
rmltop

ou

la définition interactive de programmes réactifs

Louis Mandel Florence Plateau Marc Pouzet

{mandel,plateau,pouzet}@lri.fr

Laboratoire de Recherche en Informatique, équipe Démons

Université Paris-Sud 11

Partout – 09/01/2009

ReactiveML

Langage pour la programmation de systèmes interactifs

- ▶ extension d'un langage généraliste (OCaml)
- ▶ basé sur le modèle synchrone
- ▶ suppression de la contrainte temps-réel

Motivations

- ▶ Descriptions de systèmes non bornés statiquement
 - ▶ Programmation d'interfaces graphiques, jeux, simulateurs
 - ▶ Intérêts : déterminisme, compositionnalité, sûreté
- ⇒ alternative à la programmation événementielle ou par threads

`rmltop` : Le toplevel ReactiveML

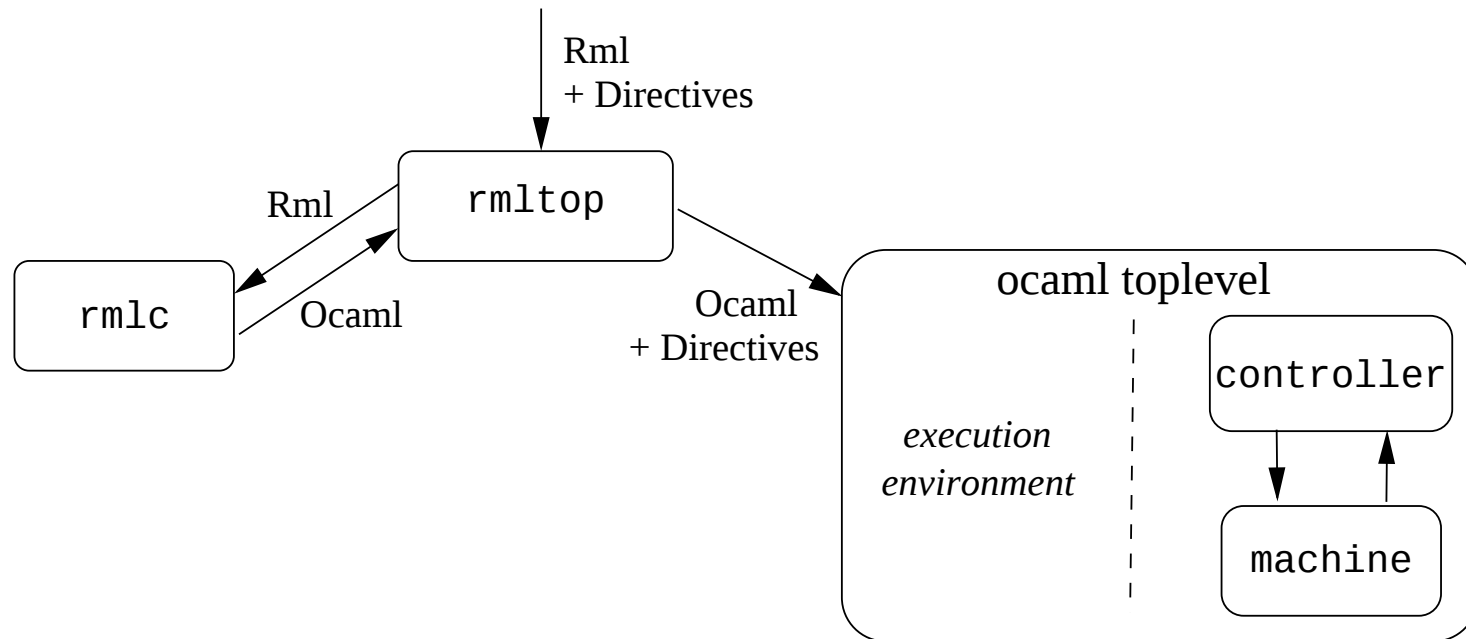
- ▶ Permet la définition et l'exécution interactive de programmes ReactiveML
 - ▶ le pendant ReactiveML du *toplevel* Ocaml
- ▶ Utile pour :
 - ▶ comprendre le modèle réactif
 - ▶ concevoir des systèmes réactifs
 - ▶ faire des expériences de reconfiguration dynamique
- ▶ Basé sur l'idée des REACTIVE SCRIPTS [Boussinot & Hazard 96]

Plan

1. Le toplevel ReactiveML
2. Implantation
3. Vers de la reconfiguration dynamique
4. Extension asynchrone

Implantation

Implantation

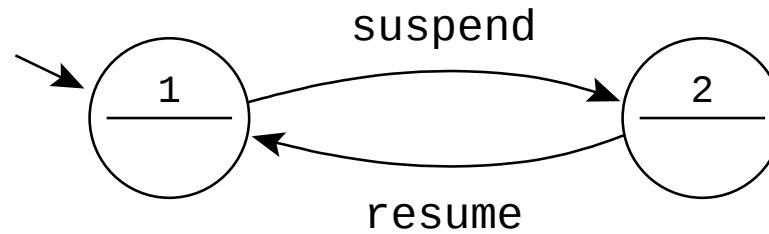


- Le contrôleur est implémenté en ReactiveML.

Contrôler

```
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
```

Contrôler



```
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/reprendre
 - ▶ ajouter des branches parallèles supplémentaires
 - ▶ ...
- ▶ Facilement programmable en ReactiveML
 - ▶ utilisation de l'ordre supérieur et du polymorphisme

killable

```
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
```

Extension asynchrone

Communication asynchrone

```
let new_cell () =  
  def state (x) & set(x) = state(Some x) & reply () to set  
  or state (Some x) & get() = state(None) & reply x to get in  
  spawn (state None);  
  (set, get)  
  
val new_cell : ('a -> unit process, unit -> 'a process)  
  
let set_step, get_step = new_cell()  
let process generate_step =  
  loop let n = run (get_step ()) in emit step n ; pause end
```

Conclusion

- ▶ Utile pour la mise au point et le débogage de systèmes réactifs
- ▶ Absence de problèmes de causalité grâce au modèle réactif de Boussinot
 - ▶ pas de toplevel dans un langage à la Esterel
- ▶ `rmltop` est implémenté en ReactiveML
 - ▶ implantation courte (500 SLOC) et assez simple
 - ▶ la partie asynchrone pourrait être améliorée
- ▶ `rmltop` et `rmlc` sont équivalents :
 - ▶ sémantique
 - ▶ efficacité
- ▶ Un bon outil pour étudier la reconfiguration dynamique

`http://rml.lri.fr`