

# Une sémantique de Kahn mécanisée pour les machines à états

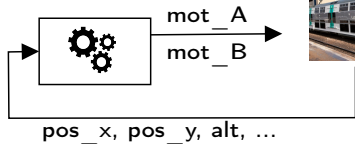
Timothy Bourke<sup>1</sup>   Paul Jeanmaire<sup>2</sup>   Marc Pouzet<sup>1</sup>

1. Inria Paris, École normale supérieure

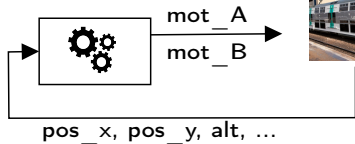
2. LIP6, Sorbonne Université

JFLA, 27 janvier 2026

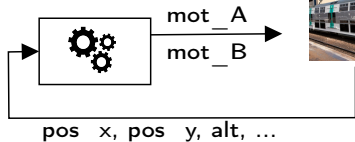
# Programmation réactive de systèmes embarqués



# Programmation réactive de systèmes embarqués

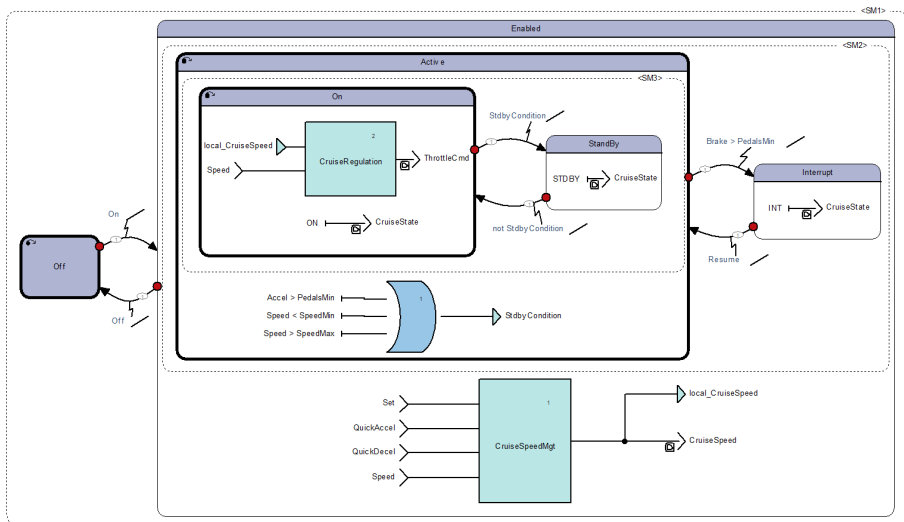


# Programmation réactive de systèmes embarqués



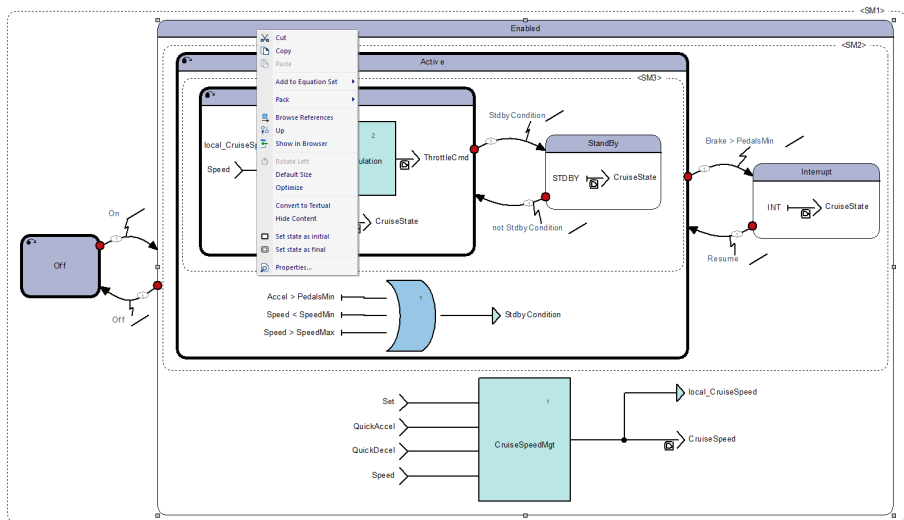
```
every trigger do :  
  read_inputs()  
  compute()  
  write_outputs()
```

# Diagrammes de schéma-blocs



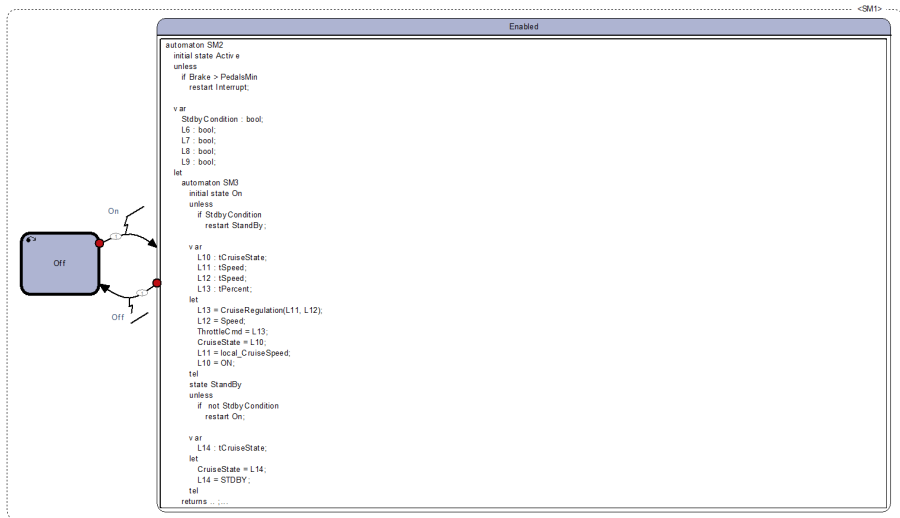
Scade Suite, régulateur de vitesse

# Diagrammes de schéma-blocs



Scade Suite, régulateur de vitesse

# Diagrammes de schéma-blocs



Lustre/Scade : langage synchrone flots de données











# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck (count_down((edge, n) when ck))
  | | | (0 when not ck);
  o = v > 0;
tel
```

programme = ensemble d'équations

# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck (count_down((edge, n) when ck))
    | | (0 when not ck);
  o = v > 0;
tel
```

i	F	T	T	T	F	F	F	T	F	T	F	F	F	F	...
n	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...
edge	F	T	F	F	F	F	F	T	F	T	F	F	F	F	...
ck	F	T	T	T	T	F	F	T	T	T	T	T	T	F	...
edge when ck		T	F	F	F			T	F	T	F	F	F		...
count_down(...)		3	2	1	0			3	2	3	2	1	0		...
v	0	3	2	1	0	0	0	3	2	3	2	1	0	0	...
o	F	T	T	T	F	F	F	T	T	T	T	T	F	F	...

programme = ensemble d'équations

sémantique = solution aux équations (point fixe)

# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck (count_down((edge, n) when ck))
    | | (0 when not ck);
  o = v > 0;
tel
```

i	F	T	T	T	F	F	F	T	F	T	F	F	F	F	...
n	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...
edge	F	T	F	F	F	F	T	F	T	F	F	F	F	F	...
ck	F	T	T	T	T	F	F	T	T	T	T	T	T	F	...
edge when ck		T	F	F	F			T	F	T	F	F	F		...
count_down(...)		3	2	1	0			3	2	3	2	1	0		...
v	0	3	2	1	0	0	0	3	2	3	2	1	0	0	...
o	F	T	T	T	F	F	F	T	T	T	T	T	F	F	...

programme = ensemble d'équations

sémantique = solution aux équations (point fixe)

variables à valeurs dans  $D^\omega$ , CPO des flots (in-)finis

# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
```

```
node rising_edge_retrigger(i : bool; n : int)
```

```
returns (o : bool)
```

```
var edge, ck : bool; v : int;
```

```
let
```

```
  edge = i and (false fby (not i));
```

```
  ck = edge or (false fby o);
```

```
  v = merge ck (count_down((edge, n) when ck))
```

```
    | | | (0 when not ck);
```

```
  o = v > 0;
```

```
tel
```

i	F	T	T	T	F	F	F	T	F	T	F	F	F	F	...
n	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...
edge	F	T	F	F	F	F	T	F	T	F	F	F	F	F	...
ck	F	T	T	T	T	F	F	T	T	T	T	T	T	F	...
edge when ck		T	F	F	F			T	F	T	F	F	F		...
count_down(...)		3	2	1	0			3	2	3	2	1	0		...
v	0	3	2	1	0	0	0	3	2	3	2	1	0	0	...
o	F	T	T	T	F	F	F	T	T	T	T	T	F	F	...

programme = ensemble d'équations

sémantique = solution aux équations (point fixe)

variables à valeurs dans  $D^\omega$ , CPO des flots (in-)finis

opérateurs continus  $D^\omega \rightarrow_c D^\omega \rightarrow_c D^\omega$  :

# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
```

```
node rising_edge_retrigger(i : bool; n : int)
```

```
returns (o : bool)
```

```
var edge, ck : bool; v : int;
```

```
let
```

```
  edge = i and (false fby (not i));
```

```
  ck = edge or (false fby 0);
```

```
  v = merge ck (count_down((edge, n) when ck))
```

```
    | | | (0 when not ck);
```

```
  o = v > 0;
```

```
tel
```

i	F	T	T	T	F	F	F	T	F	T	F	F	F	F	...
n	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...
edge	F	T	F	F	F	F	T	F	T	F	F	F	F	F	...
ck	F	T	T	T	T	F	F	T	T	T	T	T	T	F	...
edge when ck		T	F	F	F			T	F	T	F	F	F		...
count_down(...)		3	2	1	0			3	2	3	2	1	0		...
v	0	3	2	1	0	0	0	3	2	3	2	1	0	0	...
o	F	T	T	T	F	F	F	T	T	T	T	T	F	F	...

programme = ensemble d'équations

sémantique = solution aux équations (point fixe)

variables à valeurs dans  $D^\omega$ , CPO des flots (in-)finis

opérateurs continus  $D^\omega \rightarrow_c D^\omega \rightarrow_c D^\omega$  :

$(x.xs) + (y.ys) \stackrel{\text{def}}{=} (x + y).(xs + ys)$

# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
```

```
node rising_edge_retrigger(i : bool; n : int)
```

```
returns (o : bool)
```

```
var edge, ck : bool; v : int;
```

```
let
```

```
  edge = i and (false fby (not i));
```

```
  ck = edge or (false fby o);
```

```
  v = merge ck (count_down((edge, n) when ck))
```

```
    | | (0 when not ck);
```

```
  o = v > 0;
```

```
tel
```

i	F	T	T	T	F	F	F	T	F	T	F	F	F	F	...
n	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...
edge	F	T	F	F	F	F	F	T	F	T	F	F	F	F	...
ck	F	T	T	T	T	F	F	T	T	T	T	T	T	F	...
edge when ck		T	F	F	F			T	F	T	F	F	F		...
count_down(...)		3	2	1	0			3	2	3	2	1	0		...
v	0	3	2	1	0	0	0	3	2	3	2	1	0	0	...
o	F	T	T	T	F	F	F	T	T	T	T	T	F	F	...

programme = ensemble d'équations

sémantique = solution aux équations (point fixe)

variables à valeurs dans  $D^\omega$ , CPO des flots (in-)finis

opérateurs continus  $D^\omega \rightarrow_c D^\omega \rightarrow_c D^\omega$  :

$$(x.xs) + (y.y_s) \stackrel{\text{def}}{=} (x + y).(xs + ys)$$

$$(x.xs) \text{ fby } (y.y_s) \stackrel{\text{def}}{=} x.y.y_s$$

...

# Le modèle de Kahn

## En théorie

```
-- Keeps output `o` true for `n` iteration after a rising edge on `i`.
```

```
node rising_edge_retrigger(i : bool; n : int)
```

```
returns (o : bool)
```

```
var edge, ck : bool; v : int;
```

```
let
```

```
  edge = i and (false fby (not i));
```

```
  ck = edge or (false fby o);
```

```
  v = merge ck (count_down((edge, n) when ck))
```

```
    | | | (0 when not ck);
```

```
  o = v > 0;
```

```
tel
```

i	F	T	T	T	F	F	F	T	F	T	F	F	F	F	...
n	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...
edge	F	T	F	F	F	F	T	F	T	F	F	F	F	F	...
ck	F	T	T	T	F	F	T	T	T	T	T	T	T	F	...
edge when ck		T	F	F	F		T	F	T	F	F	F	F		...
count_down(...)		3	2	1	0		3	2	3	2	1	0			...
v	0	3	2	1	0	0	0	3	2	3	2	1	0	0	...
o	F	T	T	T	F	F	T	T	T	T	T	F	F		...

programme = ensemble d'équations

sémantique = solution aux équations (point fixe)

variables à valeurs dans  $D^\omega$ , CPO des flots (in-)finis

opérateurs continus  $D^\omega \rightarrow_c D^\omega \rightarrow_c D^\omega$  :

$$(x.xs) + (y.y_s) \stackrel{\text{def}}{=} (x + y).(xs + ys)$$

$$(x.xs) \text{ fby } (y.y_s) \stackrel{\text{def}}{=} x.y.y_s$$

...

point fixe construit par itérations depuis  $\perp = \epsilon$

# Le modèle de Kahn

Prototypage en OCaml avec bibliothèque **lazy**

```
(** type des flots *)  
type 'a lazy_stream = 'a stream Lazy.t  
and 'a stream = | Cons : 'a * 'a lazy_stream -> 'a stream
```

# Le modèle de Kahn

Prototypage en OCaml avec bibliothèque **lazy**

```
(** type des flots *)
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream = | Cons : 'a * 'a lazy_stream -> 'a stream

(** application point à point *)
let rec map (f : ('a -> 'b)) (x : 'a lazy_stream) : 'b lazy_stream =
  lazy (match Lazy.force x with
  | Cons(a,x) -> (Cons(f a, map f x)))
```

# Le modèle de Kahn

Prototypage en OCaml avec bibliothèque **lazy**

```
(** type des flots *)
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream = | Cons : 'a * 'a lazy_stream -> 'a stream

(** application point à point *)
let rec map (f : ('a -> 'b)) (x : 'a lazy_stream) : 'b lazy_stream =
  lazy (match Lazy.force x with
    | Cons(a,x) -> (Cons(f a, map f x)))

(** délai initialisé *)
let fby : 'a lazy_stream -> 'a lazy_stream -> 'a lazy_stream =
  fun x y -> lazy (match Lazy.force x with
    | Cons(a,_) -> Cons(a,y))
```

# Le modèle de Kahn

Prototypage en OCaml avec bibliothèque `lazy`

```
(** type des flots *)
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream = | Cons : 'a * 'a lazy_stream -> 'a stream

(** application point à point *)
let rec map (f : ('a -> 'b)) (x : 'a lazy_stream) : 'b lazy_stream =
  lazy (match Lazy.force x with
    | Cons(a,x) -> (Cons(f a, map f x)))

(** délai initialisé *)
let fby : 'a lazy_stream -> 'a lazy_stream -> 'a lazy_stream =
  fun x y -> lazy (match Lazy.force x with
    | Cons(a,_) -> Cons(a,y))

(** combinateurs de point fixe *)
let fix (f : 'a lazy_stream -> 'a lazy_stream) : 'a lazy_stream =
  let rec y = lazy (Lazy.force (f y)) in y
```

# Le modèle de Kahn

Prototypage en OCaml avec bibliothèque `lazy`

```
(** type des flots *)
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream = | Cons : 'a * 'a lazy_stream -> 'a stream

(** application point à point *)
let rec map (f : ('a -> 'b)) (x : 'a lazy_stream) : 'b lazy_stream =
  lazy (match Lazy.force x with
    | Cons(a,x) -> (Cons(f a, map f x)))

(** délai initialisé *)
let fby : 'a lazy_stream -> 'a lazy_stream -> 'a lazy_stream =
  fun x y -> lazy (match Lazy.force x with
    | Cons(a,_) -> Cons(a,y))

(** combinateurs de point fixe *)
let fix (f : 'a lazy_stream -> 'a lazy_stream) : 'a lazy_stream =
  let rec y = lazy (Lazy.force (f y)) in y

(** combinateurs de point fixe sur les environnements *)
let fix_env (f : 'a env -> 'a env) : 'a env =
  let rec y = fun i -> lazy (Lazy.force (f y i)) in y
```

# Le modèle de Kahn

En Rocq

*Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

# Le modèle de Kahn

En Rocq

*Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

Encodage des flots (finis et infinis) et de l'ordre associé

# Le modèle de Kahn

En Rocq

## *Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

Encodage des flots (finis et infinis) et de l'ordre associé

Calculs productifs grâce à un constructeur *Eps*

# Le modèle de Kahn

En Rocq

## *Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

Encodage des flots (finis et infinis) et de l'ordre associé

Calculs productifs grâce à un constructeur *Eps*

## Adaptation au compilateur [Bourke, Jeanmaire, Pouzet, 2025]

Langage noyau Lustre/Scade avec réinitialisation

# Le modèle de Kahn

En Rocq

## *Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

Encodage des flots (finis et infinis) et de l'ordre associé

Calculs productifs grâce à un constructeur *Eps*

## Adaptation au compilateur [Bourke, Jeanmaire, Pouzet, 2025]

Langage noyau Lustre/Scade avec réinitialisation

Traitement de la synchronisation et des erreurs d'exécution

# Le modèle de Kahn

En Rocq

## *Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

Encodage des flots (finis et infinis) et de l'ordre associé

Calculs productifs grâce à un constructeur *Eps*

## Adaptation au compilateur [Bourke, Jeanmaire, Pouzet, 2025]

Langage noyau Lustre/Scade avec réinitialisation

Traitement de la synchronisation et des erreurs d'exécution

Preuve d'adéquation avec le modèle relationnel de Vélus

# Le modèle de Kahn

En Rocq

## *Constructive denotational semantics for Kahn networks*

Bibliothèque de CPO, points fixes [Paulin-Mohring, 2009]

Encodage des flots (finis et infinis) et de l'ordre associé

Calculs productifs grâce à un constructeur *Eps*

## Adaptation au compilateur [Bourke, Jeanmaire, Pouzet, 2025]

Langage noyau Lustre/Scade avec réinitialisation

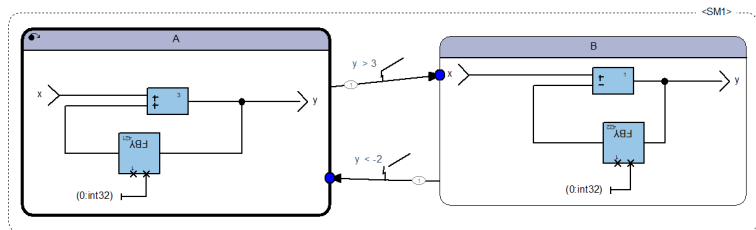
Traitement de la synchronisation et des erreurs d'exécution

Preuve d'adéquation avec le modèle relationnel de Vélus

**Aujourd'hui** : extension aux machines à états

# Transitions réinitialisantes

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

until  $y > 3$  then B

state B do

$y = (0 \text{ fby } y) - x;$

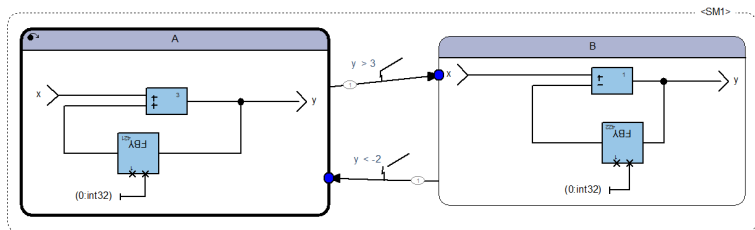
until  $y < -2$  then A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	0	1	3

# Transitions réinitialisantes

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

until  $y > 3$  then B

state B do

$y = (0 \text{ fby } y) - x;$

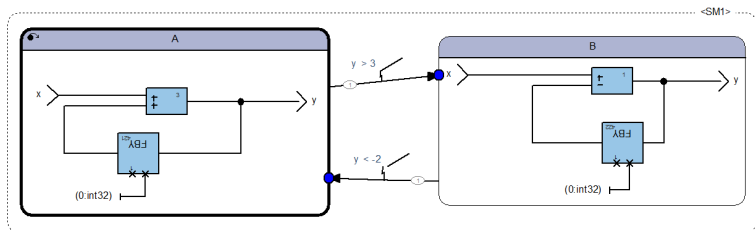
until  $y < -2$  then A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	0	1	3

# Transitions réinitialisantes

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

until  $y > 3$  then B

state B do

$y = (0 \text{ fby } y) - x;$

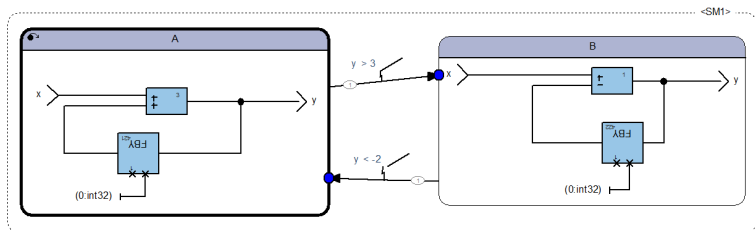
until  $y < -2$  then A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	0	1	3

# Transitions réinitialisantes

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

until  $y > 3$  then B

state B do

$y = (0 \text{ fby } y) - x;$

until  $y < -2$  then A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	0	1	3

# Transitions réinitialisantes

## Sémantique

x		1	2	1	0	1	2	0	1	2
---	--	---	---	---	---	---	---	---	---	---

---

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until y > 3 then B
state B do
  y = (0 fby y) - x;
  until y < -2 then A
end
```

---

```
let auto_reset i fs ft env =
  let f = fixp (fun f (i, env) →
    let env' = fs i env in
    let t = ft i env' in
    merge_until t env' (fun j → f (j, rem_until t env)))
  in f (i, env)
```

# Transitions réinitialisantes

## Sémantique

**automaton** initially A

**state A do**

$y = (0 \text{ fby } y) + x;$

**until**  $y > 3$  **then B**

**state B do**

$y = (0 \text{ fby } y) - x;$

**until**  $y < -2$  **then A**

**end**

x		1	2	1	0	1	2	0	1	2
y		1	3	4	4	5	7	7	8	10

**let** auto\_reset i fs ft env =

**let** f = fixp (**fun** f (i, env) →

**let** env' = fs i env **in**

**let** t = ft i env' **in**

merge\_until t env' (**fun** j → f (j, rem\_until t env)))

**in** f (i, env)

# Transitions réinitialisantes

## Sémantique

**automaton** initially A

**state A do**

$y = (0 \text{ fby } y) + x;$

**until**  $y > 3$  **then** B

**state B do**

$y = (0 \text{ fby } y) - x;$

**until**  $y < -2$  **then** A

**end**

x	1	2	1	0	1	2	0	1	2
y	1	3	4	4	5	7	7	8	10
t	N	N	B	B	B	B	B	B	B

**let** auto\_reset i fs ft env =

**let** f = fixp (**fun** f (i, env) →

**let** env' = fs i env **in**

**let** t = ft i env' **in**

merge\_until t env' (**fun** j → f (j, rem\_until t env)))

**in** f (i, env)

# Transitions réinitialisantes

## Sémantique

**automaton** initially A

**state A do**

$y = (0 \text{ fby } y) + x;$

**until**  $y > 3$  **then** B

**state B do**

$y = (0 \text{ fby } y) - x;$

**until**  $y < -2$  **then** A

**end**

x	1	2	1	0	1	2	0	1	2
y	1	3	4	4	5	7	7	8	10
t	N	N	B	B	B	B	B	B	B

**let** auto\_reset i fs ft env =

**let** f = fixp (**fun** f (i, env) →

**let** env' = fs i env **in**

**let** t = ft i env' **in**

    merge\_until t env' (**fun** j → f (j, rem\_until t env)))

**in** f (i, env)

# Transitions réinitialisantes

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until y > 3 then B
state B do
  y = (0 fby y) - x;
  until y < -2 then A
end
```

x	1	2	1	0	1	2	0	1	2
y	1	3	4	4	5	7	7	8	10
t	N	N	B	B	B	B	B	B	B
y				0	-1	-3	-3	-4	-6
t				N	N	A	A	A	A

```
let auto_reset i fs ft env =
  let f = fixp (fun f (i, env) →
    let env' = fs i env in
    let t = ft i env' in
    merge_until t env' (fun j → f (j, rem_until t env)))
  in f (i, env)
```

# Transitions réinitialisantes

## Sémantique

**automaton** initially A

**state A do**

  y = (0 fby y) + x;

**until** y > 3 **then** B

**state B do**

  y = (0 fby y) - x;

**until** y < -2 **then** A

**end**

x	1	2	1	0	1	2	0	1	2
y	1	3	4	4	5	7	7	8	10
t	N	N	B	B	B	B	B	B	B
y				0	-1	-3	-3	-4	-6
t				N	N	A	A	A	A
y							0	1	3
t							N	N	N

**let** auto\_reset i fs ft env =

**let** f = fixp (**fun** f (i, env) →

**let** env' = fs i env **in**

**let** t = ft i env' **in**

    merge\_until t env' (**fun** j → f (j, rem\_until t env)))

**in** f (i, env)

# Transitions réinitialisantes

## Sémantique

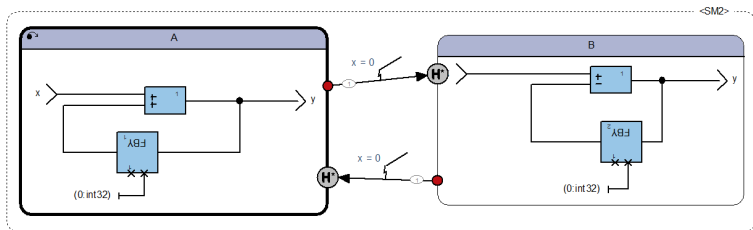
```
automaton initially A
state A do
  y = (0 fby y) + x;
  until y > 3 then B
state B do
  y = (0 fby y) - x;
  until y < -2 then A
end
```

x	1	2	1	0	1	2	0	1	2
y	1	3	4	4	5	7	7	8	10
t	N	N	B	B	B	B	B	B	B
y				0	-1	-3	-3	-4	-6
t				N	N	A	A	A	A
y							0	1	3
t							N	N	N
y	1	3	4	0	-1	-3	0	1	3

```
let auto_reset i fs ft env =
  let f = fixp (fun f (i, env) →
    let env' = fs i env in
    let t = ft i env' in
    merge_until t env' (fun j → f (j, rem_until t env)))
  in f (i, env)
```

# Transitions avec histoire

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

unless  $x = 0$  continue B

state B do

$y = (0 \text{ fby } y) - x;$

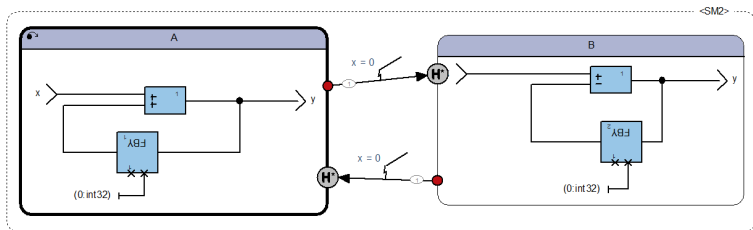
unless  $x = 0$  continue A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	4	5	7

# Transitions avec histoire

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

unless  $x = 0$  continue B

state B do

$y = (0 \text{ fby } y) - x;$

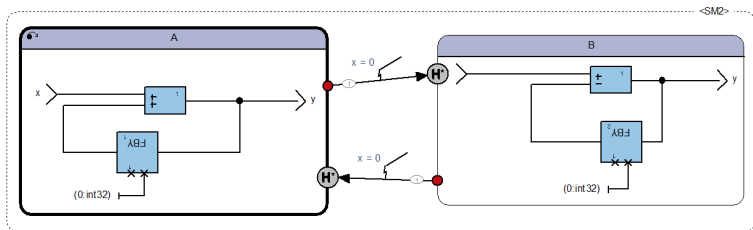
unless  $x = 0$  continue A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	4	5	7

# Transitions avec histoire

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

unless  $x = 0$  continue B

state B do

$y = (0 \text{ fby } y) - x;$

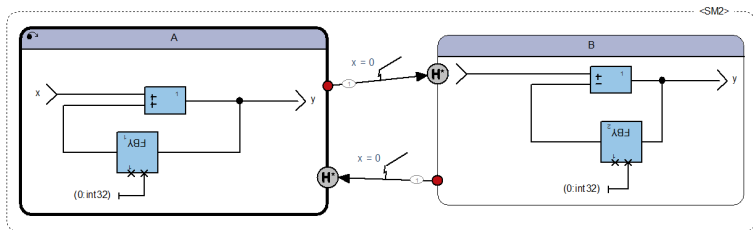
unless  $x = 0$  continue A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	4	5	7

# Transitions avec histoire

## Principe



automaton initially A

state A do

$y = (0 \text{ fby } y) + x;$

unless  $x = 0$  continue B

state B do

$y = (0 \text{ fby } y) - x;$

unless  $x = 0$  continue A

end

x	1	2	1	0	1	2	0	1	2
y	1	3	4	0	-1	-3	4	5	7

# Transitions avec histoire

## Sémantique

x		1	2	1	0	1	2	0	1	2
---	--	---	---	---	---	---	---	---	---	---

```
automaton initially A
state A do
  y = (0 fby y) + x;
  unless x = 0 continue B
state B do
  y = (0 fby y) - x;
  unless x = 0 continue A
end
```

---

```
let auto_continue i fs ft env =
  let rec ft' : id → id lazy_stream =
    fun i → map (Option.value ~default:i) (ft i (wheni i ts env))
  and ts = lazy (Cons (i, mergei ts ft')) in
  mergei (rem ts) (fun i → fs i (wheni i (rem ts) env))
```

# Transitions avec histoire

## Sémantique

**automaton** initially A

**state A do**

**y = (0 fby y) + x;**

**unless x = 0 continue B**

**state B do**

**y = (0 fby y) - x;**

**unless x = 0 continue A**

**end**

x		1	2	1	0	1	2	0	1	2
y		1	3	4				4	5	7

**let** auto\_continue i fs ft env =

**let rec** ft' : id → id lazy\_stream =

**fun** i → map (Option.value ~default:i) (ft i (wheni i ts env))

**and** ts = **lazy** (Cons (i, mergei ts ft')) **in**

mergei (rem ts) (**fun** i → fs i (wheni i (rem ts) env))

# Transitions avec histoire

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  unless x = 0 continue B
state B do
  y = (0 fby y) - x;
  unless x = 0 continue A
end
```

x	1	2	1	0	1	2	0	1	2
y	1	3	4				4	5	7
y				0	-1	-3			

```
let auto_continue i fs ft env =
  let rec ft' : id → id lazy_stream =
    fun i → map (Option.value ~default:i) (ft i (wheni i ts env))
  and ts = lazy (Cons (i, mergei ts ft')) in
  mergei (rem ts) (fun i → fs i (wheni i (rem ts) env))
```

# Transitions avec histoire

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  unless x = 0 continue B
state B do
  y = (0 fby y) - x;
  unless x = 0 continue A
end
```

x	1	2	1	0	1	2	0	1	2
y	1	3	4				4	5	7
y				0	-1	-3			
y	1	3	4	0	-1	-3	4	5	7

```
let auto_continue i fs ft env =
  let rec ft' : id → id lazy_stream =
    fun i → map (Option.value ~default:i) (ft i (wheni i ts env))
  and ts = lazy (Cons (i, mergei ts ft')) in
  mergei (rem ts) (fun i → fs i (wheni i (rem ts) env))
```

# Transitions avec histoire

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  unless x = 0 continue B
state B do
  y = (0 fby y) - x;
  unless x = 0 continue A
end
```

x	1	2	1	0	1	2	0	1	2
y	1	3	4				4	5	7
t	A	A	A	B				A	A
y				0	-1	-3			
t					B	B	A		
ts	A	A	A	B	B	B	A	A	A
y	1	3	4	0	-1	-3	4	5	7

```
let auto_continue i fs ft env =
  let rec ft' : id → id lazy_stream =
    fun i → map (Option.value ~default:i) (ft i (wheni i ts env))
  and ts = lazy (Cons (i, mergei ts ft')) in
  mergei (rem ts) (fun i → fs i (wheni i (rem ts) env))
```

# Transitions mixtes

## Sémantique

**automaton** initially A

**state A do**

  y = (0 **fb**y) + x;

**until** x = 0 **then** B

**state B do**

  y = (0 **fb**y) - x;

**until** y < -2 **continue** A

    | x = 0 **then** A

**end**

x		1	2	0	1	2	1	1	1	1
---	--	---	---	---	---	---	---	---	---	---

**let** auto i fs ft env =

**let** rec f (i, reset, env, hist, trim) =

**let** s := **if** reset **then** fs i env **else** trim i (fs i (hist i env)) **in**

**let** t := **if** reset **then** ft i s **else** trim i (ft i (hist i env)) **in**

    (\* update continuations \*)

**let** hist j env' = **if** j = i **then** hist i (merge\_until t s env') **else** hist j env' **in**

**let** trim j env' = **if** j = i **then** rem\_until t (trim i env') **else** trim j env' **in**

    merge\_until t s (**fun** (j, reset) → f̄ (j, reset, rem\_until t env, hist, trim)))

**in** f (i, false, env, (**fun** i env → env), (**fun** i s → s)).

# Transitions mixtes

## Sémantique

**automaton** initially A

**state A do**

$y = (0 \text{ fby } y) + x;$

**until**  $x = 0$  **then B**

**state B do**

$y = (0 \text{ fby } y) - x;$

**until**  $y < -2$  **continue** A

|  $x = 0$  **then A**

**end**

x	1	2	0	1	2	1	1	1	1
y	1	3	3						
t	N	N	$B^R$						

**let** auto i fs ft env =

**let** rec f (i, reset, env, hist, trim) =

**let** s := **if** reset **then** fs i env **else** trim i (fs i (hist i env)) **in**

**let** t := **if** reset **then** ft i s **else** trim i (ft i (hist i env)) **in**

(\* update continuations \*)

**let** hist j env' = **if** j = i **then** hist i (merge\_until t s env') **else** hist j env' **in**

**let** trim j env' = **if** j = i **then** rem\_until t (trim i env') **else** trim j env' **in**

merge\_until t s (**fun** (j, reset) →  $\bar{f}$  (j, reset, rem\_until t env, hist, trim)))

**in** f (i, false, env, (**fun** i env → env), (**fun** i s → s)).

# Transitions mixtes

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until x = 0 then B
state B do
  y = (0 fby y) - x;
  until y < -2 continue A
  | x = 0 then A
end
```

x	1	2	0	1	2	1	1	1	1
y	1	3	3						
t	N	N	$B^R$						
y				-1	-3				
t				N	A				

```
let auto i fs ft env =
  let rec f (i, reset, env, hist, trim) =
    let s := if reset then fs i env else trim i (fs i (hist i env)) in
    let t := if reset then ft i s else trim i (ft i (hist i env)) in
    (* update continuations *)
    let hist j env' = if j = i then hist i (merge_until t s env') else hist j env' in
    let trim j env' = if j = i then rem_until t (trim i env') else trim j env' in
    mergei_until t s (fun (j, reset) → f (j, reset, rem_until t env, hist, trim)))
  in f (i, false, env, (fun i env → env), (fun i s → s)).
```

# Transitions mixtes

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until x = 0 then B
state B do
  y = (0 fby y) - x;
  until y < -2 continue A
  | x = 0 then A
end
```

x	1	2	0	1	2	1	1	1	1
y	1	3	3						
t	N	N	$B^R$						
y				-1	-3				
t				N	A				
y	1	3	3			4	5	6	7
t	N	N	$B^R$			N	N	N	N

```
let auto i fs ft env =
  let rec f (i, reset, env, hist, trim) =
    let s := if reset then fs i env else trim i (fs i (hist i env)) in
    let t := if reset then ft i s else trim i (ft i (hist i env)) in
    (* update continuations *)
    let hist j env' = if j = i then hist i (merge_until t s env') else hist j env' in
    let trim j env' = if j = i then rem_until t (trim i env') else trim j env' in
    mergei_until t s (fun (j, reset) → f (j, reset, rem_until t env, hist, trim)))
  in f (i, false, env, (fun i env → env), (fun i s → s)).
```

# Transitions mixtes

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until x = 0 then B
state B do
  y = (0 fby y) - x;
  until y < -2 continue A
  | x = 0 then A
end
```

x	1	2	0	1	2	1	1	1	1
y	1	3	3						
t	N	N	$B^R$						
y				-1	-3				
t				N	A				
y	1	3	3			4	5	6	7
t	N	N	$B^R$			N	N	N	N

```
let auto i fs ft env =
  let rec f (i, reset, env, hist, trim) =
    let s := if reset then fs i env else trim i (fs i (hist i env)) in
    let t := if reset then ft i s else trim i (ft i (hist i env)) in
    (* update continuations *)
    let hist j env' = if j = i then hist i (merge_until t s env') else hist j env' in
    let trim j env' = if j = i then rem_until t (trim i env') else trim j env' in
    merge_until t s (fun (j, reset) → f (j, reset, rem_until t env, hist, trim)))
  in f (i, false, env, (fun i env → env), (fun i s → s)).
```

# Transitions mixtes

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until x = 0 then B
state B do
  y = (0 fby y) - x;
  until y < -2 continue A
  | x = 0 then A
end
```

x	1	2	0	1	2	1	1	1	1
y	1	3	3						
t	N	N	$B^R$						
y				-1	-3				
t				N	A				
y	1	3	3			4	5	6	7
t	N	N	$B^R$			N	N	N	N

```
let auto i fs ft env =
  let rec f (i, reset, env, hist, trim) =
    let s := if reset then fs i env else trim i (fs i (hist i env)) in
    let t := if reset then ft i s else trim i (ft i (hist i env)) in
    (* update continuations *)
    let hist j env' = if j = i then hist i (merge_until t s env') else hist j env' in
    let trim j env' = if j = i then rem_until t (trim i env') else trim j env' in
    merge_until t s (fun (j, reset) → f (j, reset, rem_until t env, hist, trim)))
  in f (i, false, env, (fun i env → env), (fun i s → s)).
```

# Transitions mixtes

## Sémantique

```
automaton initially A
state A do
  y = (0 fby y) + x;
  until x = 0 then B
state B do
  y = (0 fby y) - x;
  until y < -2 continue A
  | x = 0 then A
end
```

x	1	2	0	1	2	1	1	1	1
y	1	3	3						
t	N	N	$B^R$						
y				-1	-3				
t				N	A				
y	1	3	3			4	5	6	7
t	N	N	$B^R$			N	N	N	N
y	1	3	3	-1	-3	4	5	6	7

```
let auto i fs ft env =
  let rec f (i, reset, env, hist, trim) =
    let s := if reset then fs i env else trim i (fs i (hist i env)) in
    let t := if reset then ft i s else trim i (ft i (hist i env)) in
    (* update continuations *)
    let hist j env' = if j = i then hist i (merge_until t s env') else hist j env' in
    let trim j env' = if j = i then rem_until t (trim i env') else trim j env' in
    mergei_until t s (fun (j, reset) → f (j, reset, rem_until t env, hist, trim)))
  in f (i, false, env, (fun i env → env), (fun i s → s)).
```

# Conclusion

Travail accompli

# Conclusion

## Travail accompli

Interprétation des machines à états dans le modèle de Kahn

# Conclusion

## Travail accompli

Interprétation des machines à états dans le modèle de Kahn  
Fonctionnalités de Scade/Lustre : transitions fortes, faibles, réinitialisantes ou non, variables partagées

# Conclusion

## Travail accompli

Interprétation des machines à états dans le modèle de Kahn  
Fonctionnalités de Scade/Lustre : transitions fortes, faibles, réinitialisantes ou non, variables partagées

## Travail à accomplir

Traitement des horloges et de la synchronisation

# Conclusion

## Travail accompli

Interprétation des machines à états dans le modèle de Kahn  
Fonctionnalités de Scade/Lustre : transitions fortes, faibles, réinitialisantes ou non, variables partagées

## Travail à accomplir

Traitement des horloges et de la synchronisation  
Preuve d'adéquation avec le modèle relationnel

# Conclusion

## Travail accompli

Interprétation des machines à états dans le modèle de Kahn  
Fonctionnalités de Scade/Lustre : transitions fortes, faibles, réinitialisantes ou non, variables partagées

## Travail à accomplir

Traitement des horloges et de la synchronisation  
Preuve d'adéquation avec le modèle relationnel

## Mais avant ça

Définitions plus simples ? (Plus efficaces ?)  
Quelles règles de raisonnement ?