Clock domains in a reactive functional language

Cédric Pasteur

INRIA Team PARKAS
Laboratoire d’Informatique de l’École normale supérieure

28 nov. 2011
A reactive extension to ML

- ML: higher order functional language
- Synchronous primitives a la Esterel
  - Logical instants, processes, signals (first-class values)
  - Causality by construction
- Efficient sequential implementation
  - Dynamic scheduling (dynamic creation)
- Discrete simulation (sensor networks)
A simple example

- The factorial in n steps

```ml
let rec process pfact n =
    pause;
    if n <= 1 then 1
    else
        let v = run (pfact (n-1)) in
        print_int (n*v)
run (pfact 2) || run (pfact 3)
```
A simple example

- The factorial in n steps

```ml
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

pfact 3

pfact 2
A simple example

- The factorial in n steps

```ocaml
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

pfact 3

pfact 2
A simple example

- The factorial in n steps

```ocaml
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

pfact 3

pfact 2 '2'

3
A simple example

- The factorial in n steps

```ml
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)
run (pfact 2) || run (pfact 3)
```

pfact 2  `2`
pfact 3  `6`
Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

```ml
signal s default 0 gather (+) in
  emit s(1)
|| emit s(-1)
|| await s(v) in print_int v
```
Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

```ml
signal s default 0 gather (+) in
emit s(1)
|| emit s(-1)
|| await s(v) in print_int v
```
Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

```ml
signal s default 0 gather (+) in
  emit s(1)
|| emit s(-1)
|| await s(v) in print_int v
```

\{1, -1\}
Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

```ml
signal s default 0 gather (+) in
    emit s(1)
    || emit s(-1)
    || await s(v) in print_int v
```

{lundi 28 novembre 2011}
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
type state =
  { mutable b_pos : vector; mutable b_vel : vector;
    b_weight : float; }

let dt = 0.1
signal env default (fun _ -> zero_vector) gather add_force

let compute_euler st f =
  st.b_pos <- add_v st.b_pos (sc_mult dt st.b_vel);
  st.b_vel <- add_v st.b_vel (sc_mult dt f)

let process body =
  let st = new_state () in
  loop
    emit env (force st);
    await env f in
      compute_euler st (f st.b_pos)
  end
Modularity problem
Modularity problem

\[ \text{await } x(v) \text{ in } \]
\[ \text{emit } y(f(g \ v)) \]
Modularity problem

\[ f \circ g \]

```
await x(v) in
  emit y(f(g(v)))
```
Modularity problem

\[ f \circ g \]

\[
\text{await } x(v) \text{ in } \\
\text{emit } y(f(gv))
\]

\[ f(gv) \]
Modularity problem

\[
\begin{align*}
\text{await } x(v) \text{ in} \\
\text{emit } y \left( f \left( g \left( v \right) \right) \right)
\end{align*}
\]

signal \( z \) in
\[
\begin{align*}
\text{await } x(v) \text{ in} \\
\text{emit } z \left( g \left( v \right) \right) \\
\| \text{await } z(v) \text{ in} \\
\text{emit } y \left( f \left( v \right) \right)
\end{align*}
\]
Modularity problem
Modularity problem

\[
\begin{align*}
\text{signal } z \text{ in} \\
& \quad \text{await } x(v) \text{ in} \\
& \quad \quad \text{emit } z(g \ v) \\
& \quad || \text{ await } z(v) \text{ in} \\
& \quad \quad \text{emit } y(f \ v)
\end{align*}
\]

\[
\begin{align*}
\text{await } x(v) \text{ in} \\
\text{emit } y(f(g \ v))
\end{align*}
\]
Modularity problem

\[
\begin{align*}
\text{signal } z \text{ in} & \quad \text{await } x(v) \text{ in} \\
& \quad \text{emit } z(g\ v) \\
& \quad || \quad \text{await } z(v) \text{ in} \\
& \quad \text{emit } y(f\ v)
\end{align*}
\]

\[
\begin{align*}
\text{await } x(v) \text{ in} & \quad \text{emit } y(f\ (g\ v))
\end{align*}
\]

\[
f \circ g
\]

\[
\begin{align*}
x & \quad \rightarrow \quad f(g\ v) \\
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
x \\
\rightarrow \\
f
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
v \\
\rightarrow \\
f(g\ v)
\end{array}
\end{align*}
\]
Modularity problem

\[
\begin{align*}
\text{await } x(v) \text{ in} & \quad \text{emit } y(f(gv)) \\
\phantom{\text{await } x(v) \text{ in} } & \quad \downarrow \\
\phantom{\text{emit } y(f(gv))} & \quad f(gv)
\end{align*}
\]

\[
\begin{align*}
\text{signal } z \text{ in} & \quad \text{await } x(v) \text{ in} \quad \text{emit } z(gv) \\
& \quad || \quad \text{await } z(v) \text{ in} \quad \text{emit } y(fv) \\
& \quad \downarrow \\
& \quad f(gv)
\end{align*}
\]
Modularity problem

Communicating takes time

- No causality analysis
- Enables dynamic creation of processes

Problems of compositionality

- It is hard to put processes with different rates in parallel
- Time refinement is hard
The proposed solution

Clock domains  *(aka ‘Clock refinement’, Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
  await x(v) in
  emit z(g v)
|| await z(v) in
  emit y(f v)
```
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```plaintext
signal z in
  await x(v) in
  emit z(g v)
|| await z(v) in
  emit y(f v)
```

v
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```plaintext
signal z in
    await x(v) in
    emit z(g v)
|| await z(v) in
    emit y(f v)
```

lundi 28 novembre 2011
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
   await x(v) in
   emit z(g v)
|| await z(v) in
   emit y(f v)
```

v

\[ f(g v) \]
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```plaintext
signal z in
    await x(v) in
    emit z(g v)
|| await z(v) in
    emit y(f v)
```

```plaintext
newclock ck in
    signal z at ck in
    await x(v) in
    emit z(g v)
|| await z(v) in
    emit y(f v)
```

---

lundi 28 novembre 2011
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünden et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```plaintext
signal z in
  await x(v) in
  emit z(g v)
|| await z(v) in
  emit y(f v)
```

```plaintext
newclock ck in
  signal z at ck in
    await x(v) in
    emit z(g v)
|| await z(v) in
    emit y(f v)
```

émonce 28 novembre 2011
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünden et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
  await x(v) in
  emit z(g v)
|| await z(v) in
  emit y(f v)
```

```
newclock ck in
  signal z at ck in
    await x(v) in
    emit z(g v)
  || await z(v) in
    emit y(f v)
```
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

\[
\begin{align*}
\text{signal } z \text{ in} \\
\quad \text{await } x(v) \text{ in} \\
\quad \text{emit } z(g \ v) \\
\quad \text{|| await } z(v) \text{ in} \\
\quad \text{emit } y(f \ v)
\end{align*}
\]

\[
\begin{align*}
\text{newclock } ck \text{ in} \\
\quad \text{signal } z \text{ at } ck \text{ in} \\
\quad \text{await } x(v) \text{ in} \\
\quad \text{emit } z(g \ v) \\
\quad \text{|| await } z(v) \text{ in} \\
\quad \text{emit } y(f \ v)
\end{align*}
\]
The proposed solution

Clock domains (aka ‘Clock refinement’, Gemünde et al, Synchron 2009)

- Local notion of instant
- Hides internal steps from the outside

\[
\begin{align*}
signal z \ in \\
\quad \text{await } x(v) \ in \\
\quad \text{emit } z(g \ v) \\
\quad \lor \ \text{await } z(v) \ in \\
\quad \text{emit } y(f \ v)
\end{align*}
\]

\[
\begin{align*}
\text{newclock } ck \ in \\
\quad signal z \ at \ ck \ in \\
\quad \text{await } x(v) \ in \\
\quad \text{emit } z(g \ v) \\
\quad \lor \ \text{await } z(v) \ in \\
\quad \text{emit } y(f \ v)
\end{align*}
\]
Some examples

let process f s_out =
newclock ck in
loop
  emit s_out 1; pause ck;
  emit s_out 2; pause ck;
  emit s_out 3; pause topck
end

A periodic clock domain

- Each instant of the top clock, \texttt{ck} does three steps
A periodic clock domain

- Each instant of the top clock, \texttt{ck} does three steps
A periodic clock domain

- Each instant of the top clock, \texttt{ck} does three steps
Some examples

```haskell
let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, \texttt{ck} does three steps
A periodic clock domain

- Each instant of the top clock, `ck` does three steps

```cpp
let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end
```
Some examples

```plaintext
let process f s_out =
newclock ck in
loop
  emit s_out 1; pause ck;
  emit s_out 2; pause ck;
  emit s_out 3; pause topck
end
```

A periodic clock domain

- Each instant of the top clock, \( ck \) does three steps
Some examples

```ocaml
define process { let s_out = newclock ck in loop emit s_out 1; pause ck; emit s_out 2; pause ck; emit s_out 3; pause topck end }
```

A periodic clock domain

- Each instant of the top clock, \texttt{ck} does three steps
Some examples

```ocaml
let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, \( ck \) does three steps

---

lundi 28 novembre 2011
Some examples

```plaintext
let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, ck does three steps
A periodic clock domain

- Each instant of the top clock, `ck` does three steps

```plaintext
let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end
```
Some examples

```plaintext
let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, `ck` does three steps
Some examples

let process f s_out =
  newclock ck in
  loop
    emit s_out 1; pause ck;
    emit s_out 2; pause ck;
    emit s_out 3; pause topck
  end

let process g s_out =
  loop
    emit s_out 1;
    emit s_out 2;
    emit s_out 3; pause
  end

A periodic clock domain

- Each instant of the top clock, \textit{ck} does three steps
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
  ||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
Some examples

```ocaml
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
```

lundi 28 novembre 2011
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
Some examples

let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
Some examples

```plaintext
let process p s_out =
    newclock ck1 in
    loop
        emit s_out 1; pause ck1; emit s_out 2; pause ck1;
        emit s_out 3; pause topck
    end
 ||
    newclock ck2 in
    loop
        emit s_out 4; pause topck; emit s_out 5; pause ck2;
        emit s_out 6; pause topck
    end
```

lundi 28 novembre 2011
Some examples

```plaintext
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
end
```

---

lundi 28 novembre 2011
let process p s_out =
    newclock ck1 in
    loop
        emit s_out 1; pause ck1; emit s_out 2; pause ck1;
        emit s_out 3; pause topck
    end
    ||
    newclock ck2 in
    loop
        emit s_out 4; pause topck; emit s_out 5; pause ck2;
        emit s_out 6; pause topck
    end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
    newclock ck1 in
    loop
        emit s_out 1; pause ck1; emit s_out 2; pause ck1;
        emit s_out 3; pause topck
    end

||
newclock ck2 in
    loop
        emit s_out 4; pause topck; emit s_out 5; pause ck2;
        emit s_out 6; pause topck
    end
let process p s_out =

newclock ck1 in

loop
  emit s_out 1; pause ck1; emit s_out 2; pause ck1;
  emit s_out 3; pause topck
end

||

newclock ck2 in

loop
  emit s_out 4; pause topck; emit s_out 5; pause ck2;
  emit s_out 6; pause topck
end
Some examples

```plaintext
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
```

||
let process p s_out =
    newclock ck1 in
    loop
        emit s_out 1; pause ck1; emit s_out 2; pause ck1;
        emit s_out 3; pause topck
    end
||
newclock ck2 in
    loop
        emit s_out 4; pause topck; emit s_out 5; pause ck2;
        emit s_out 6; pause topck
    end
let process p s_out =
newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
 ||
  newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
Some examples

```plaintext
let process p s_out =
newclock ck1 in
  loop
  emit s_out 1; pause ck1; emit s_out 2; pause ck1;
  emit s_out 3; pause topck
end
||
newclock ck2 in
  loop
  emit s_out 4; pause topck; emit s_out 5; pause ck2;
  emit s_out 6; pause topck
end
```

lundi 28 novembre 2011
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
    newclock ck1 in
    loop
        emit s_out 1; pause ck1; emit s_out 2; pause ck1;
        emit s_out 3; pause topck
    end

||
newclock ck2 in
loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
end
let process p s_out =
    newclock ck1 in
    loop
        emit s_out 1; pause ck1; emit s_out 2; pause ck1;
        emit s_out 3; pause topck
    end
||
newclock ck2 in
    loop
        emit s_out 4; pause topck; emit s_out 5; pause ck2;
        emit s_out 6; pause topck
    end
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
  ||
  newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
let process p s_out =
 newclock ck1 in
 loop
   emit s_out 1; pause ck1; emit s_out 2; pause ck1;
   emit s_out 3; pause topck
 end
||
newclock ck2 in
 loop
   emit s_out 4; pause topck; emit s_out 5; pause ck2;
   emit s_out 6; pause topck
 end
let process p s_out =
newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
Some examples

```
let process p s_out =
  newclock ck1 in
  loop
    emit s_out 1; pause ck1; emit s_out 2; pause ck1;
    emit s_out 3; pause topck
  end
  ||
newclock ck2 in
  loop
    emit s_out 4; pause topck; emit s_out 5; pause ck2;
    emit s_out 6; pause topck
  end
```

![Diagram showing the process]

---

*lundi 28 novembre 2011*
Some examples

```
let process f compute s_in s_out =
  newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end

f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts
Some examples

```ocaml
let process f compute s_in s_out =
newclock ck in
loop
  await s_in(v) in
  let res = run (compute ck v) in
  emit s_out res
end

f : (clock -> 'a -> 'b process) ->
  ('c, 'a) event -> ('b, 'd) event -> unit process

- Same external behaviour no matter how long compute lasts
```
let process f compute s_in s_out = newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
end

f : (clock -> 'a -> 'b process) ->
  ('c, 'a) event -> ('b, 'd) event -> unit process

- Same external behaviour no matter how long compute lasts
Some examples

```ocaml
let process f compute s_in s_out =
  newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end

f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts
Some examples

```ocaml
let process f compute s_in s_out =
  newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end

f : (clock -> 'a -> 'b process) ->
  ('c, 'a) event -> ('b, 'd) event -> unit process

- Same external behaviour no matter how long compute lasts
```
Some examples

```haskell
let process f compute s_in s_out =
    newclock ck in
    loop
        await s_in(v) in
        let res = run (compute ck v) in
        emit s_out res
end
```

\[ f : (\text{clock} \rightarrow \text{'a} \rightarrow \text{'b} \ \text{process}) \rightarrow \)
\[ ('\text{c}', \text{'a}) \text{ event} \rightarrow ('\text{b}', \text{'d}) \text{ event} \rightarrow \text{unit} \ \text{process} \]

- Same external behaviour no matter how long compute lasts

---

lundi 28 novembre 2011
Some examples

```ocaml
let process f compute s_in s_out =
  newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end

f : (clock -> 'a -> 'b process) ->
  ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts
Some examples

```ocaml
going f compute s_in s_out =
newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
end
```

\[ f : (\text{clock} \to \ 'a \to \ 'b \ \text{process}) \to (\ 'c, \ 'a) \ \text{event} \to (\ 'b, \ 'd) \ \text{event} \to \text{unit} \ \text{process} \]

- Same external behaviour no matter how long compute lasts
signal s in
ewclock ck1 in ( await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in emit s
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0  
)
||
newclock ck2 in
emit s
Clocks and data dependencies

```haskell
signal s in
newclock ck1 in (  
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
emit s
```
Clocks and data dependencies

```plaintext
signal s in
ewclock ck1 in (  
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```
Clocks and data dependencies

```plaintext
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
  )
||
newclock ck2 in
emit s
```

Clocks and data dependencies
Clocks and data dependencies

```javascript
signal s in
ewclock ck1 in (  
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
emit s
```

### Topck

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ck2</td>
<td>ck1</td>
<td>ck1</td>
<td>ck1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Clocks and data dependencies

```c
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
||
newclock ck2 in
emit s
```
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
Clocks and data dependencies

```plaintext
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
  ||
newclock ck2 in
  emit s
```
signal s in
newclock ck1 in (  
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in  
emit s

signal s in
newclock ck1 in (  
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in  
emit s
Clocks and data dependencies

```plaintext
signal s in
newclock ck1 in (
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
```

```plaintext
signal s in
newclock ck1 in (
    await s; print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
```
Clocks and data dependencies

```
signal s in
newclock ck1 in (  
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

```
signal s in
newclock ck1 in (  
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
  )
  ||
newclock ck2 in
  emit s
```
Clocks and data dependencies

```plaintext
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
```

```plaintext
signal s in
newclock ck1 in (  
    await s; print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
```
Clocks and data dependencies

```plaintext
signal s in
clock ck1 in (  
  await immediate s;
  print_int 1
||
  pause ck1; pause ck1;
  print_int 0
)
||
| newclock ck2 in
  emit s
```

```plaintext
signal s in
clock ck1 in (  
  await s; print_int 1
||
  pause ck1; pause ck1;
  print_int 0
)
||
| newclock ck2 in
  emit s
```

(lundi 28 novembre 2011)
Clocks and data dependencies

```plaintext
signal s in
newclock ck1 in (  
    await immediate s;
    print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
emit s
```

```plaintext
signal s in
newclock ck1 in (  
    await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
emit s
```
Clocks and data dependencies

signal s in
newclock ck1 in ( 
    await immediate s;
    print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s

signal s in
newclock ck1 in ( 
    await s; print_int 1
    ||
    pause ck1; pause ck1;
    print_int 0
)
||
newclock ck2 in
emit s
信号 s 在
新时钟 ck1 在 (
  等待 immediate s;
  print_int 1
 ||
  pause ck1; pause ck1;
  print_int 0
)
||
新时钟 ck2 在
发出 s
Clock domains

The clocks tree

```plaintext
newclock ck in
    newclock ck2 in p2
    ||
    newclock ck3 in p3
```

Execution of a clock domain

- Do one step of the internal clock
- If all processes are waiting on a slower clock, wait for the next instant of the parent clock
- Otherwise, do another step
- Dynamic, unbounded number of steps

lundi 28 novembre 2011
Clocks and Reactivity

An uncooperative clock domain

\[
\text{let process p s\_out =}
\text{newclock ck in}
\text{loop}
\text{emit s\_out 3; pause ck}
\text{end}
\]
An uncooperative clock domain

```plaintext
let process p s_out =
    newclock ck in
    loop
        emit s_out 3; pause ck
    end
```
Clocks and Reactivity

An uncooperative clock domain

```ocaml
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
```

lundi 28 novembre 2011
An uncooperative clock domain

```
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
```
An uncooperative clock domain

```plaintext
let process p s_out =
    newclock ck in
    loop
        emit s_out 3; pause ck
    end
```
An uncooperative clock domain

```plaintext
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
```

![Clock diagram]

lundi 28 novembre 2011
An uncooperative clock domain

```haskell
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
```

---

**Clocks and Reactivity**

lundi 28 novembre 2011
Pausing a clock domain

- **pauseclock** \( ck \) forces the clock domain \( ck \) to wait for the next instant of its parent clock

```plaintext
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
  ||
  loop
    pause ck; pause ck; pauseclock ck
  end
```

lundi 28 novembre 2011
Pausing a clock domain

- **pauseclock** \( ck \) forces the clock domain \( ck \) to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
  loop
    pause ck; pause ck; pauseclock ck
  end
```
Pausing a clock domain

- **pauseclock** **ck** forces the clock domain **ck** to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
  loop
    pause ck; pause ck; pauseclock ck
  end
```

**topck**
Pausing a clock domain

- `pauseclock ck` forces the clock domain `ck` to wait for the next instant of its parent clock

```plaintext
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
loop
  pause ck; pause ck; pauseclock ck
end
```
Pausing a clock domain

- `pauseclock ck` forces the clock domain `ck` to wait for the next instant of its parent clock.

```latex
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
loop
  pause ck; pause ck; pauseclock ck
end
```

**topck**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ck</td>
<td>3</td>
</tr>
<tr>
<td>ck</td>
<td>3</td>
</tr>
</tbody>
</table>
Pausing a clock domain

- `pauseclock ck` forces the clock domain `ck` to wait for the next instant of its parent clock

```haskell
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
  loop
    pause ck; pause ck; pauseclock ck
  end
```

Clocks and Reactivity
Pausing a clock domain

- `pauseclock ck` forces the clock domain `ck` to wait for the next instant of its parent clock

```plaintext
let process p s_out =
    newclock ck in
    loop
        emit s_out 3; pause ck
    end
    ||
    loop
        pause ck; pause ck; pauseclock ck
    end
```
Pausing a clock domain

- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock.

```plaintext
let process p s_out =
    newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

![Diagram of clock domains]
Pausing a clock domain

- **pauseclock** `ck` forces the clock domain `ck` to wait for the next instant of its parent clock

```plaintext
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
  loop
    pause ck; pause ck; pauseclock ck
  end
```

### Figure

![Clocks and Reactivity](image)
Pausing a clock domain

- `pauseclock ck` forces the clock domain `ck` to wait for the next instant of its parent clock.

```plaintext
let process p s_out =
  newclock ck in
  loop
    emit s_out 3; pause ck
  end
||
  loop
    pause ck; pause ck; pauseclock ck
  end
```

![Diagram](image-url)
A signal has a clock

- Syntax: `signal s at ck in e`
- It cannot be used on a slower clock
- Type and effect system to prevent the use of a signal outside of its clock domain
Parallelization is done at the level of clock domains

- Less synchronizations
- Locality of signals

```plaintext
clock ck in
newclock ck2 in p2
||
newclock ck3 in p3
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
In a data-flow setting

node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

a₀
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
c = true fby false fby c;
x = 0 fby ((merge c a (x whenot c)) + 1);
o = x when c
tel
```

lundi 28 novembre 2011
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x when not c)) + 1);
  o = x when c
tel
```

a0

↓

↓

↓

0
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c

tel
```

a₀  a₁
    ↓  ↓
    |  |
    ↓  ↓
₀
In a data-flow setting

\[
\begin{align*}
\text{node } f(a : \text{int}) &= (o : \text{int}) \\
\text{var } x : \text{int}; c : \text{bool}; \\
\text{let } \\
\quad c &= \text{true fby false fby c}; \\
\quad x &= 0 \ fby ((\text{merge c a (x whenot c)}) + 1); \\
\quad o &= x \ \text{when c} \\
\end{align*}
\]
In a data-flow setting

```plaintext
node f(a : int) = (o : int)

var x : int; c : bool;

let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whennot c)) + 1);
  o = x when c

tel
```

```
\begin{array}{c c c}
a_0 & a_1 \\
\downarrow & \downarrow & \downarrow \\
\hline & & \\
0 & a_0+2 \\
\end{array}
```
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
c = true fby false fby c;
x = 0 fby ((merge c a (x whenot c)) + 1);
o = x when c
```

```
a_0  a_1  a_2
```

```
0     a_0+2
```
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
```

\[
\begin{array}{c|c|c}
a_0 & a_1 & a_2 \\
\hline
0 & a_0+2 & a_1+2 \\
\end{array}
\]
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
```

\[
f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c
\]

\[a_0 \quad a_1 \quad a_2 \]

\[0 \quad a_0 + 2 \quad a_1 + 2\]

lundi 28 novembre 2011
In a data-flow setting

```java
node f(a : int) = (o : int)
var x : int; c : bool;
let
    c = true fby false fby c;
    x = 0 fby ((merge c a (x whenot c)) + 1);
    o = x when c
```

\[
f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c
\]
In a data-flow setting

```ocaml
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

\[ f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c \]
In a data-flow setting

\[
\text{node } f(a : \text{ int}) = (o : \text{ int})
\]
\[
\text{var } x : \text{ int}; c : \text{ bool};
\]
\[
\text{let}
\]
\[
\begin{align*}
    c &= \text{ true fby false fby c;} \\
    x &= 0 \text{ fby ((merge c a (x whenot c)) + 1)}; \\
    o &= x \text{ when c}
\end{align*}
\]
\[
\text{tel}
\]

\[
f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c
\]
In a data-flow setting

node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c

\[
\begin{align*}
\node & f(a : \text{int}) = (o : \text{int}) \\
\text{var} & x : \text{int}; c : \text{bool}; \\
\text{let} & \\
& c = \text{true} \ fby \ \text{false} \ fby \ c; \\
& x = 0 \ fby \ ((\text{merge} \ c \ a \ (x \ \text{whenot} \ c)) + 1); \\
& o = x \ \text{when} \ c \\
\end{align*}
\]

\[
f :: \alpha \ \text{on} \ c \xrightarrow{\alpha} \alpha \ \text{on} \ c
\]
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x when not c)) + 1);
  o = x when c
tel

\[
f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c
\]
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
    c = true fby false fby c;
    x = 0 fby ((merge c a (x when not c)) + 1);
    o = x when c
```

\[ f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c \]
In a data-flow setting

```python
node f(a : int) = (o : int)
var x : int; c : bool;
let
    c = true fby false fby c;
    x = 0 fby ((merge c a (x whennot c)) + 1);
    o = x when c
tel
```

\[ f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c \]
In a data-flow setting

```plaintext
node f(a : int) = (o : int)
var x : int; c : bool;
let
c = true fby false fby c;
x = 0 fby ((merge c a (x whennot c)) + 1);
o = x when c
tel
```

\[
f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c
\]

\[
f :: \beta \xrightarrow{\beta} \beta
\]
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
    c = true fby false fby c;
    x = 0 fby ((merge c a (x when not c)) + 1);
    o = x when c
tel
```

```
int f_step (int a) {
    do {
        ...
        if(c)
            o = ...
        ...
    } while(!c)
    return o;
}
```
Demo: n-body with multiple steps

Multiple step integration methods

- The computation of the next position is done in multiple steps, shared by all planets
- Internal steps hidden by a clock domain
- Can easily and transparently switch between methods
- eg Runge-Kutta:

\[
y' = f(t, y) \\
y(t_0) = y_0 \\
y_{n+1} = y_n + \frac{1}{6} (k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 = hf(t_n, y_n) \\
k_2 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\
k_3 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\
k_4 = hf(t_n + h, y_n + k_3)
\]
let process compute_a env x w =
    emit env (force (x, w));
  await env(f) in
  f x

let process compute_rk4 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
  ....
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 clock domains
- Still one step on the top clock
Current and Future Work

Done so far

- Operational semantics
- Proof of concept of type system
- Sequential implementation

Future work

- Distributed implementation (message passing)
New notion of clock domain

- Improves modularity
- Helps parallelization
- Little changes to the language and runtime