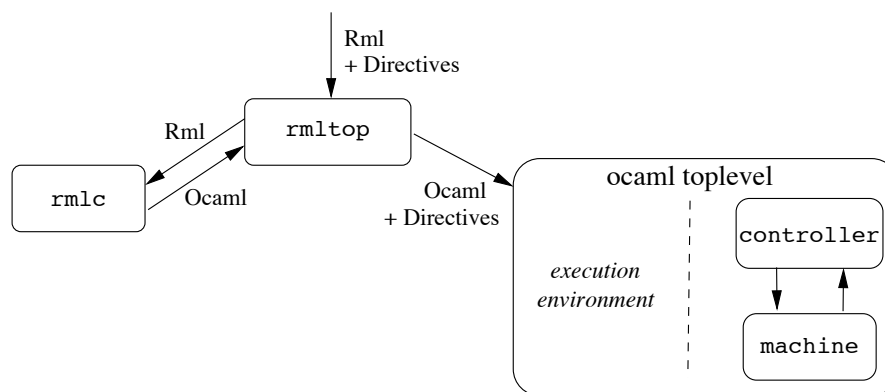
- ▶ `rmltop graphics.cma glonemo.cma`

n-Corps

- ▶ <http://rml.lri.fr/rmltop>

## Implantation

---



contrôleur implémenté en ReactiveML

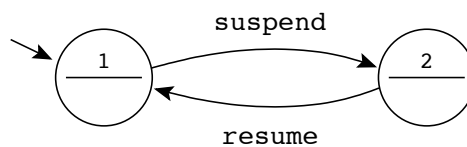
## Contrôleur

---

```
let process sampled =  
  loop Rmltop_reactive_machine.rml_react(get_to_run()); pause end  
  
let process step_by_step =  
  loop  
    await step(n) in  
    do  
      for i = 1 to n do  
        Rmltop_reactive_machine.rml_react(get_to_run()); pause  
      done  
    until suspend done  
  end
```

## Contrôleur

---



```
let process machine_controller =  
  loop  
    do run sampled until suspend done;  
    do run step_by_step until resume done  
  end
```

## Reconfiguration dynamique

### Langage pour étudier la reconfiguration dynamique

Des combinateurs pour manipuler (individuellement) des processus en cours d'exécution

- ▶ tuer
- ▶ suspendre/repandre
- ▶ ajouter des branches parallèles supplémentaires
- ▶ ...

Facilement programmable en ReactiveML

- ▶ utilisation de l'ordre supérieur et du polymorphisme

```
signal kill
val kill : (int, int list) event

let process killable p =
  let id = gen_id () in print_endline ("["^(string_of_int id)^"];");
  do run p
  until kill(ids) when List.mem id ids done
val killable : unit process -> unit process
```

## Création dynamique : rappel

---

```
let rec process extend to_add =
  await to_add(p) in
  run p || run (extend to_add)
val extend : ('a, 'b process) event -> unit process

signal to_add
  default process ()
  gather (fun p q -> process (run p || run q))
val add_to_me : (unit process, unit process) event
```

## Création dynamique avec état

---

```
let rec process extend to_add state =
  await to_add(p) in
  run (p state) || run (extend to_add state)
val extend : ('a , ('b -> 'c process)) event -> 'b -> unit process

signal to_add
  default (fun s -> process ())
  gather (fun p q s -> process (run (p s) || run (q s)))
val to_add : (('state -> unit process) , ('state -> unit process)) event
```

## extensible

---

```
signal add
val add : ((int * (state -> unit process)),
           (int * (state -> unit process)) list) event

let process extensible p_init state =
  let id = gen_id () in print_endline ("{"^(string_of_int id)^"}");
  signal add_to_me
    default (fun s -> process ())
    gather (fun p q s -> process (run (p s) || run (q s))) in
  run (p_init state) || run (extend add_to_me state)
  || loop
    await add(ids) in
    List.iter (fun (x,p) -> if x = id then emit add_to_me p) ids
  end
val extensible : (state -> 'a process) -> state -> unit process
```

## Bibliothèque pour le toplevel

---

```
type ident
```

```
val kill: (int , ident list) event
```

```
val killable: 'a process -> 'a option process
```

```
val sr: (int , ident list) event
```

```
val suspendable: 'a process -> 'a process
```

```
val extensible:
```

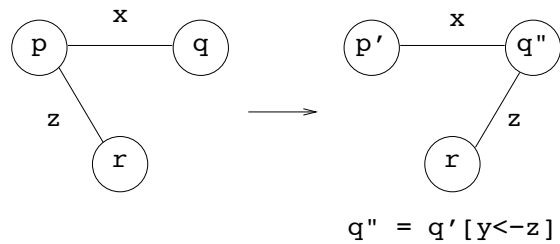
```
  ('a, (int * ('state -> unit process)) list) event ->
```

```
  ('state -> unit process) -> 'state -> unit process
```

```
val ps: unit -> unit
```

## Mobilité du $\pi$ -calcul

---

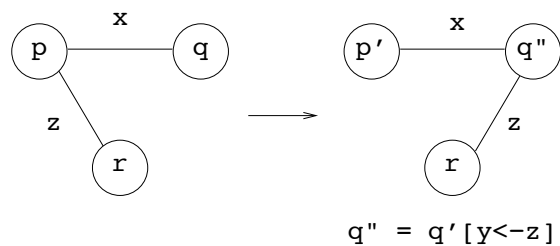


```
let process p x z =  
  emit x z;  
  run (p' x)  
val p : ('a, 'b) event -> 'a -> unit process
```

```
let process q x =  
  await one x(y) in  
  run (q' y)  
val q : ('a, 'a list) event -> unit process
```

## Mobilité du $\pi$ -calcul

---



```
let process r z = ...  
val r : ('a, 'b) event -> unit process
```

```
let process mobility x z =  
  run (p x z) || run (q x) || run (r z)  
val mobility :  
  (('a, 'b) event, ('a, 'b) event list) event ->  
  ('a, 'b) event -> unit process
```

ReactiveML

---

Collaboration entre ReactiveML et JoCaml



## JoCaml

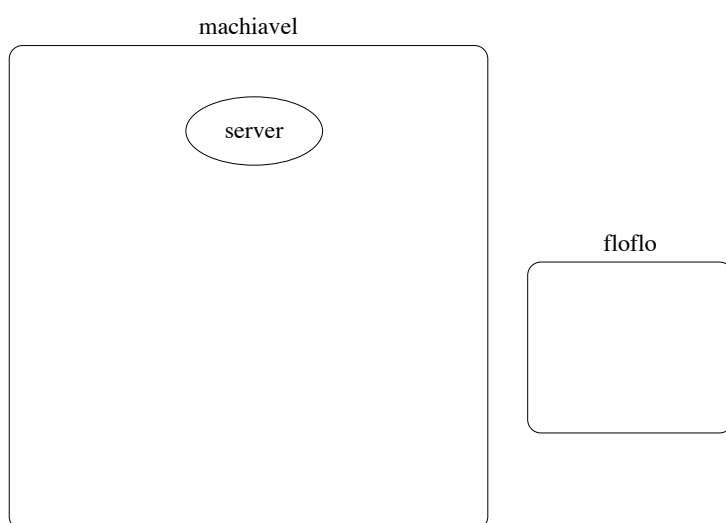
---

Extension de Ocaml basée sur le join-calcul

- ▶ asynchrone
- ▶ exécution distribuée

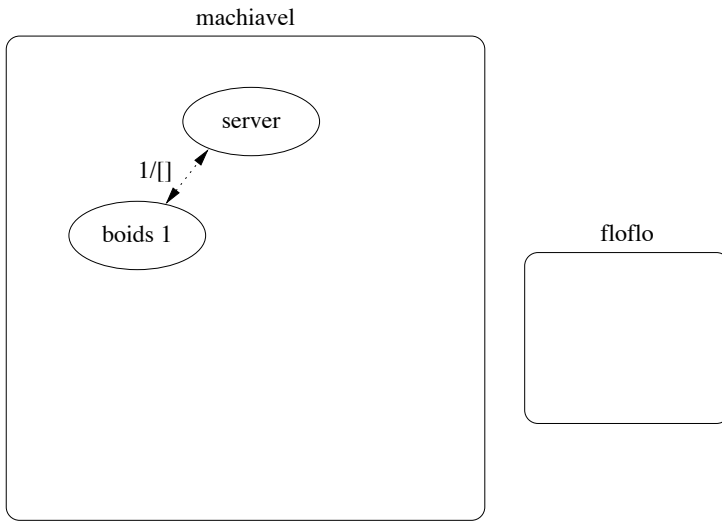
## Boids

---



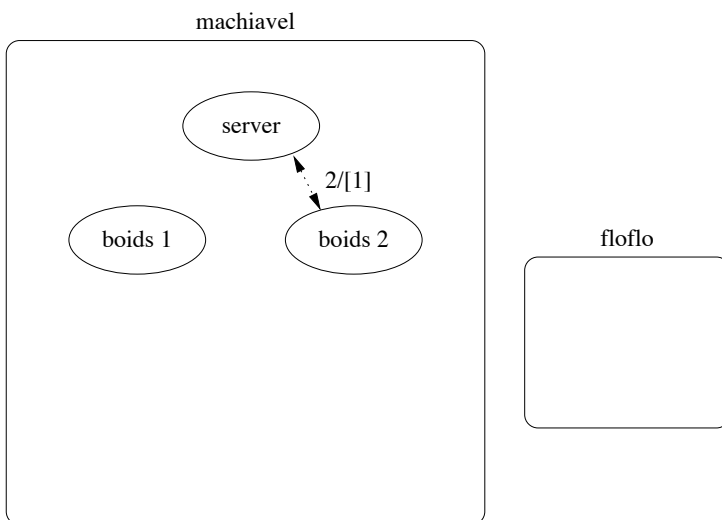
# Boids

---



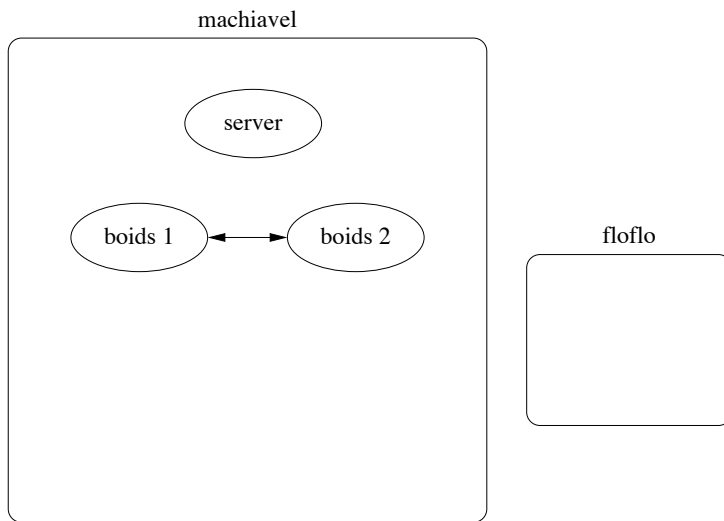
# Boids

---



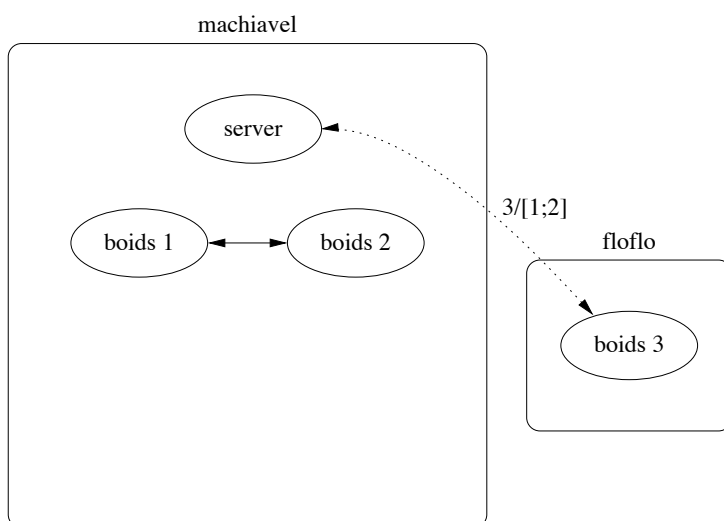
# Boids

---



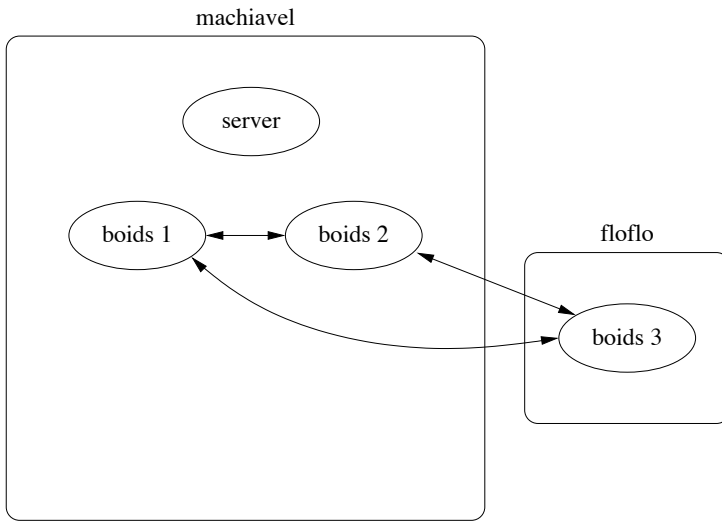
# Boids

---



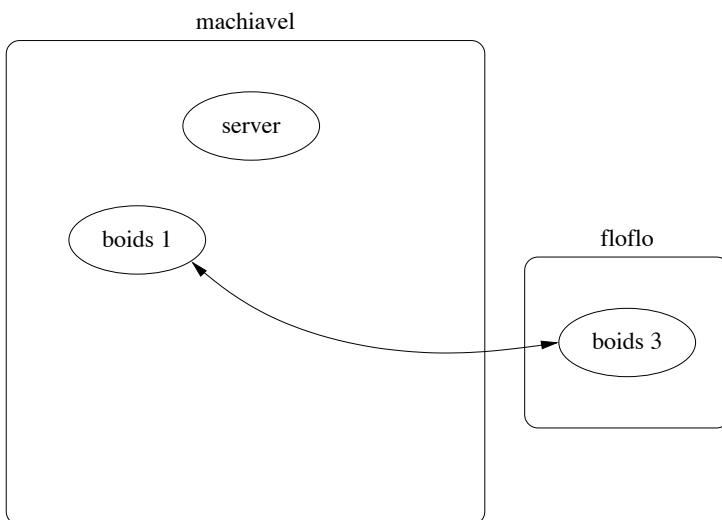
# Boids

---



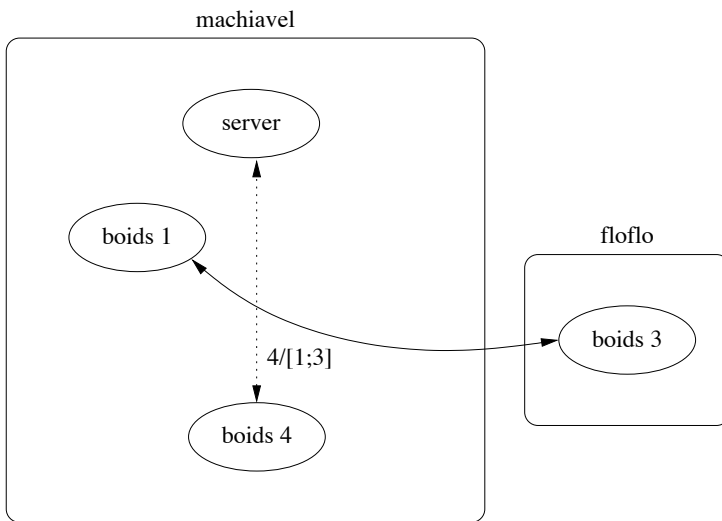
# Boids

---



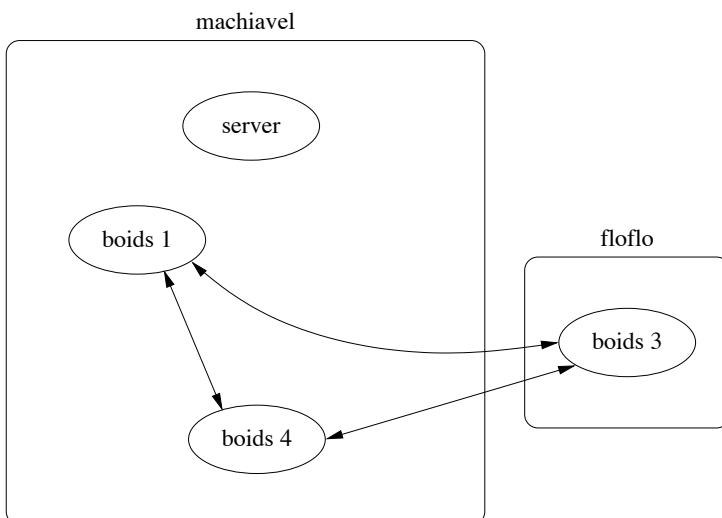
# Boids

---



# Boids

---



`http://rml.lri.fr`