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

Les systèmes réactifs

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

Synchrone/Asynchrone

```
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
```

```
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, ...

ReactiveML : les processus

demo1.rml

Déclaration de processus :

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

Expressions de base :

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

Composition :

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

ReactiveML : les communications

`demo2.rml`, `demo_present.rml`

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

Causalité à la Boussinot

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
```

Création dynamique de plates-formes

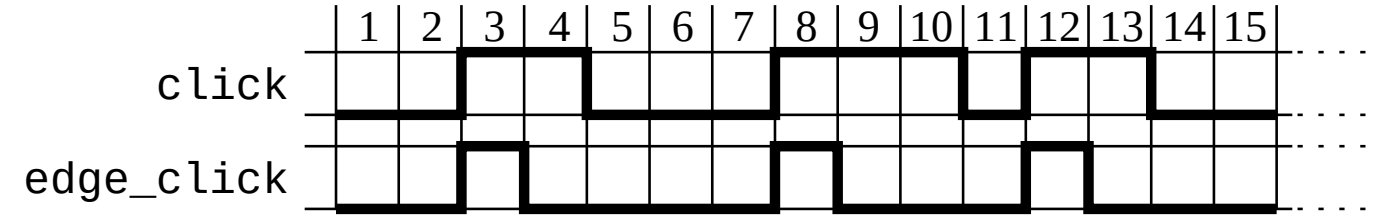
plateforme_sync_bis

```
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



Parcours d'arbres

arbre.rml

```
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
```

Structures de contrôle

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
```

Structures de contrôle

Suspension

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

Exemple d'application

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`