Séminaire MaMux

ReactiveML et aspects dynamiques dans les langages synchrones

Louis Mandel

Université Paris-Sud 11
INRIA Paris-Rocquencourt

03/02/2012

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, ...

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, ...

Présentation du langage

ReactiveML : les processus

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

Déclaration d'un signal :

- `signal` *<id>* `default` *<value>* `gather` *<function>*

Émission d'un signal :

- `emit` *<signal>* *<value>*

Statut d'un signal :

- attente : `await` *<signal>* (*patt*) `in` *<expr>*
- test de présence : `present` *<signal>* `then` *<expr>* `else` *<expr>*

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

```
let rec process add click =
  await click (x,y) in
  run (plateforme (float x, float y) 150. 0. vitesse)
  ||
  run (add click)
```

*val add : ('a, (int * int)) event -> unit process*

ReactiveML : structures de contrôle

Préemption

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

Suspension

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

Placement de plates-formes


```
let process generate_new_plateforme click key new_plateforme =
  loop
    await click (p1) in
    do
      await click (p2) in
      emit new_plateforme (p1, p2)
    until key(Key_ESC) done
  end
```

Programmation événementielle

```
class generate_new_plateforme = object(self)
  val mutable state = 0
  val mutable last_clock = (0, 0)

  method on_click pos =
    match state with
    | 0 -> last_click <- pos;
           state <- 1
    | 1 -> emit new_plateforme (last_click, pos);
           state <- 0

  method on_key_down k =
    match k with
    | Key_ESC -> state <- 0
    | _ -> ()
end
```

Reconfiguration dynamique

`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

```
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
```

Parallélisme

ReactiveML et parallélisme

Exécution parallèle de programmes ReactiveML (Cédric Pasteur)

- simulation à grande échelle

Tâches asynchrones (Mehdi Dogguy)

- entrées/sorties asynchrones

Systèmes Globalement Asynchrones et Globalement Synchrones

- mélange ReactiveML et JoCaml

Conclusion

http://rml.lri.fr