ReactiveML

The temporal expressiveness of synchronous languages
with the power of functional programming.

Louis Mandel Marc Pouzet Cédric Pasteur

Guillaume Baudart, Mehdi Dogguy, Louis Jachiet

What is ReactiveML?

Why?

Programming Reactive Systems with :

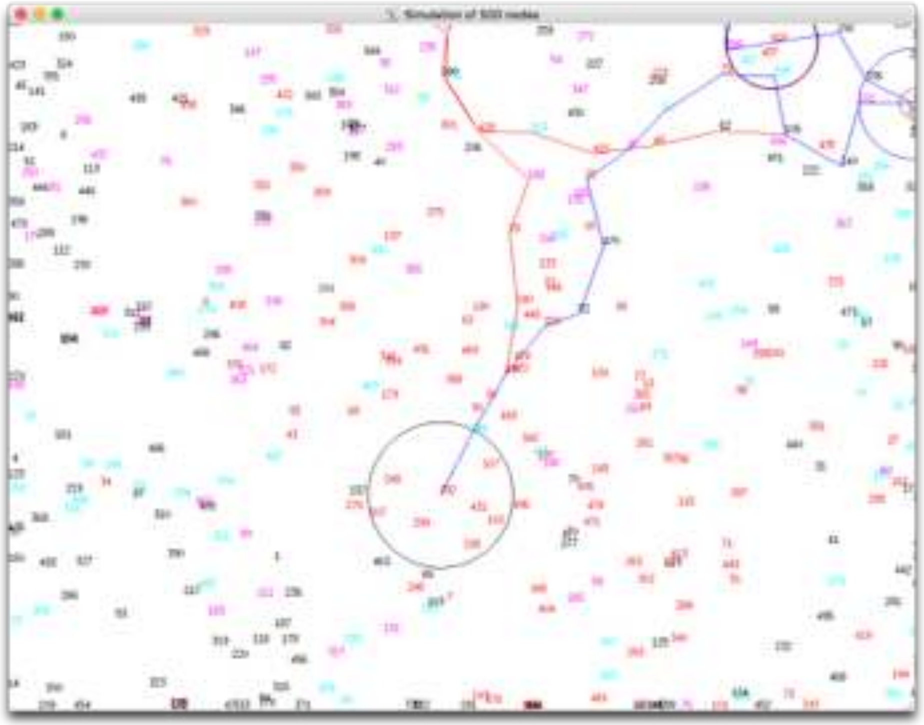
- complex control and data structures
- lots of
  - communication
  - synchronization
  - concurrency

What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of | communication
  synchronization
  concurrency

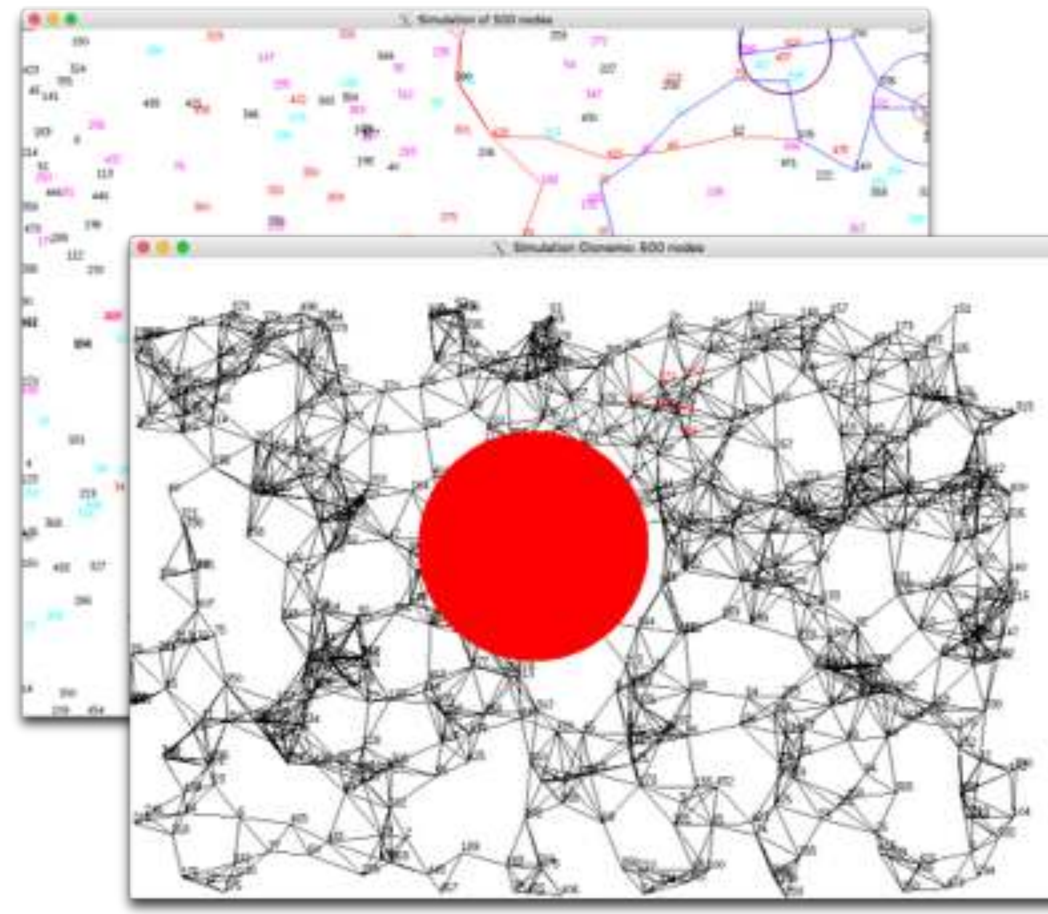


What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of
  - communication
  - synchronization
  - concurrency

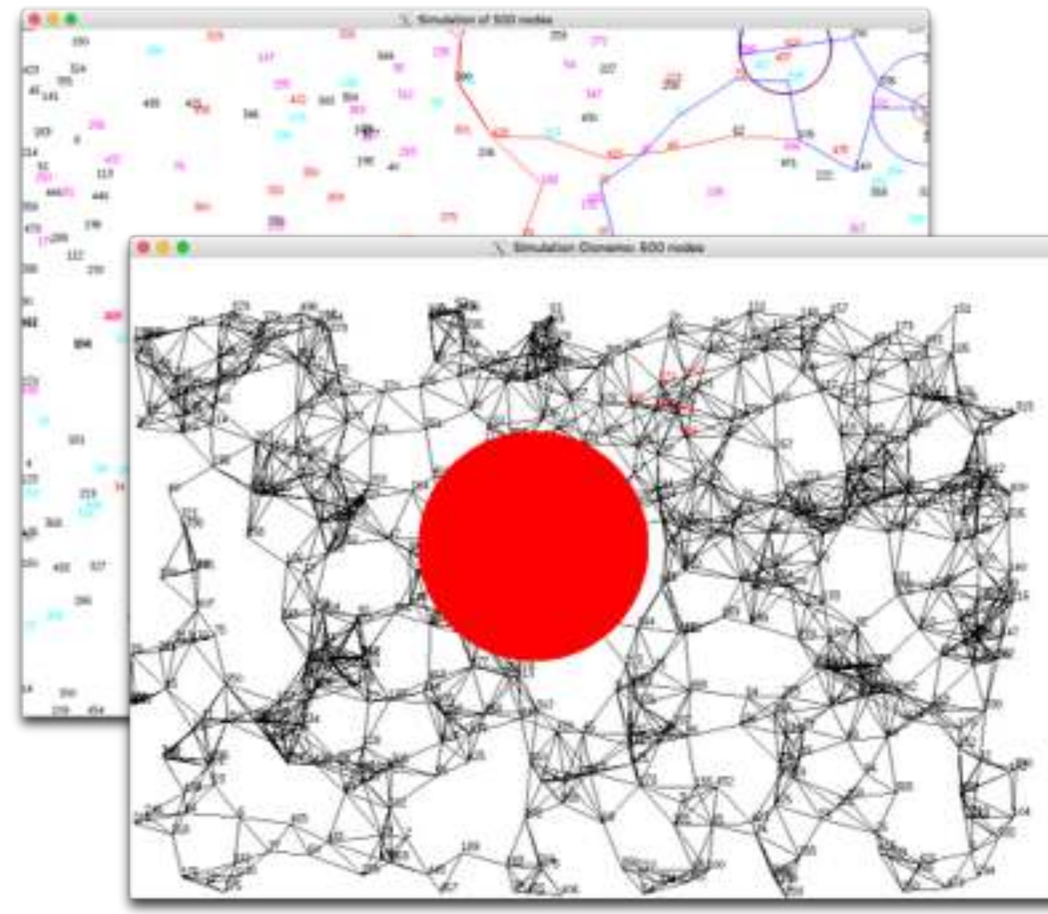


What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of | communication
  synchronization
  concurrency



What?

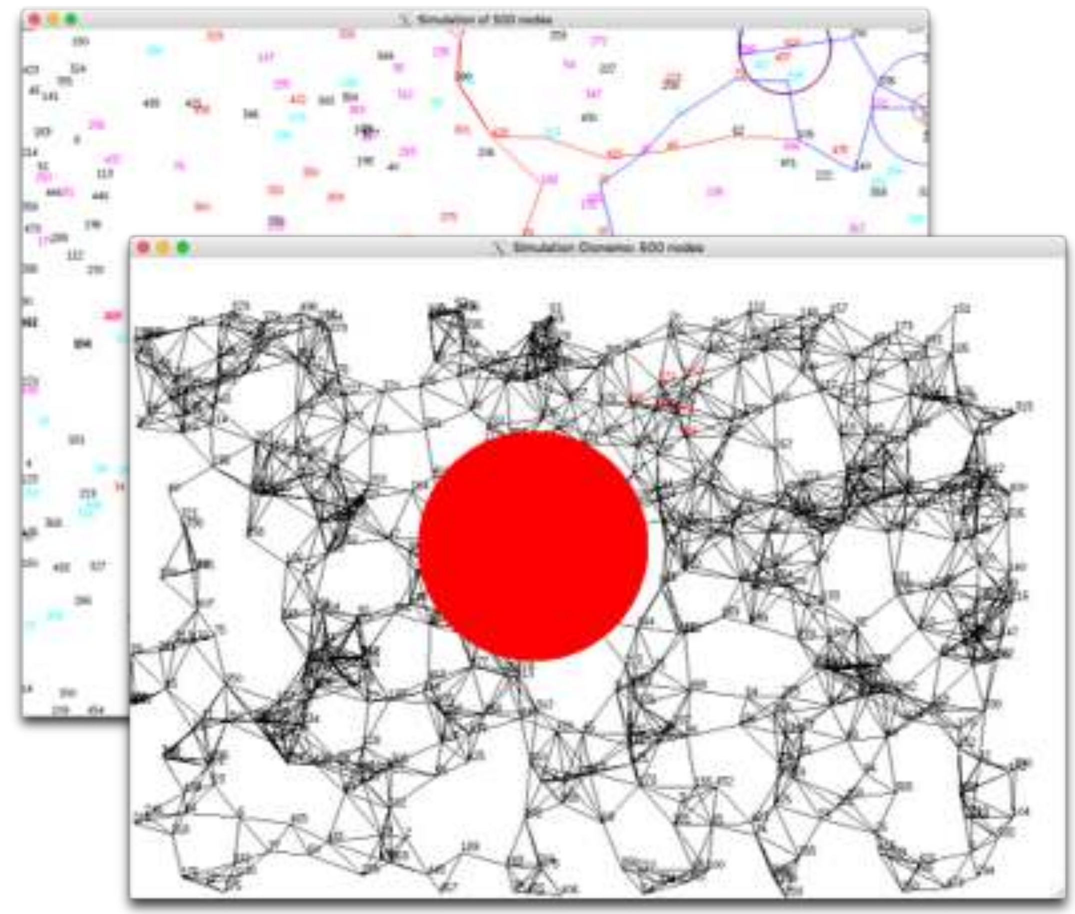
General purpose programming language

What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of | communication
  synchronization
  concurrency



What?

General purpose programming language

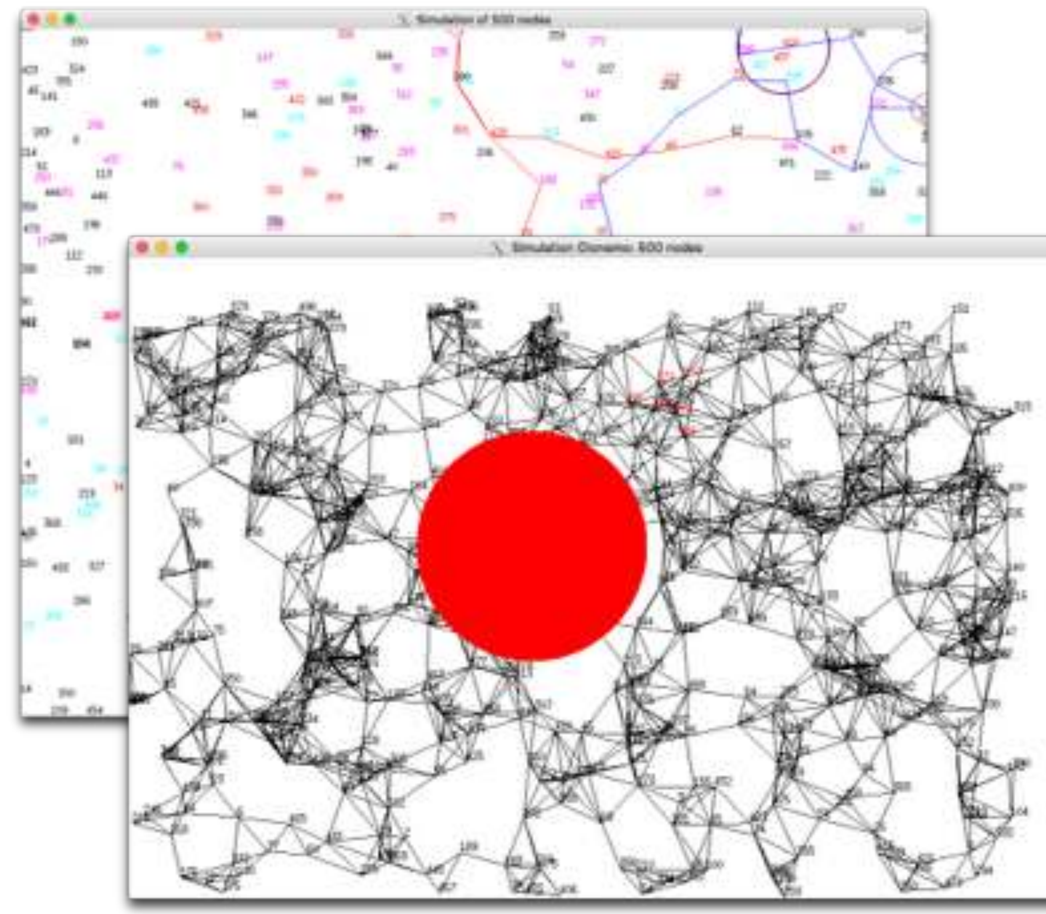
- Functional à la ML (OCaml)
- Synchronous lock-step concurrency (Esterel)
- Dynamic creation of processes

What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of | communication
  synchronization
  concurrency

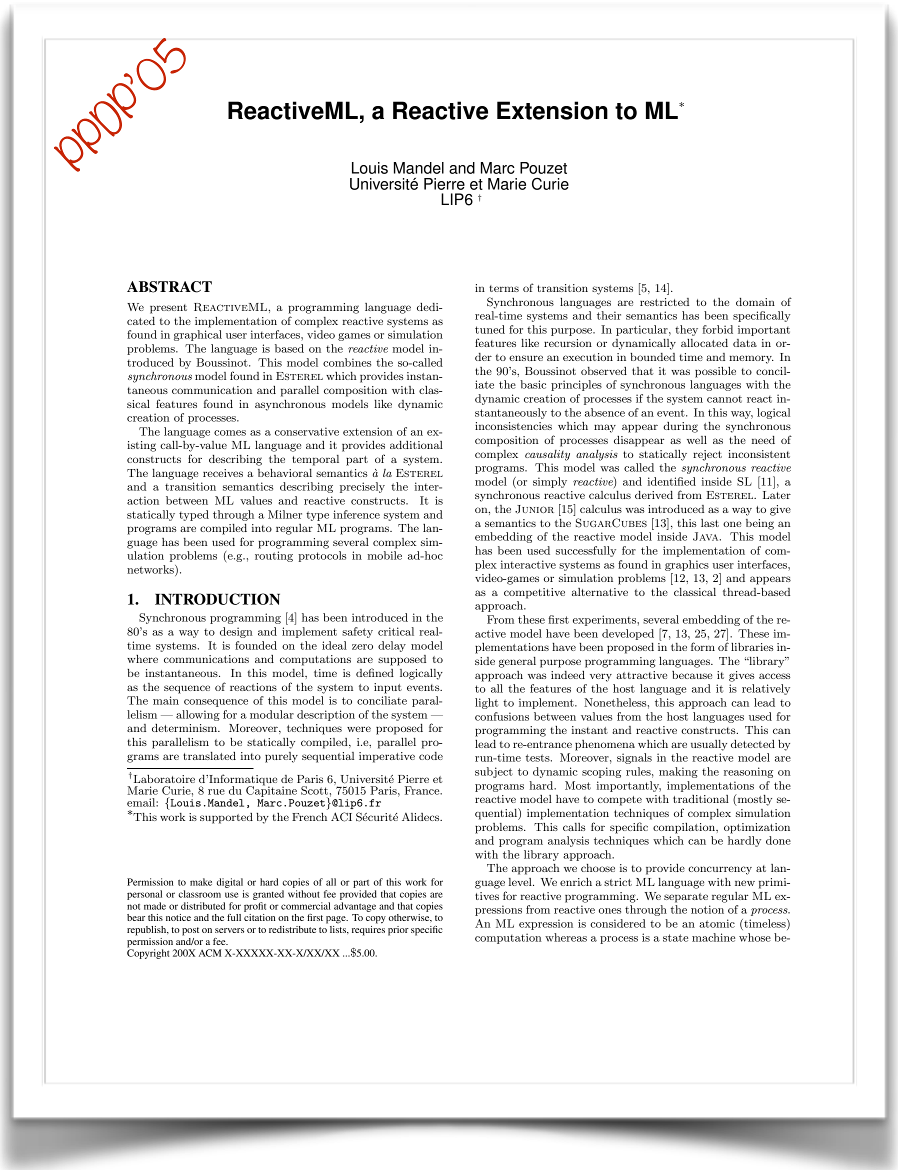


What?

General purpose programming language

- Functional à la ML (OCaml)
- Synchronous lock-step concurrency (Esterel)
- Dynamic creation of processes

No real-time nor bounded
memory constraints



pppp'05

ReactiveML, a Reactive Extension to ML*

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6 †

ABSTRACT

We present ReactiveML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Boussinot. This model combines the so-called *synchronous* model found in Esterel which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics *à la* Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type inference system and programs are compiled into regular ML programs. The language has been used for programming several complex simulation problems (e.g., routing protocols in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement safety critical real-time systems. It is founded on the ideal zero delay model where communications and computations are supposed to be instantaneous. In this model, time is defined logically as the sequence of reactions of the system to input events. The main consequence of this model is to conciliate parallelism — allowing for a modular description of the system — and determinism. Moreover, techniques were proposed for this parallelism to be statically compiled, i.e, parallel programs are translated into purely sequential imperative code in terms of transition systems [5, 14].

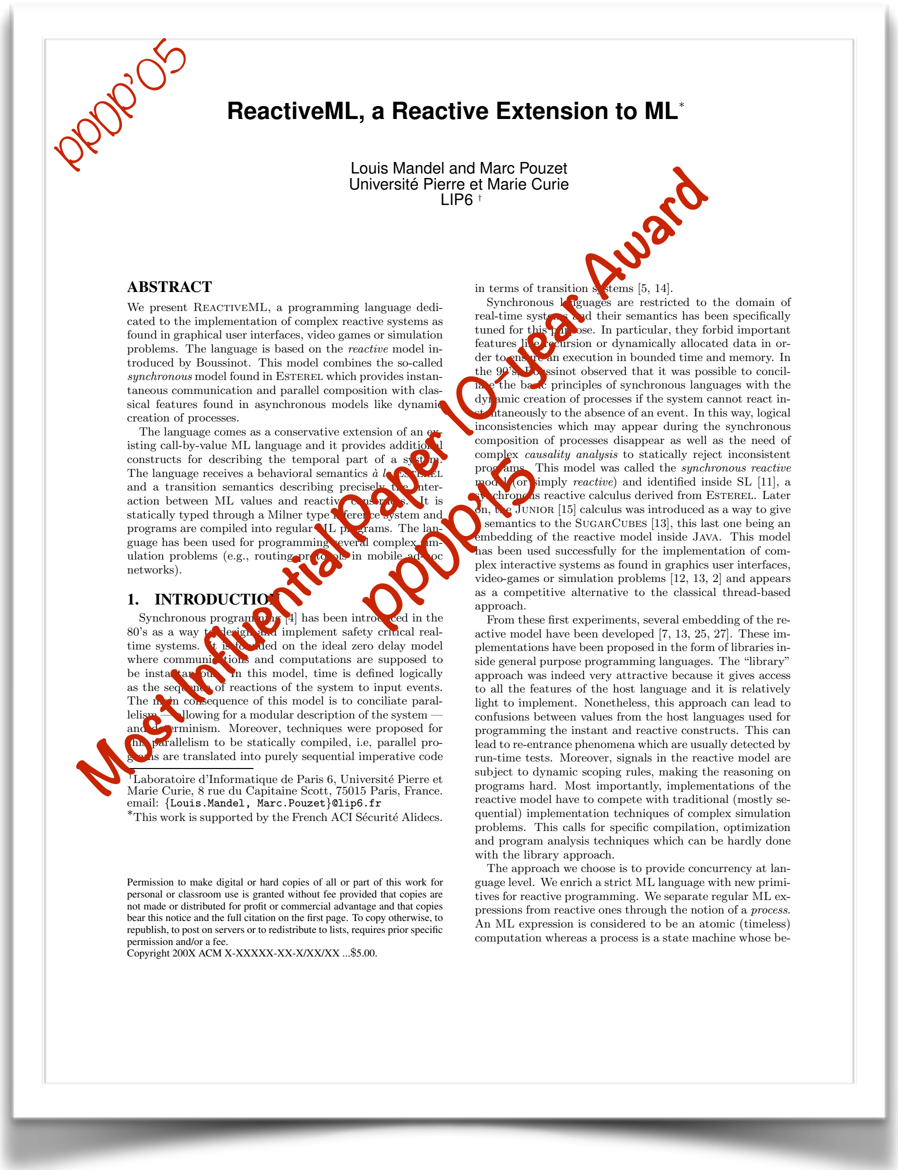
Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's, Boussinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a synchronous reactive calculus derived from Esterel. Later on, the Junior [15] calculus was introduced as a way to give a semantics to the SugarCubes [13], this last one being an embedding of the reactive model inside Java. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the reactive model have been developed [7, 13, 25, 27]. These implementations have been proposed in the form of libraries inside general purpose programming languages. The "library" approach was indeed very attractive because it gives access to all the features of the host language and it is relatively light to implement. Nonetheless, this approach can lead to confusions between values from the host languages used for programming the instant and reactive constructs. This can lead to re-entrance phenomena which are usually detected by run-time tests. Moreover, signals in the reactive model are subject to dynamic scoping rules, making the reasoning on programs hard. Most importantly, implementations of the reactive model have to compete with traditional (mostly sequential) implementation techniques of complex simulation problems. This calls for specific compilation, optimization and program analysis techniques which can be hardly done with the library approach.

The approach we choose is to provide concurrency at language level. We enrich a strict ML language with new primitives for reactive programming. We separate regular ML expressions from reactive ones through the notion of a *process*. An ML expression is considered to be an atomic (timeless) computation whereas a process is a state machine whose be-

†Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, 8 rue du Capitaine Scott, 75015 Paris, France. email: {Louis.Mandel, Marc.Pouzet}@lip6.fr

*This work is supported by the French ACI Sécurité Alidecs.



ReactiveML, a Reactive Extension to ML*

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6 †

ABSTRACT

We present ReactiveML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Boussinot. This model combines the so-called *synchronous* model found in Esterel which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics *à la* Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type inference system and programs are compiled into regular ML programs. The language has been used for programming several complex simulation problems (e.g., routing protocols in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement safety critical real-time systems. It is founded on the ideal zero delay model where communications and computations are supposed to be instantaneous. In this model, time is defined logically as the sequence of reactions of the system to input events. The main consequence of this model is to conciliate parallelism — allowing for a modular description of the system — and determinism. Moreover, techniques were proposed for this parallelism to be statically compiled, i.e, parallel programs are translated into purely sequential imperative code in terms of transition systems [5, 14].

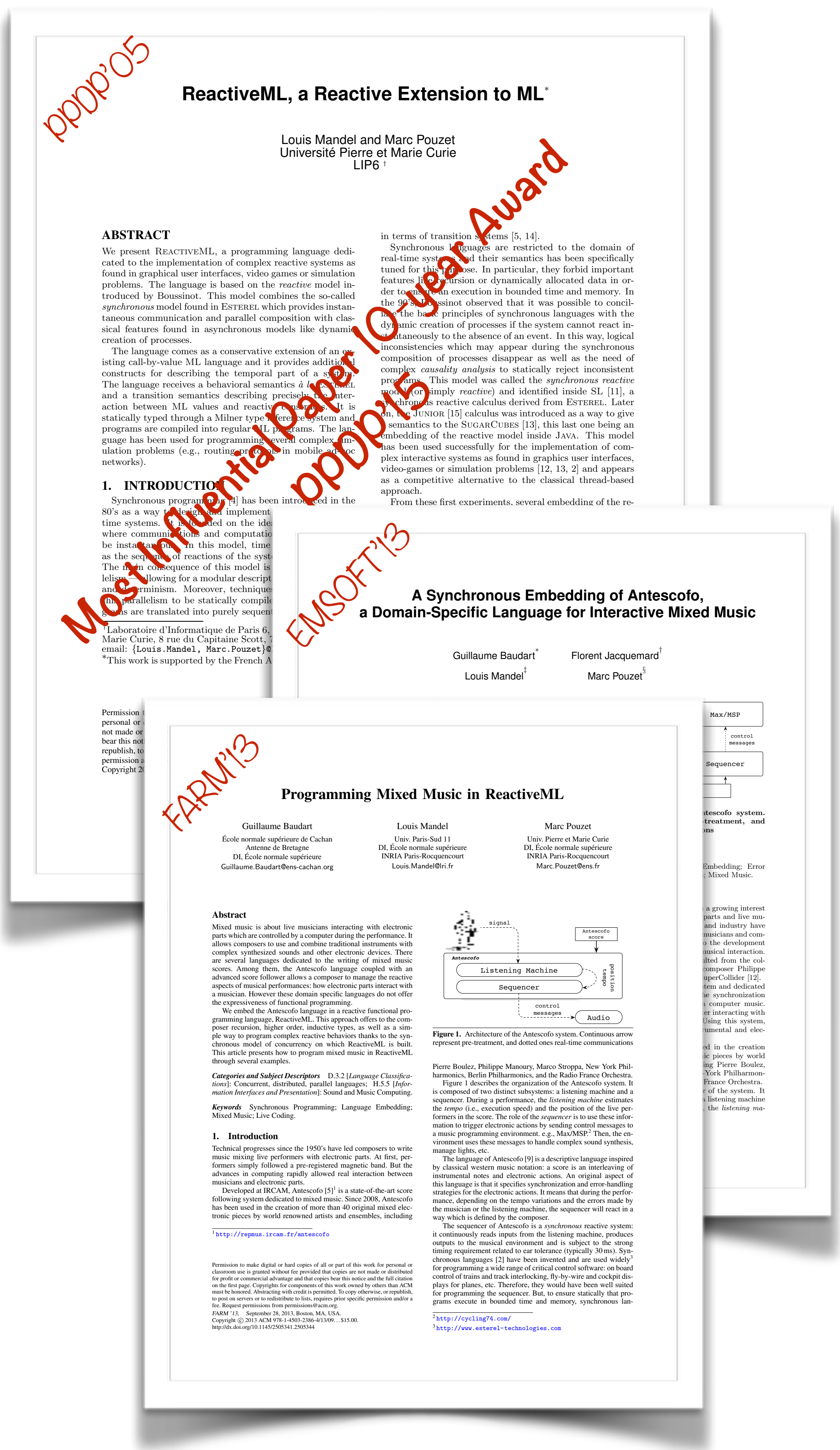
Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's, Boussinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a synchronous reactive calculus derived from Esterel. Later on, the Junior [15] calculus was introduced as a way to give a semantics to the SugarCubes [13], this last one being an embedding of the reactive model inside Java. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the reactive model have been developed [7, 13, 25, 27]. These implementations have been proposed in the form of libraries inside general purpose programming languages. The "library" approach was indeed very attractive because it gives access to all the features of the host language and it is relatively light to implement. Nonetheless, this approach can lead to confusions between values from the host languages used for programming the instant and reactive constructs. This can lead to re-entrance phenomena which are usually detected by run-time tests. Moreover, signals in the reactive model are subject to dynamic scoping rules, making the reasoning on programs hard. Most importantly, implementations of the reactive model have to compete with traditional (mostly sequential) implementation techniques of complex simulation problems. This calls for specific compilation, optimization and program analysis techniques which can be hardly done with the library approach.

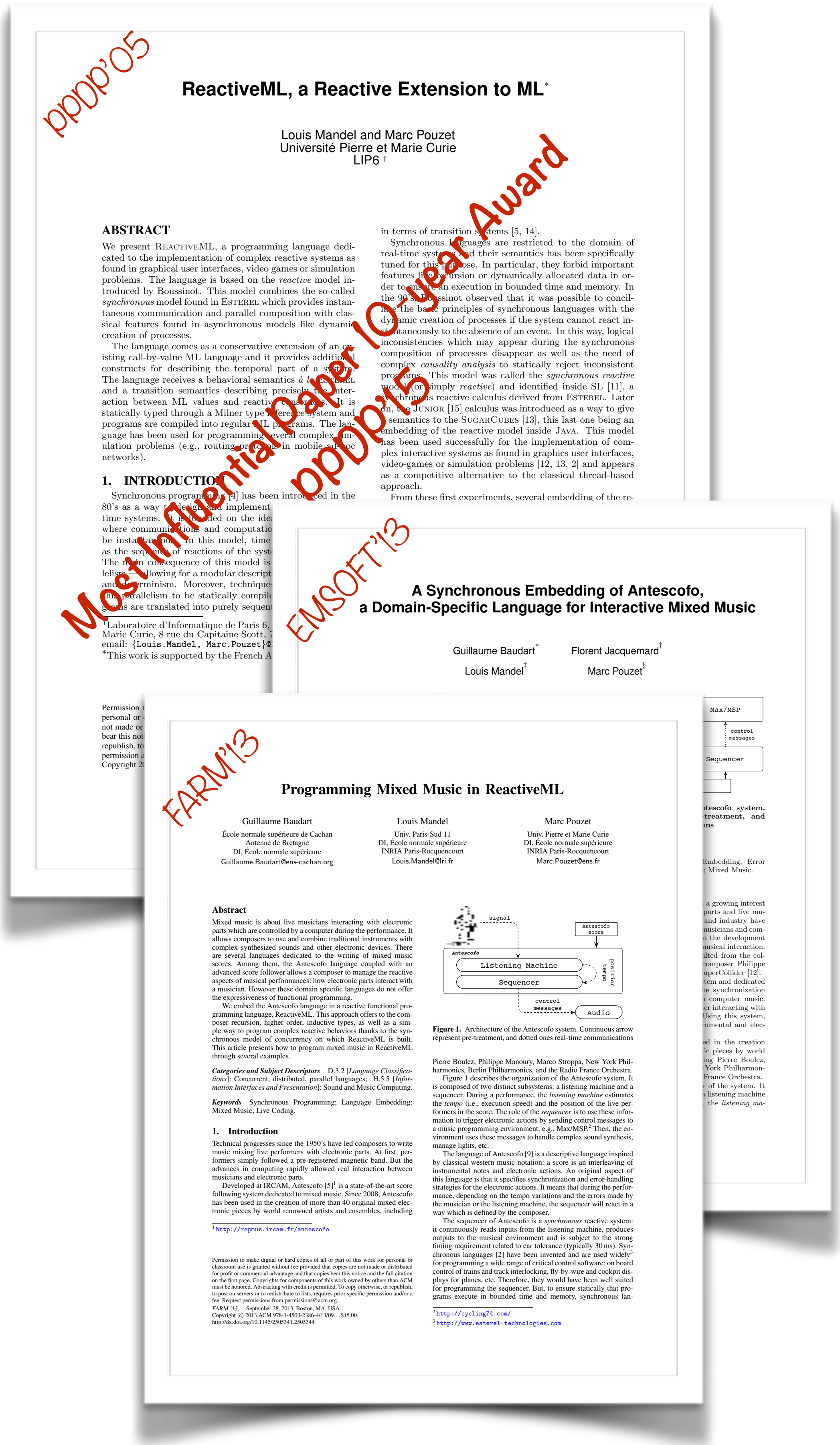
The approach we choose is to provide concurrency at language level. We enrich a strict ML language with new primitives for reactive programming. We separate regular ML expressions from reactive ones through the notion of a *process*. An ML expression is considered to be an atomic (timeless) computation whereas a process is a state machine whose be-

†Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, 8 rue du Capitaine Scott, 75015 Paris, France. email: {Louis.Mandel, Marc.Pouzet}@lip6.fr

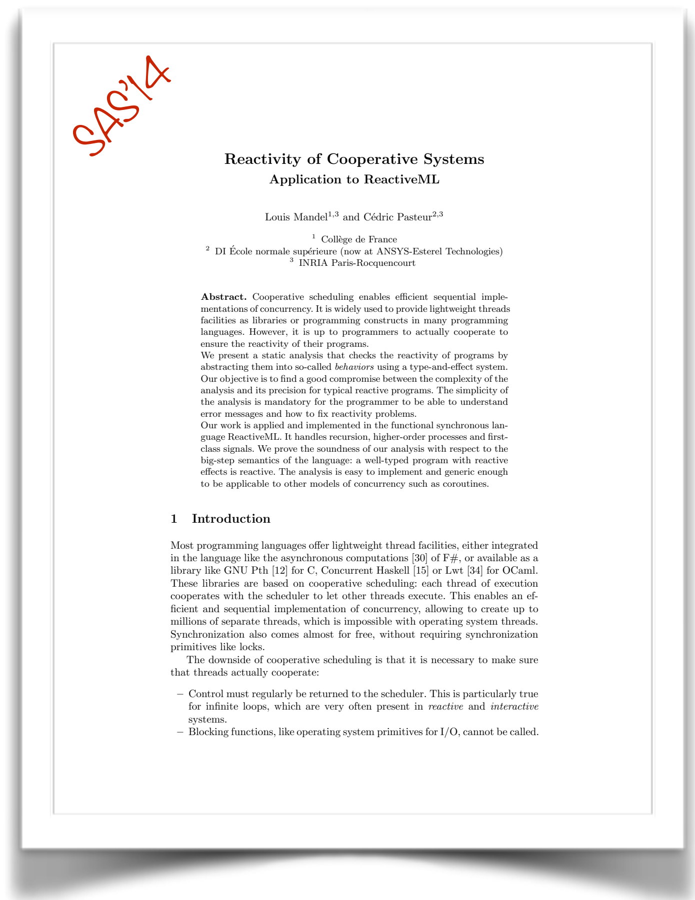
*This work is supported by the French ACI Sécurité Alidecs.



Application: Mixed Music

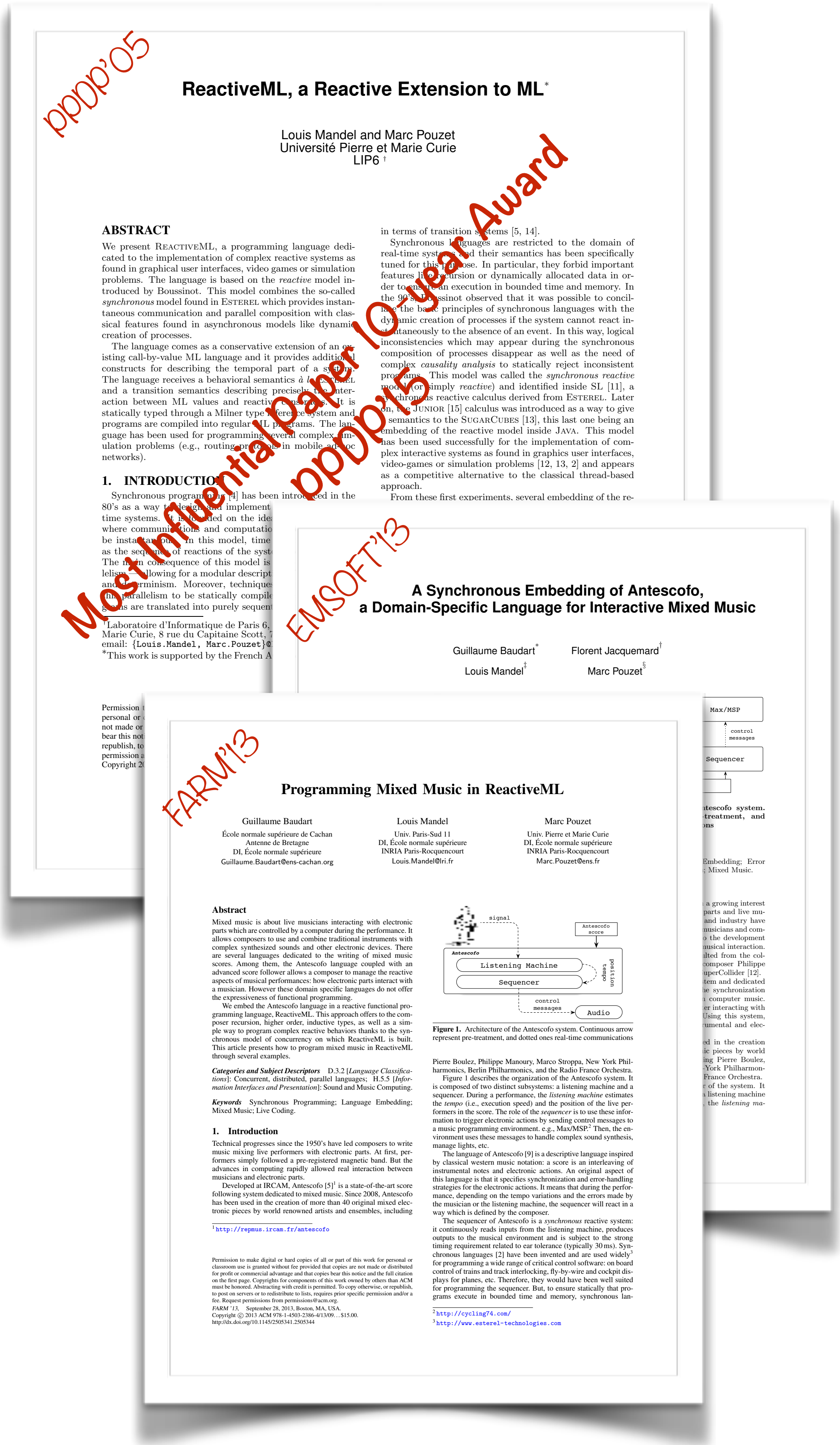


Application: Mixed Music



Extensions

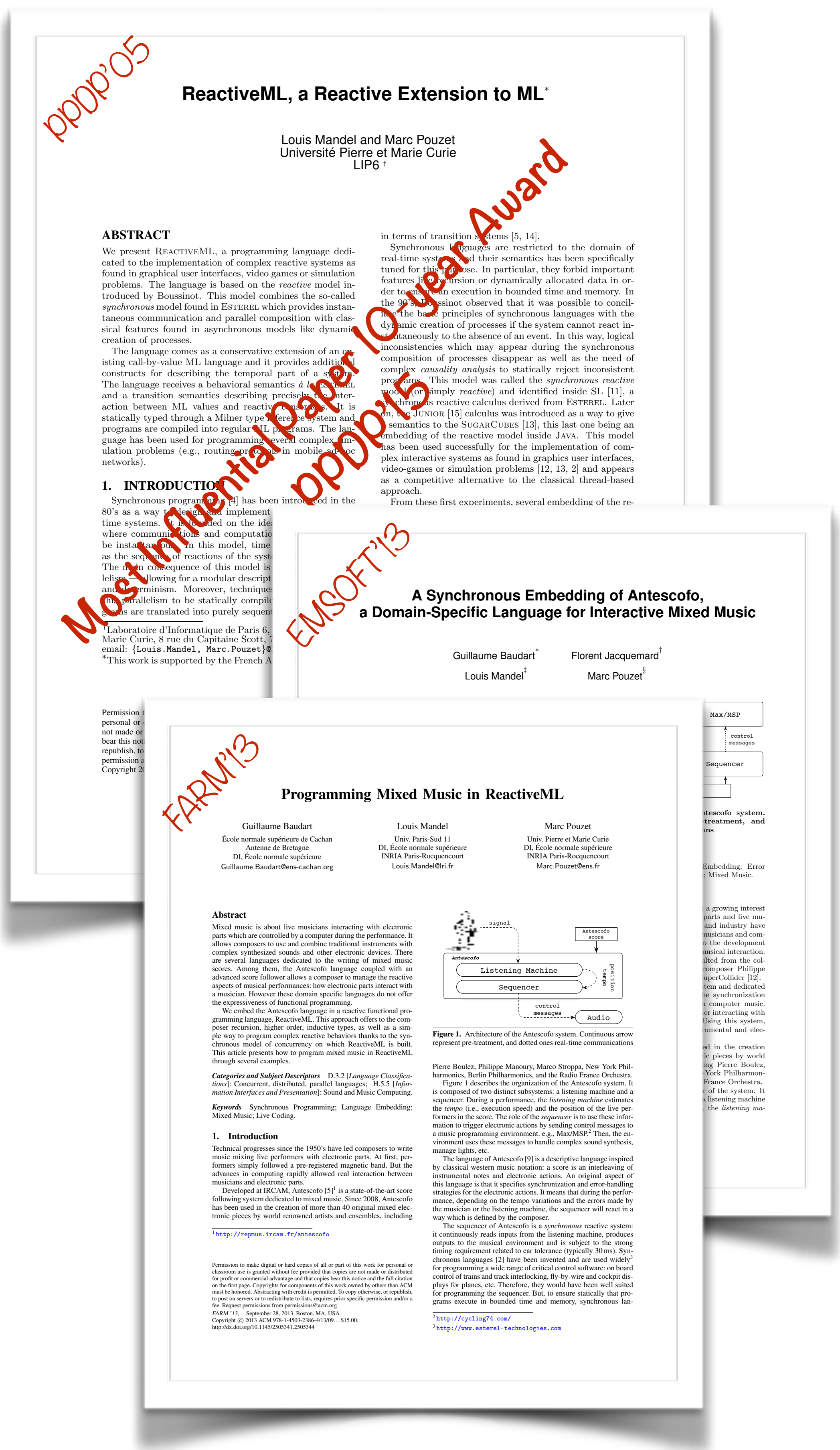
- Reactivity Analysis

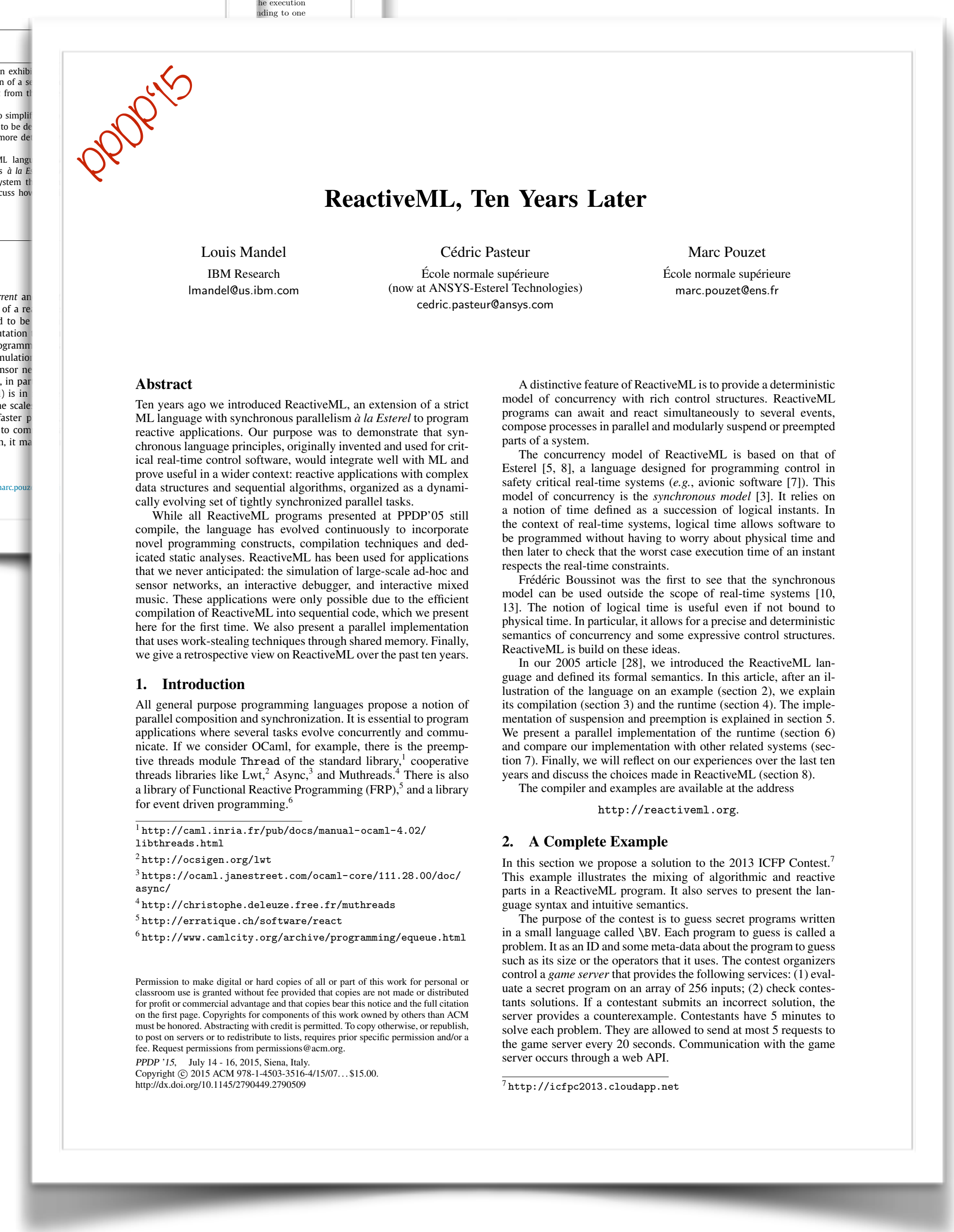


Extensions

- Reactivity Analysis
- Time Refinement

Application: Mixed Music

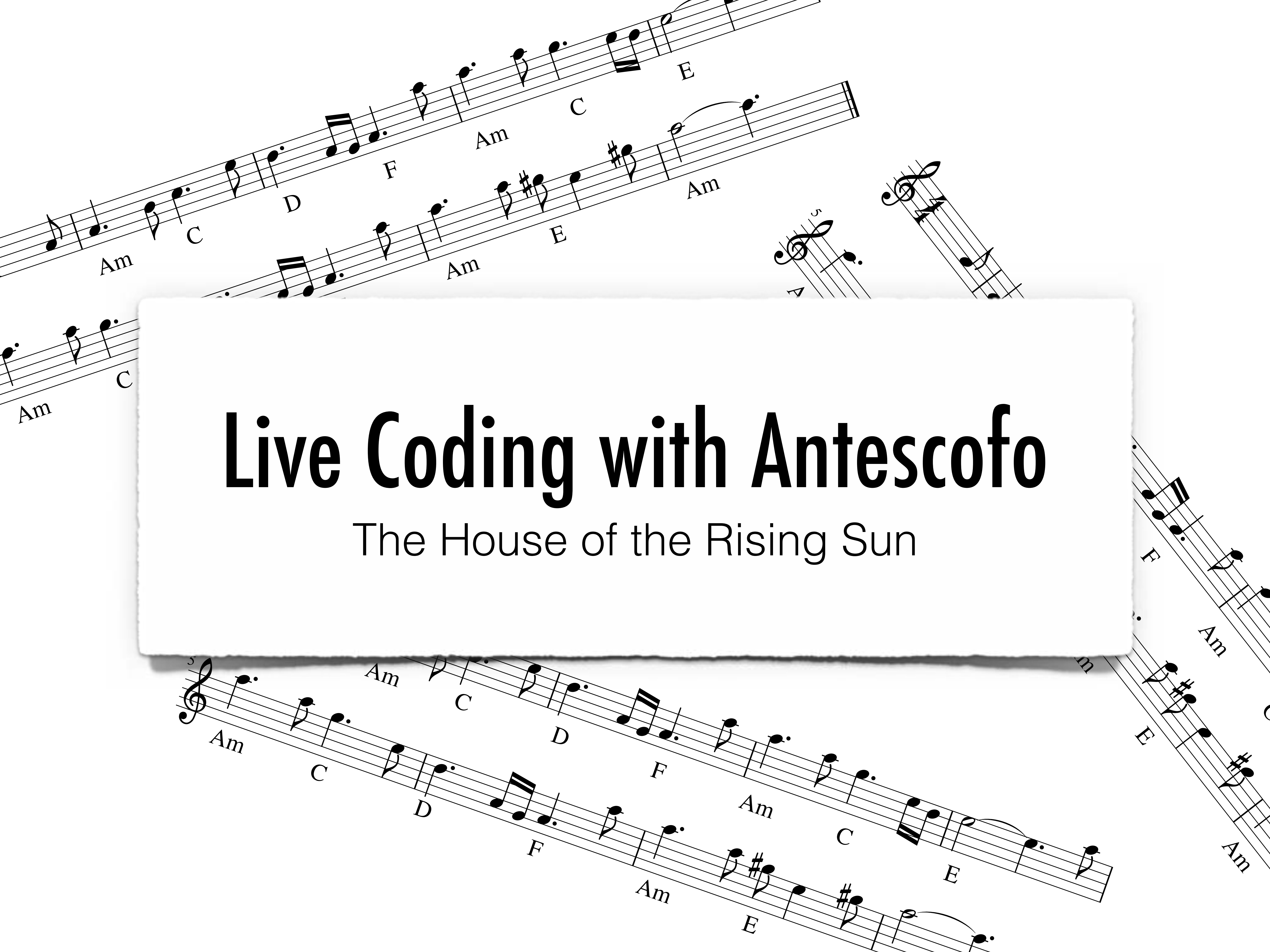




Extensions

- Reactivity Analysis
- Time Refinement

Application: Mixed Music



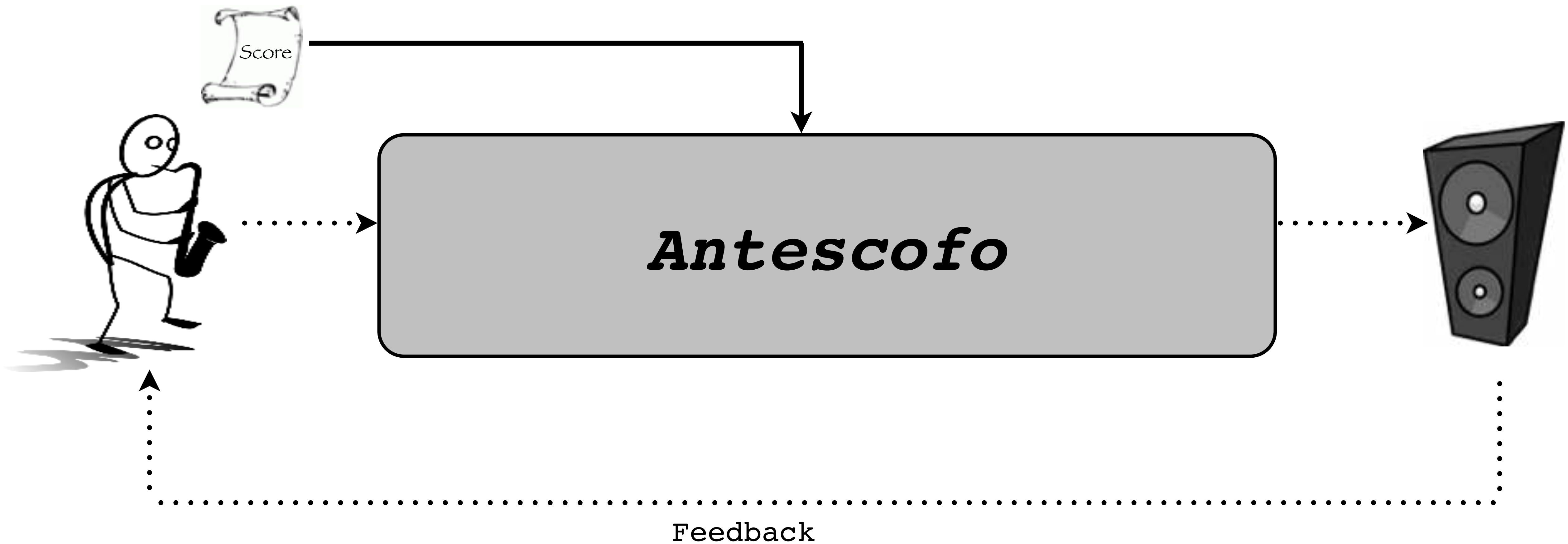
Live Coding with Antescofo

The House of the Rising Sun

Mixed Music with Antescofo

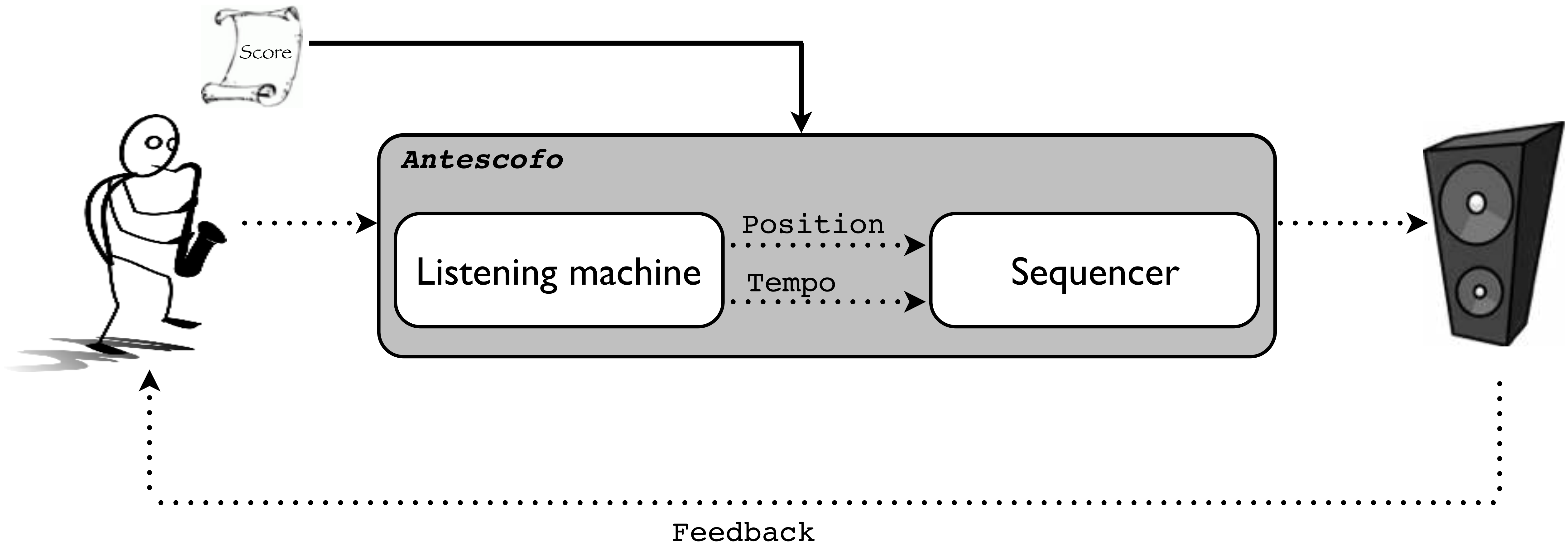
[Cont 2008]

Mixed Music with Antescofo



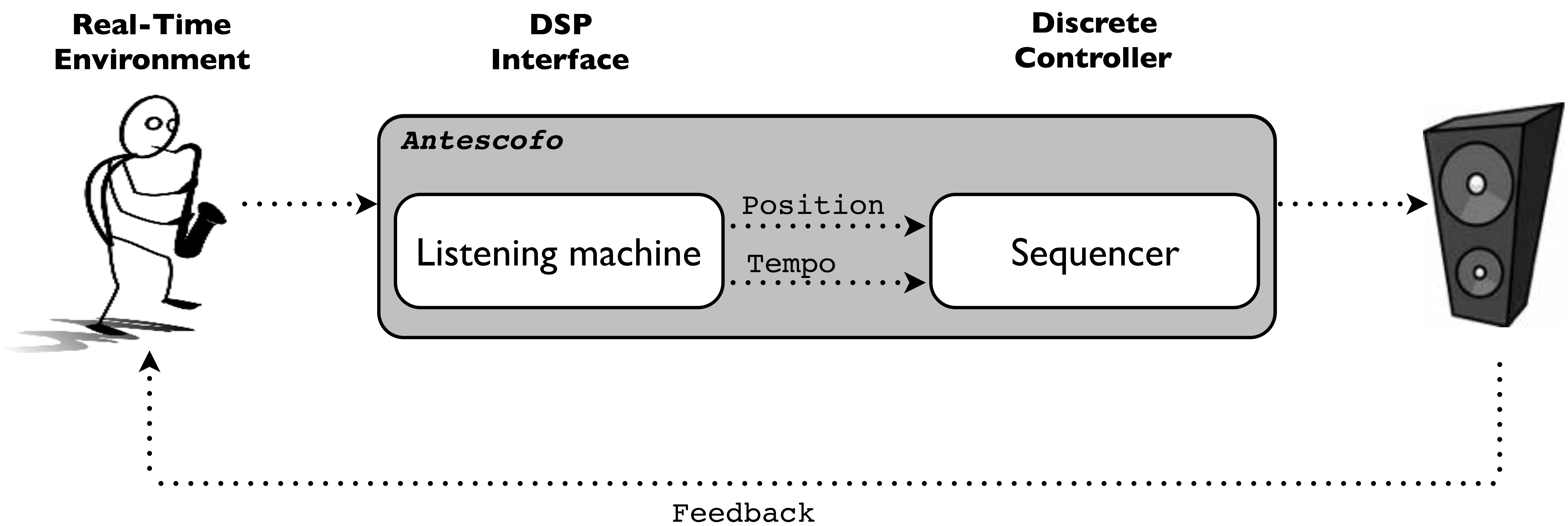
[Cont 2008]

Mixed Music with Antescofo



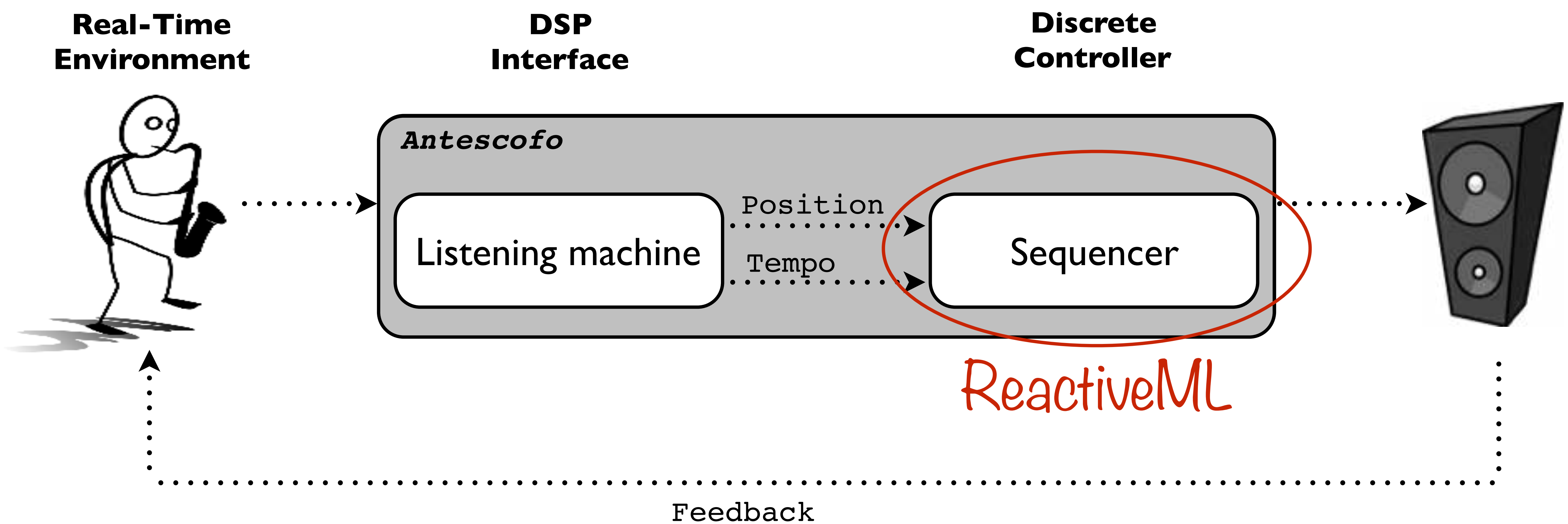
[Cont 2008]

Mixed Music with Antescofo



[Cont 2008]

Mixed Music with Antescofo



[Cont 2008]

```
let process killable kill p =
   do
     run p
   until kill done
```

val killable: (unit, unit) event -> unit process -> unit process

```
let process killable kill p =
   do
     run p
   until kill done
```

control signal

val killable: (unit, unit) event -> unit process -> unit process

```
let process killable kill p =
   do
     run p
   until kill done
```

control signal

val killable: (unit, unit) event -> unit process -> unit process

```
let rec process replaceable replace p =
  do
    run p
  until replace(q) ->
    run (replaceable replace q)
  done
```

val replaceable: (unit process, unit process) event -> unit process -> unit process

```
let process killable kill p =
    do
      run p
    until kill done
```

control signal

val killable: (unit, unit) event -> unit process -> unit process

```
let rec process replaceable replace p =
  do
    run p
  until replace(q) ->
    run (replaceable replace q)
  done
```

control signal

carries a process

val replaceable: (unit process, unit process) event -> unit process -> unit process

Reactive Domains

n-body Simulation

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

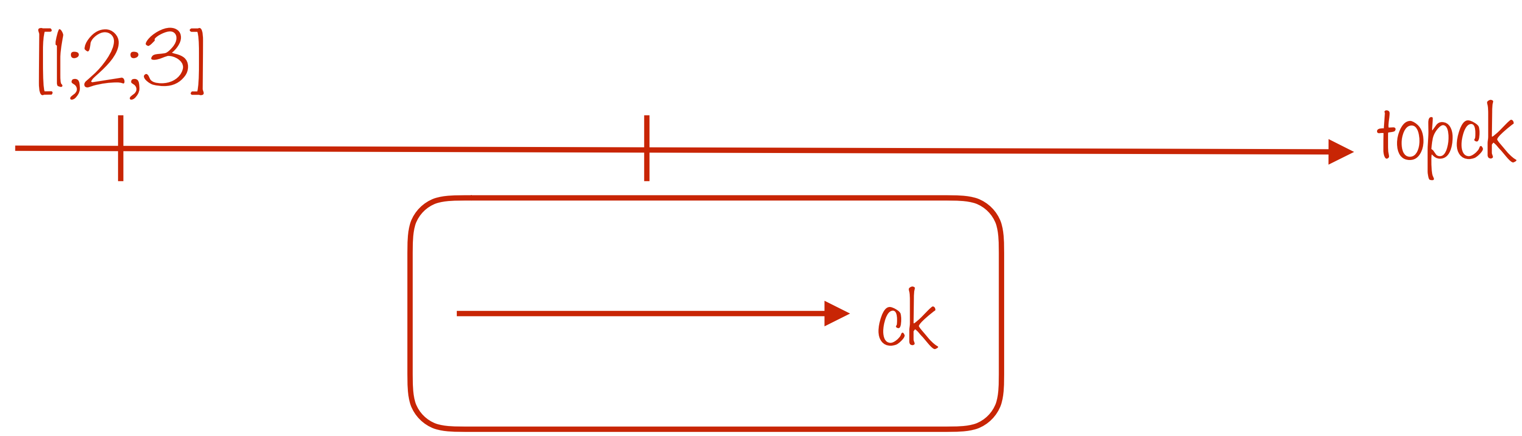
Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

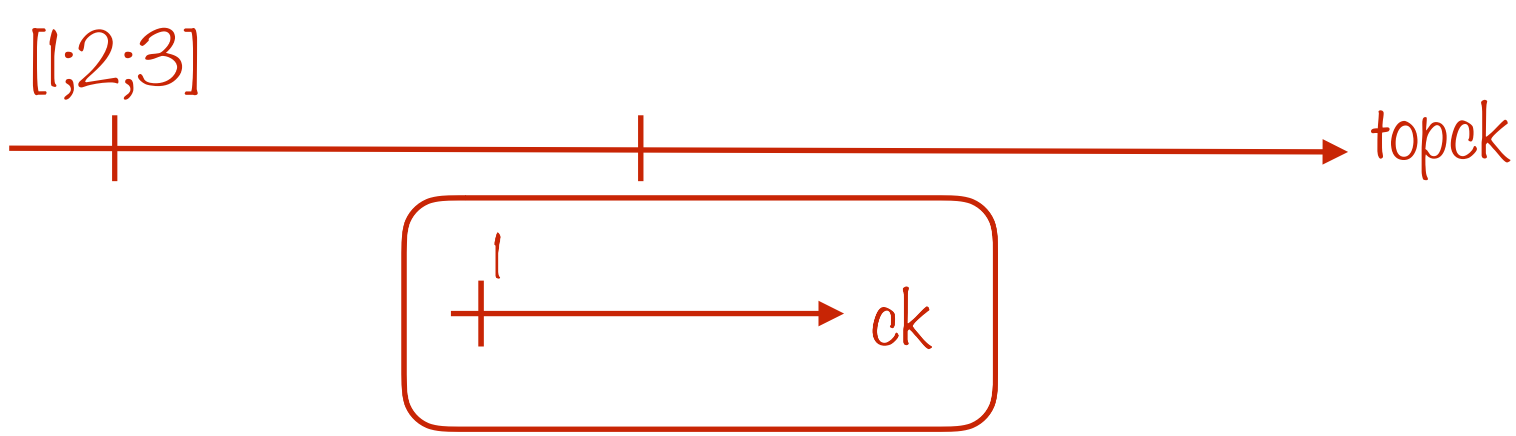


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

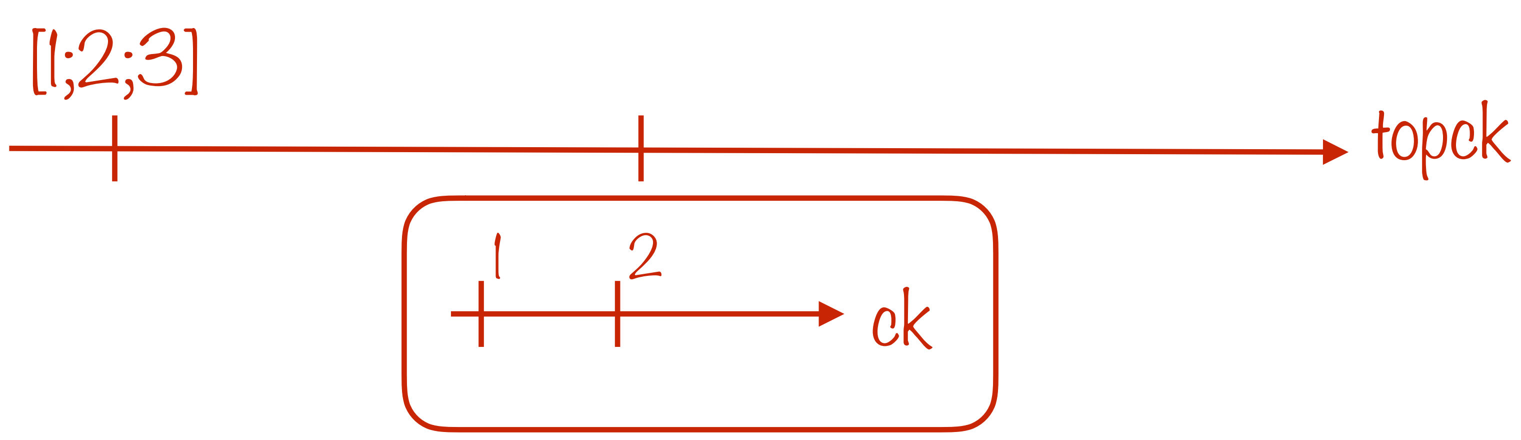


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

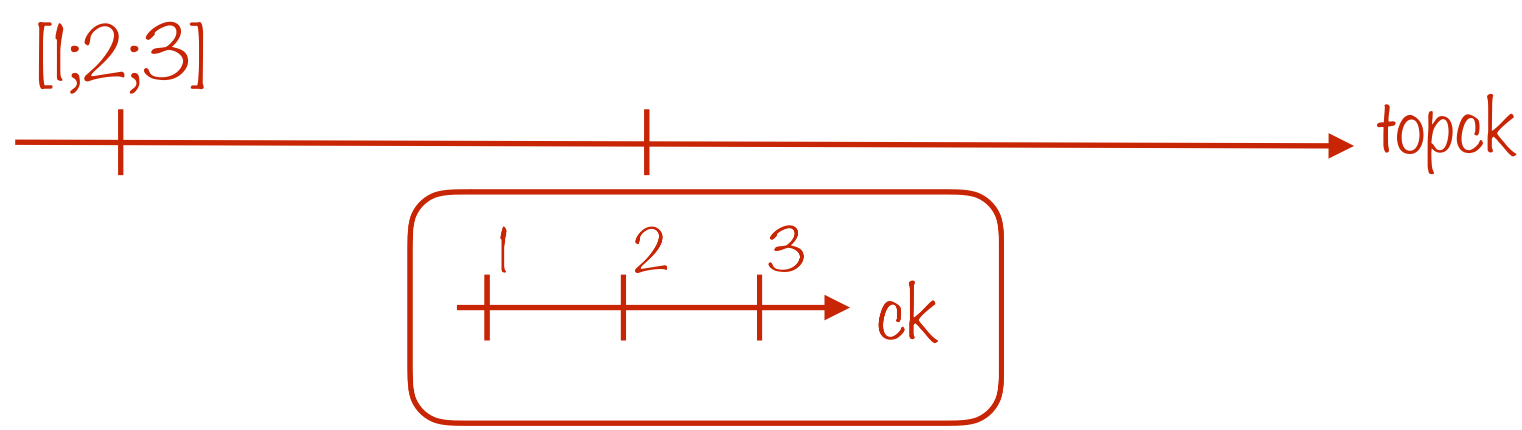


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck



Reactive Domains

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

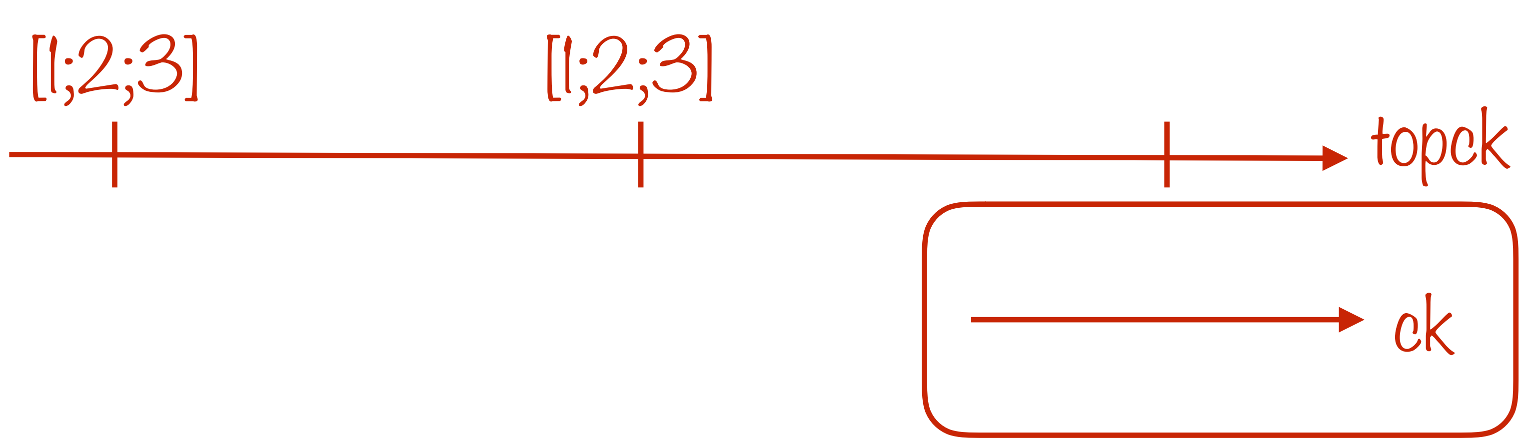
Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck



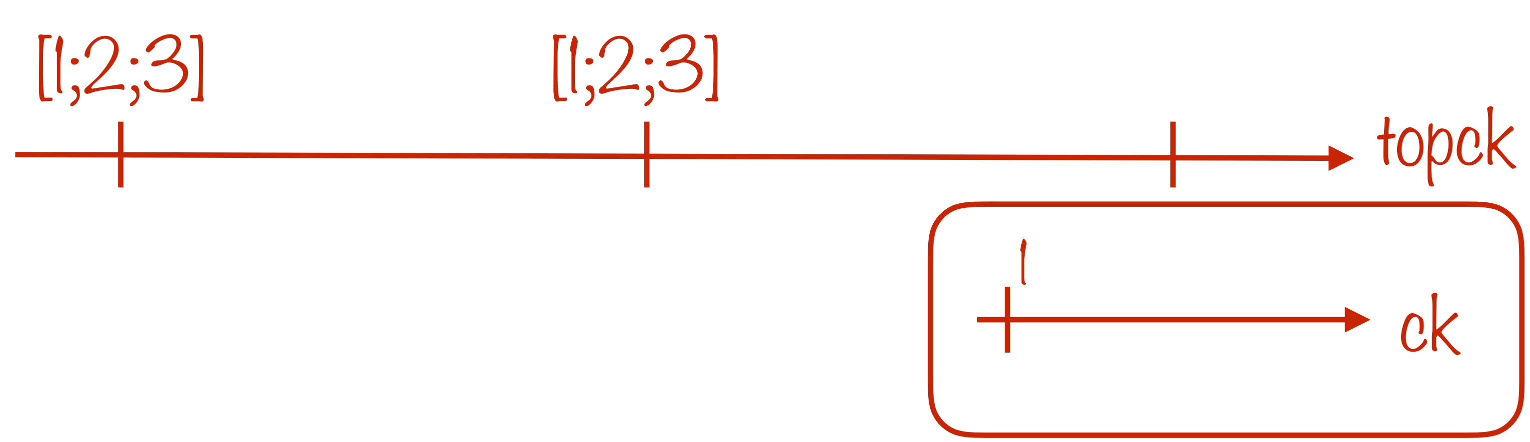
Reactive Domains

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck

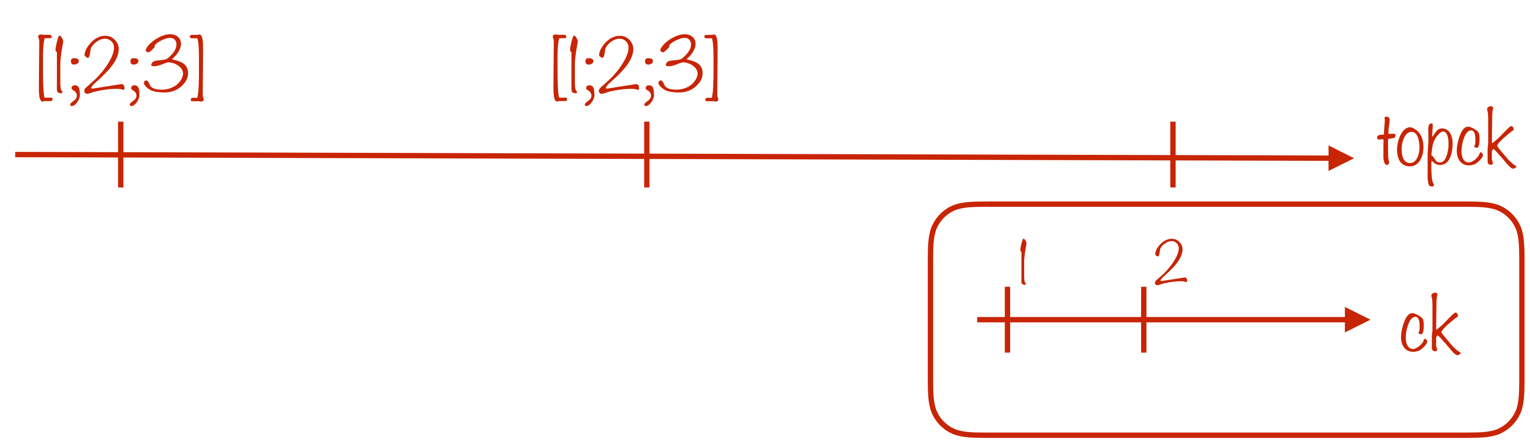


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck



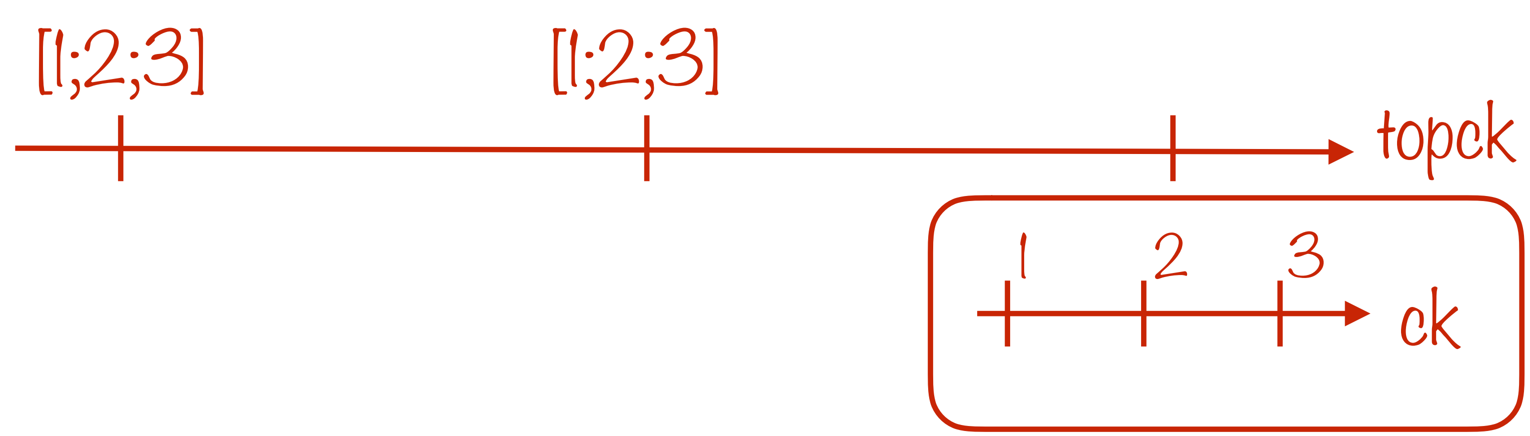
Reactive Domains

Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised by a domain

Here 3 steps of ck for one of topck



```
let process main =
  ...
  domain computation_ck do
    signal env default zero_force gather add_force in
    for i = 1 to 10 dopar
      run body env (planet_sprite vp) (random_planet ())
    done
     ||
    run body env (sun_sprite vp) (sun ())
  done
  ...
```

n-body simulation

Hide internal step for integration

```
let process main =
  ...
  domain computation_ck do
    signal env default zero_force gather add_force in
    for i = 1 to 10 dopar
      run body env (planet_sprite vp) (random_planet ())
    done
     ||
    run body env (sun_sprite vp) (sun ())
  done
  ...
```

n-body simulation

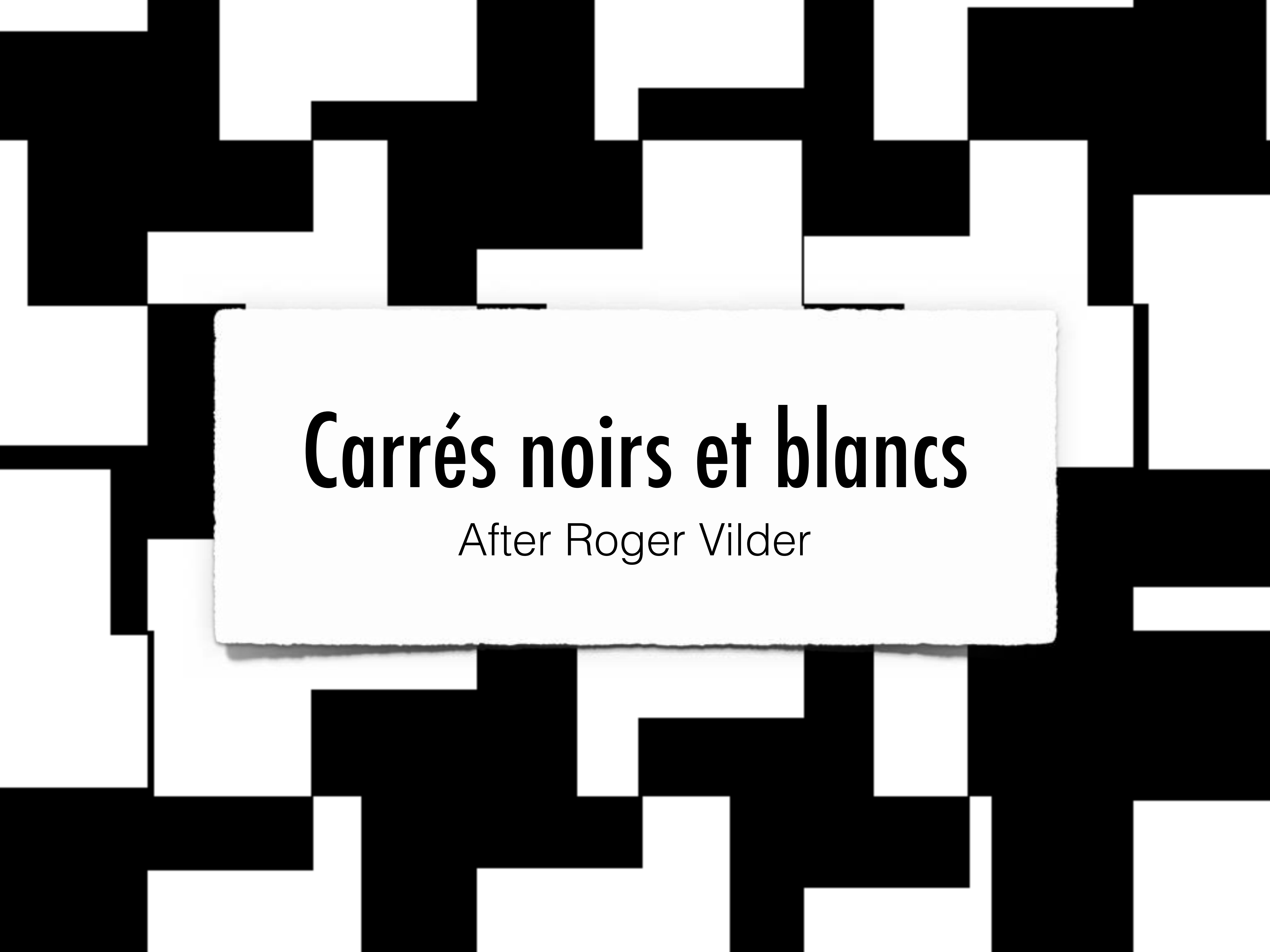
Hide internal step for integration

```
let process main =
  ...
  domain computation_ck do
    signal env default zero_force gather add_force in
    for i = 1 to 10 dopar
      run body env (planet_sprite vp) (random_planet ())
    done
     ||
    run body env (sun_sprite vp) (sun ())
  done
  ...
```

Example RK4: 4 steps

```
let process compute_k4 env st =
  (* step 1 *)
  let k1_v = run (compute_a env st.b_pos st.b_weight) in
  let k1_p = st.b_vel in
  (* step 2 *)
  let k2_p = add_v st.b_vel (sc_mult (dt /. 2.0) k1_v) in
  let x_2 = add_v st.b_pos (sc_mult (dt /. 2.0) k1_p) in
  let k2_v = run (compute_a env x_2 st.b_weight) in
  ...
```

http://reactiveml.org



Carrés noirs et blancs

After Roger Vilder

A simple integrator

x(n+1) = x(n) + delta

```
let process sum state delta =
  loop
    emit state (last ?state +. delta);
    pause
  end
```

val sum: (float, float) event -> float -> unit process

A simple integrator

x(n+1) = x(n) + delta

```
let process sum state delta =
  loop
    emit state (last ?state +. delta);
    pause
  end
```

Discrete-Time step

val sum: (float, float) event -> float -> unit process

A simple integrator

x(n+1) = x(n) + delta

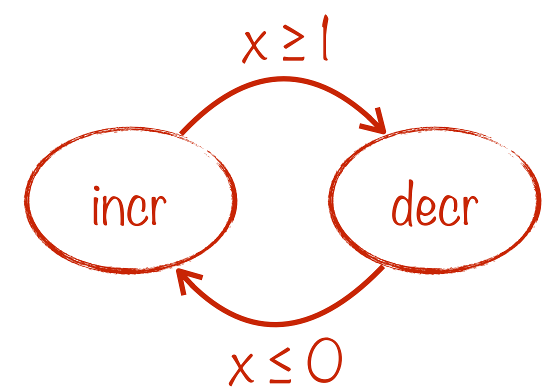
Communication via valued signals

```
let process sum state delta =
  loop
    emit state (last ?state +. delta);
    pause
  end
```

Discrete-Time step

val sum: (float, float) event -> float -> unit process

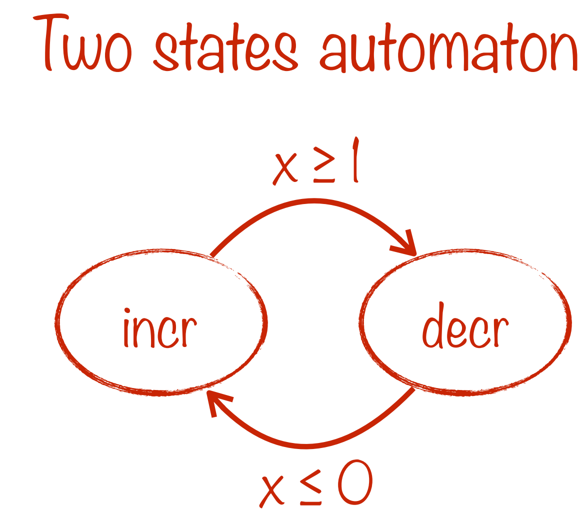
Two states automaton




```
let process incr_decr state delta =

  let rec process incr =
    do run sum state delta
    until state(x) when x >= 1. -> run decr done

  and process decr =
    do run sum state (-. delta)
    until state(x) when x <= 0. -> run incr done

  in run incr
```

val incr_ decr: (float, float) event -> float -> unit process

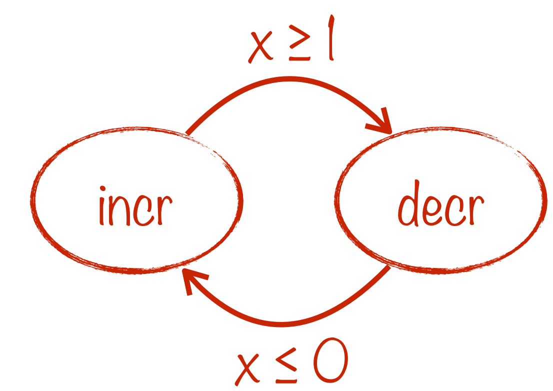
Two mutually recursive processes




```
let process incr_decr state delta =

  let rec process incr =
    do run sum state delta
    until state(x) when x >= 1. -> run decr done

  and process decr =
    do run sum state (-. delta)
    until state(x) when x <= 0. -> run incr done

  in run incr
```

val incr_ decr: (float, float) event -> float -> unit process

Two states automaton



Two mutually recursive processes

```
let process incr_decr state delta =

  let rec process incr =
    do run sum state delta
    until state(x) when x >= 1. -> run decr done

  and process decr =
    do run sum state (-. delta)
    until state(x) when x <= 0. -> run incr done

  in run incr
```

Preemption: do ... until

val incr_decr: (float, float) event -> float -> unit process

```
type dir = Up | Down | Left | Right

let process draw dir x y size state =
  loop
    await state(p) in
    begin match dir with
    | Up -> Graphics.fill_rect x y size (size *: p)
    | ...
  end
end
```

val draw: dir -> int -> int -> int -> (‘a, float) event -> unit process

Drawing process

Types definition and match similar to OCaml

```
type dir = Up | Down | Left | Right

let process draw dir x y size state =
  loop
    await state(p) in
    begin match dir with
    | Up -> Graphics.fill_rect x y size (size *: p)
    | ...
  end
end
```

val draw: dir -> int -> int -> int -> (‘a, float) event -> unit process

A simple Pump

```
let rec process pump dir x y size state delta =
  run incr_decr state delta ||
  run draw dir x y size state
```

val pump: dir -> int -> int -> int -> (float, float) event -> float -> unit process

A simple Pump

```
let rec process pump dir x y size state delta =
  run incr_decr state delta ||
  run draw dir x y size state
```

Parallel composition

val pump: dir -> int -> int -> int -> (float, float) event -> float -> unit process

```
let rec process splittable split dir x y size init =
  signal state default 0. gather (+.) in
  do
    emit state init;
    run pump dir x y size state (random_speed ())
  until split ->
    run cell split x y size (last ?state)
  done

and process cell split x y size init =
  let half = size/2 in
  run splittable split Left x y half init ||
  run splittable split Down (x+half) y half init ||
  run splittable split Up x (y+half) half init ||
  run splittable split Right (x+half) (y+half) half init
```

Run the process cell
when the signal split is emitted

```
let rec process splittable split dir x y size init =
  signal state default 0. gather (+.) in
  do
    emit state init;
    run pump dir x y size state (random_speed ())
  until split ->
    run cell split x y size (last ?state)
  done

and process cell split x y size init =
  let half = size/2 in
  run splittable split Left x y half init ||
  run splittable split Down (x+half) y half init ||
  run splittable split Up x (y+half) half init ||
  run splittable split Right (x+half) (y+half) half init
```

Dynamic creation of pumps

Run the process cell
when the signal split is emitted

```
let rec process splittable split dir x y size init =
  signal state default 0. gather (+.) in
  do
    emit state init;
    run pump dir x y size state (random_speed ())
  until split ->
    run cell split x y size (last ?state)
  done

and process cell split x y size init =
  let half = size/2 in
  run splittable split Left x y half init ||
  run splittable split Down (x+half) y half init ||
  run splittable split Up x (y+half) half init ||
  run splittable split Right (x+half) (y+half) half init
```

Split in 4 half-pumps
in the 4 directions