Time Refinement in a Functional Synchronous Language

Louis Mandel, Cédric Pasteur and Marc Pouzet
Ecole normale supérieure, Paris, France
PPDP’13, Madrid

20 sept. 2013
Synchronous languages

- Discrete logical time

- Programming critical real-time embedded systems (Lustre, Signal, Esterel)

- Discrete simulation (ReactiveC, ReactiveML [Mandel05])
  - one time step = one instant
  - sensor network [Samper06]
Sensor network

Ad-hoc network of small sensors

- CPU, sensors, radio and battery
- eg, detect pollution cloud
Multiple time scales

- microseconds for hardware
- milliseconds for software

Time refinement

- eg, replace an approximation with the composition of several agents
Traditional solution: sampling
Traditional solution: sampling

Reactive domains

- Local time scale
- Subdivide instants
- Invisible from the outside
Summary

Introduction

ReactiveML

Reactive domains

Examples

Clock calculus

Conclusion
Summary

Introduction
ReactiveML
Reactive domains
Examples
Clock calculus
Conclusion
ReactiveML

- Functional language (ML, OCaml)
- Synchronous model of concurrency (Esterel)

Deterministic concurrency in a general purpose language
Level-order traversal

```ml
type 'a tree =
 | Empty
 | Node of 'a tree * 'a * 'a tree

let rec process levelorder f t = match t with
 | Empty -> ()
 | Node (l, v, r) ->
   f v;
   pause;
   (run levelorder f l || run levelorder f r)
```
Signals

- Streams of values
- Instantaneous broadcast
- Multi-emission

```plaintext
let process sig_gather =
  signal s default 0 gather (+) in
  emit s 2
  || emit s 4
  || await s(v) in print_int v
```
Example: n-body

Simulate a solar system

- Simulate the gravitational interactions of n bodies
- Equation:
  \[ m_i \ddot{a}_i = \vec{f}_i = \sum_{j} \vec{F}_{i,j} \]
- Fixed-step numerical methods
Example: n-body

```
let dt = 0.01
signal env default (fun _ -> zero_vector)
    gather add_force

let rec process body (x_t, v_t, w) =
    emit env (force (x_t, w));
    await env(f) in
(* euler semi-implicit method *)
    let v_tp = v_t ++. (dt **. (f x_t)) in
    let x_tp = x_t ++. (dt **. v_tp) in
    run body (x_tp, v_tp, w)

let process main =
    for i = 1 to 100 dopar
        run body (random_planet ())
    done
```
Introduction
ReactiveML
Reactive domains
Examples
Clock calculus
Conclusion
Reactive domains

- Local time scale
- Invisible from the outside

```plaintext
let process stutter =
  domain ck by 6 do
    loop print_string "a"; pause ck end
  done
||
loop print_string "b"; pause global_ck end
```
Reactive domains

- Local time scale
- Invisible from the outside

```plaintext
let process stutter_nested =
  domain ck1 by 3 do
    domain ck2 by 2 do
      loop print_string "a"; pause ck2 end
    done
  done
done
```
Reactive domains and signals

A signal has a clock

- One value per instant of the reactive domain
- Operations are relative to the clock of the signal

```plaintext
let process sig_gather_ck =
  domain ck by 2 do
    signal s default 0 gather (+) in
      emit s 2
      || emit s 4
      || await s(v) in print_int v
  done
```
let process delayed_hello_world = 
  signal s default 0 gather (+) in 
  domain ck by 10 do 
    pause global_ck; emit s 3 
  || 
    await s(v) in print_int v 
  done
let process delayed_hello_world =
    signal s default 0 gather (+) in
    domain ck by 10 do
        pause global_ck; emit s 3
        ||
        await s(v) in print_int v
    done

A reactive domain automatically waits for the next instant of its parent clock if its body is stuck

- By calling pause on slow clock
- By waiting for a slow signal
Automatic waiting

Period can be omitted

- Unbounded number of local instants

Make a process instantaneous

```plaintext
let rec process levelorder f t = match t with
| Empty -> ()
| Node (l, v, r) ->
  f v; pause;
  (run levelorder f l || run levelorder f r)

let process levelorder_inst f t =
  domain ck do
    run levelorder f t
  done
```
Summary

Introduction
ReactiveML
Reactive domains
Examples
Clock calculus
Conclusion
Time refinement

A sensor node

- One instant = one time step = 1 ms

```haskell
let process node state me neighbors = 
  await me(msgs) in
let state = process_msgs state msgs neighbors in
run node state me neighbors
```

Subdividing the time step

```haskell
let process node me neighbors = 
  domain ms do
  signal energy in
  domain us by 1000 do
    run radio energy neighbors
  done
  || run software energy me
  || run battery energy
done
```
Demo: n-body with multiple steps

Multiple steps integration methods

- The computation of the next position is done in multiple steps, shared by all bodies
- Internal steps hidden by a reactive domain
- Can easily and transparently switch between methods
Demo: n-body adaptive

Adaptive integration

- Compute in multiple steps
  - The new positions
  - An estimation of the error
- If the error is too big, try again with a smaller step

Implementation

- Two nested reactive domains
- Still one instant on the global clock
Reactive domains and signals

Two limitations

- Cannot use a signal outside of its domain
- No immediate dependency on slow signal

```haskell
let process immediate_dep_wrong =
  signal s in
  domain ck do
    await immediate s; print_string "Ok"
    ||
    pause ck; emit s
  done
```

Rejected because we make two contradictory assumptions on the presence of s
Clock calculus

Typing rules

In the environment $\Gamma$ and local clock $ce$, we associate to an expression $e$ a type $ct$ and an effect $cf$

$$\Gamma, ce \vdash e : ct \mid cf$$

- Effect = set of clocks of accessed signals
  
  $ce ::= \gamma \mid ck$
  
  $cf ::= \emptyset \mid \emptyset \mid \{ce\} \mid cf \cup cf$

Types

- Function: $ct \xrightarrow{cf} ct$
- Process: $ct \text{process}\{ce|cf\}$
- Signal: $(ct, ct) \text{event}\{ce\}$
Clock calculus (2)

Type-and-effect

\[
\Gamma; x : ct_1, ce \vdash e : ct_2 | cf \\
\Gamma, ce \vdash \lambda x. e : ct_1 \xrightarrow{cf} ct_2 | \emptyset \\
\Gamma, ce \vdash e_1 : ct_2 \xrightarrow{cf} ct_1 | \emptyset \\
\Gamma, ce \vdash e_2 : ct_2 | \emptyset \\
\Gamma, ce \vdash e_1 e_2 : ct_1 | cf
\]

Reactive domains

\[
\Gamma; x : \{\gamma\}, \gamma \vdash e : ct | cf \\
\gamma \not\in ftv(\Gamma, ct) \\
\Gamma, ce \vdash \text{domain } x \text{ do } e : ct | cf \setminus \{\gamma\}
\]
Introduction

ReactiveML

Reactive domains

Examples

Clock calculus

Conclusion
Future work: Clocks and Parallelism

Parallelization is done at the level of reactive domains

- Less synchronizations
- Locality of signals

```
domain ck do
  domain ck2 do
    signal s in run p2 s
  done
  ||
  domain ck3 do run p3 done
done
```
Conclusion

Reactive domains

- Local time scales

Type-and-effect system

- Sound use of signals

Implemented in the ReactiveML compiler

- http://www.reactiveml.org/ppdp13

More details in the paper

- Operational semantics of ReactiveML with domains
- Soundness proof (extended version)


Reactive domains vs oversampling

Oversampling is not modular
  - Go faster by slowing down everybody else

Cannot do two oversamplings in parallel
  - Rejected by clock calculus in Lustre/Lucid Synchrone
  - Cannot generate sequential code in Signal

Cannot make computation instantaneous
  - Only reduce to one instant
A non-cooperative domain

let process nonreactive_domain =
  domain ck do
    loop print_endline "tick"; pause ck end
  done

let process instantaneous_loop =
  loop print_endline "tick" end

Detect non-reactive processes

- Type-and-effect systems
- Behaviours [Amtoft99]
Preventing immediate dependency

- Force the clock of signals to be the local clock

\[
\Gamma, ce \vdash e : (ct_1, ct_2) \text{event}\{ce\} | \emptyset \\
\Gamma, ce \vdash \text{await immediate } e : \text{unit} | \{ce\}
\]