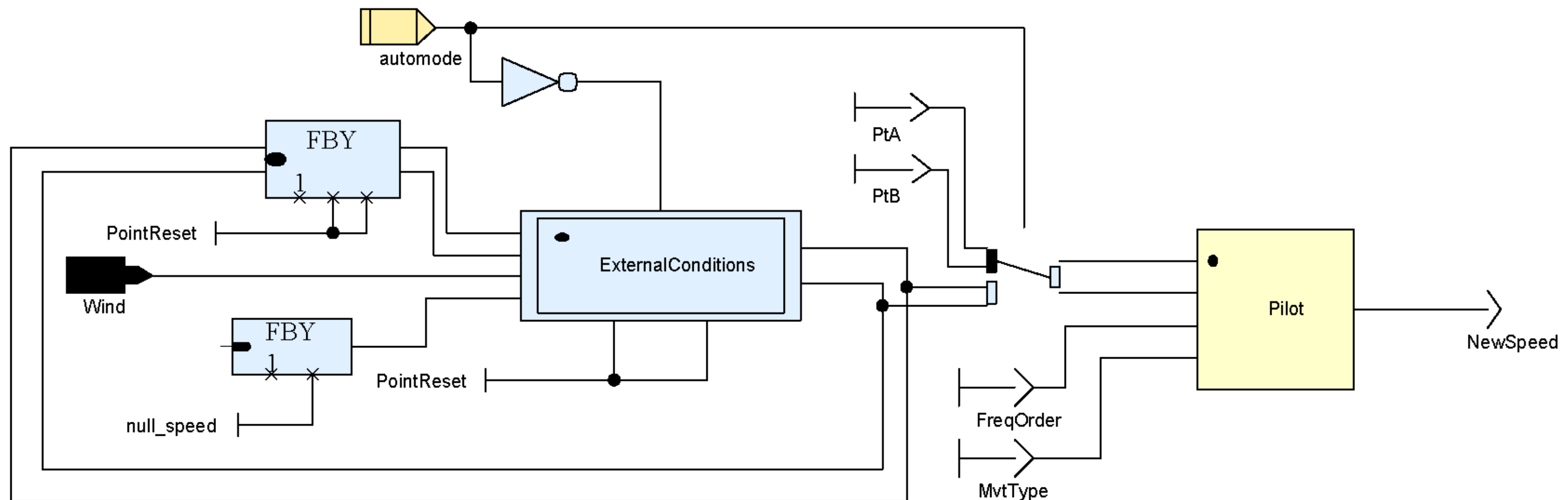
Uniform and Modular Composition of Data-flow & Control-flow in the Lazy λ-Calculus

joint work with Marc Pouzet

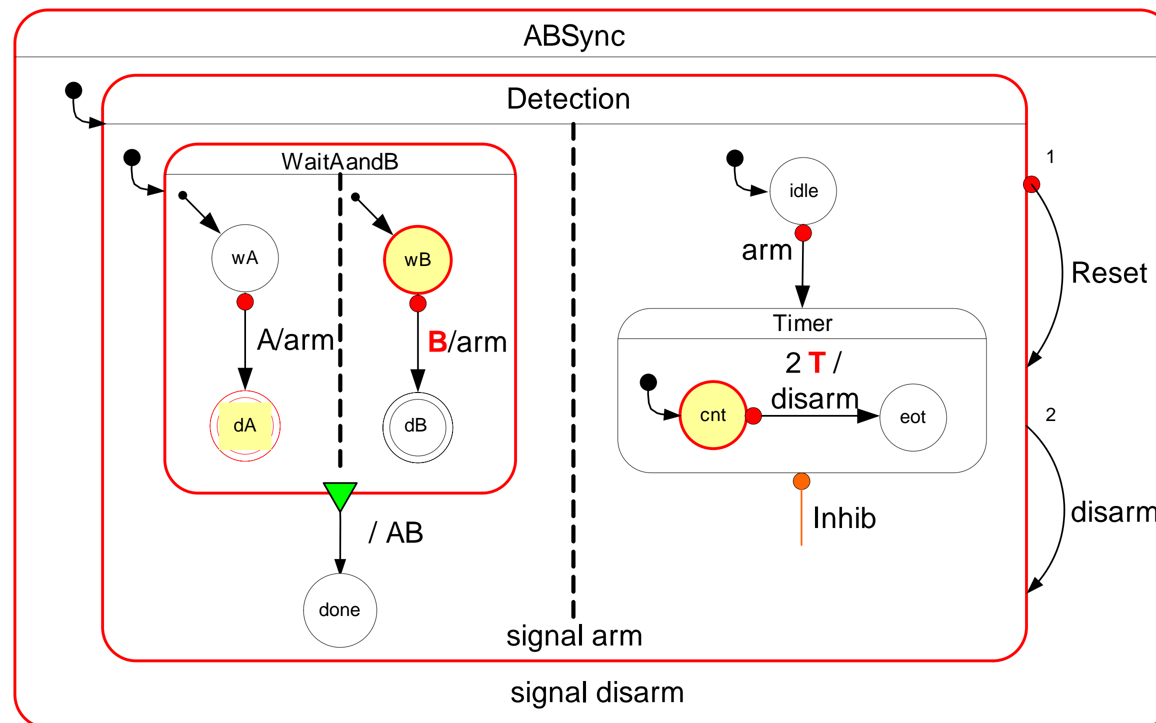
Introduction

Data Flow Programming



Signal, Lustre, SCADE V4, LabView, Simulink, ...

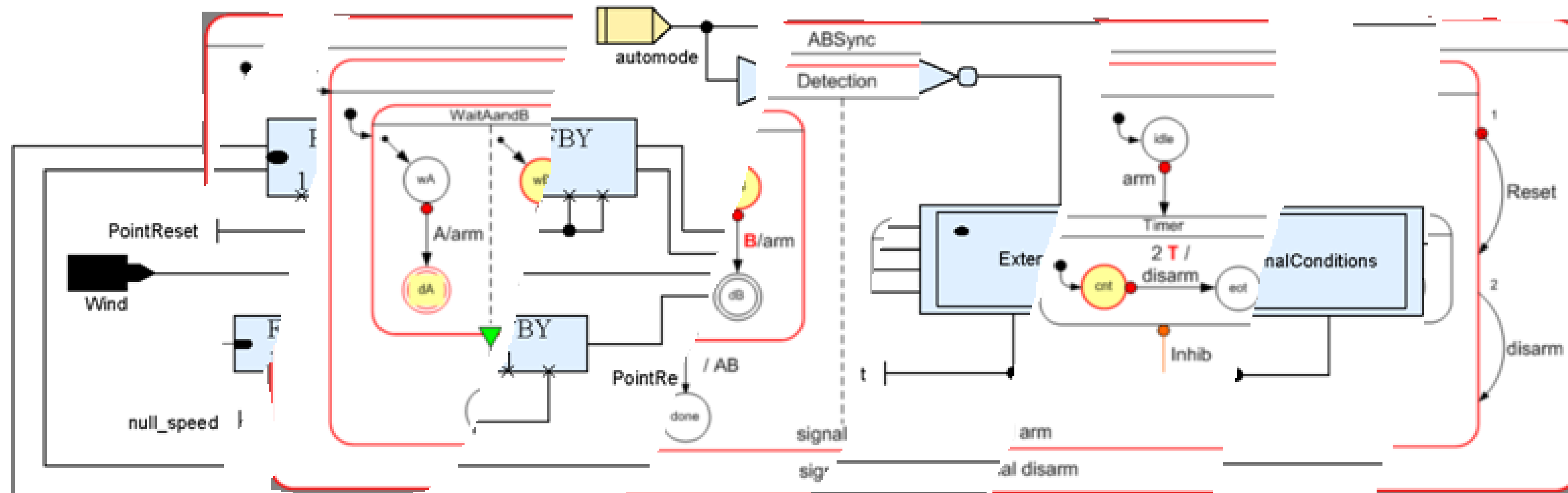
Control Flow Programming



Esterel, SyncCharts, Argos, Stateflow, Statemate, ...

Starting the 3rd Millennium ...

- ... we are looking at hybrid engines with tight integration of data flow and control flow:



ReLuC, Lucide Synchrone, SCADE 6, 7, 8, Synoptic, ...

Starting the 3rd Millennium ...

- ... we need to understand the semantical mechanics of such a hybrid (4 education, certification, verification,...)

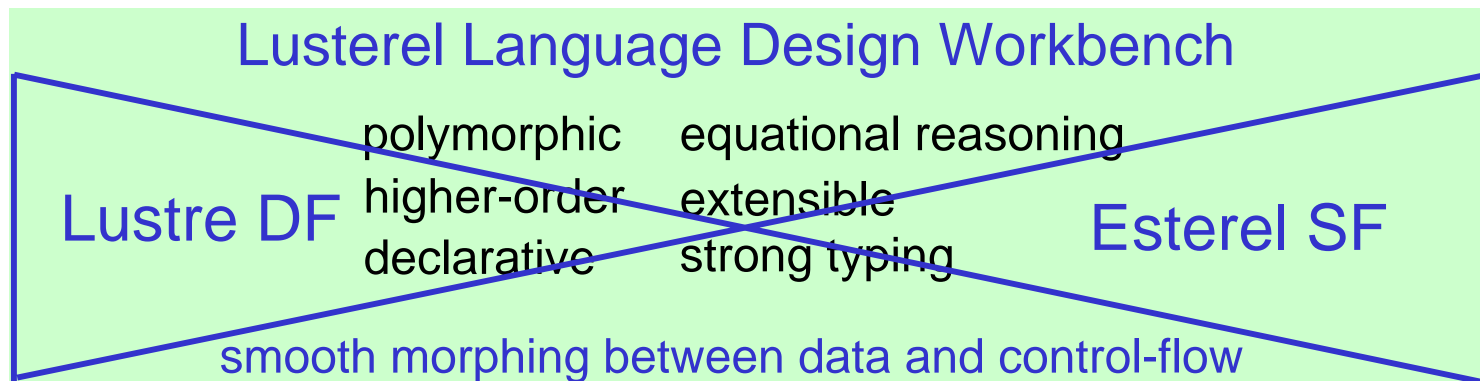
- Recent work (2005): J.-L. Colaço, B. Pagano, M. Pouzet give semantics of Scade V6 by
  - syntactic compilation of
  - SF → DF using *states-as-activation-clocks* with *explicit absence* values

- Open Question: Can SF and DF be unified into a
  - compositional semantic model
  - on equal footing, without „absence“ values ?

The Lusteral Experiment

We aim to integrate

- Lustre-style DF + Esterel-style SF under the same (co-recursive, functional) reactive process model
- shallow embedding in Haskell, aka lazy λ-calculus
- separation control $\neq$ data (no absence values)
- simple set of well-understood functional combinators



Kahn Principle

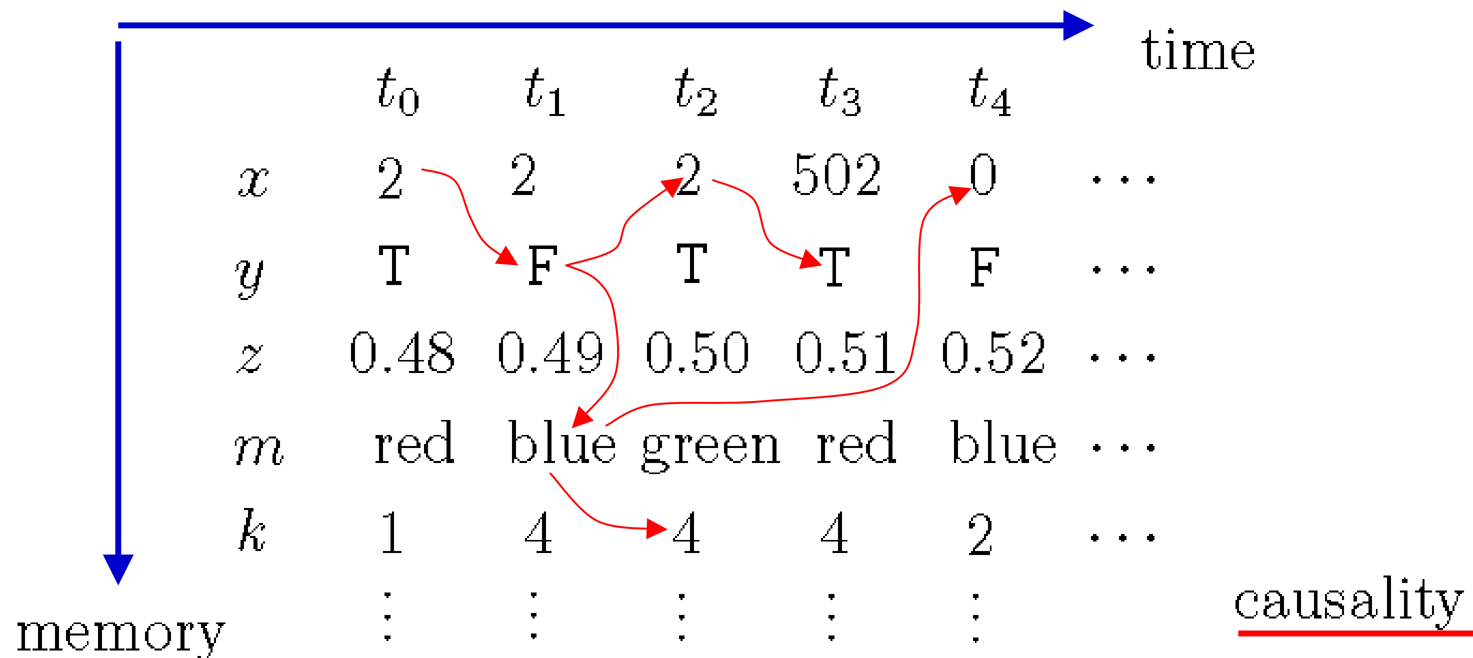
- At any time, a computing station is either computing or waiting for information on one of its input lines
- Each computation station follows a sequential program

Working Hypothesis:

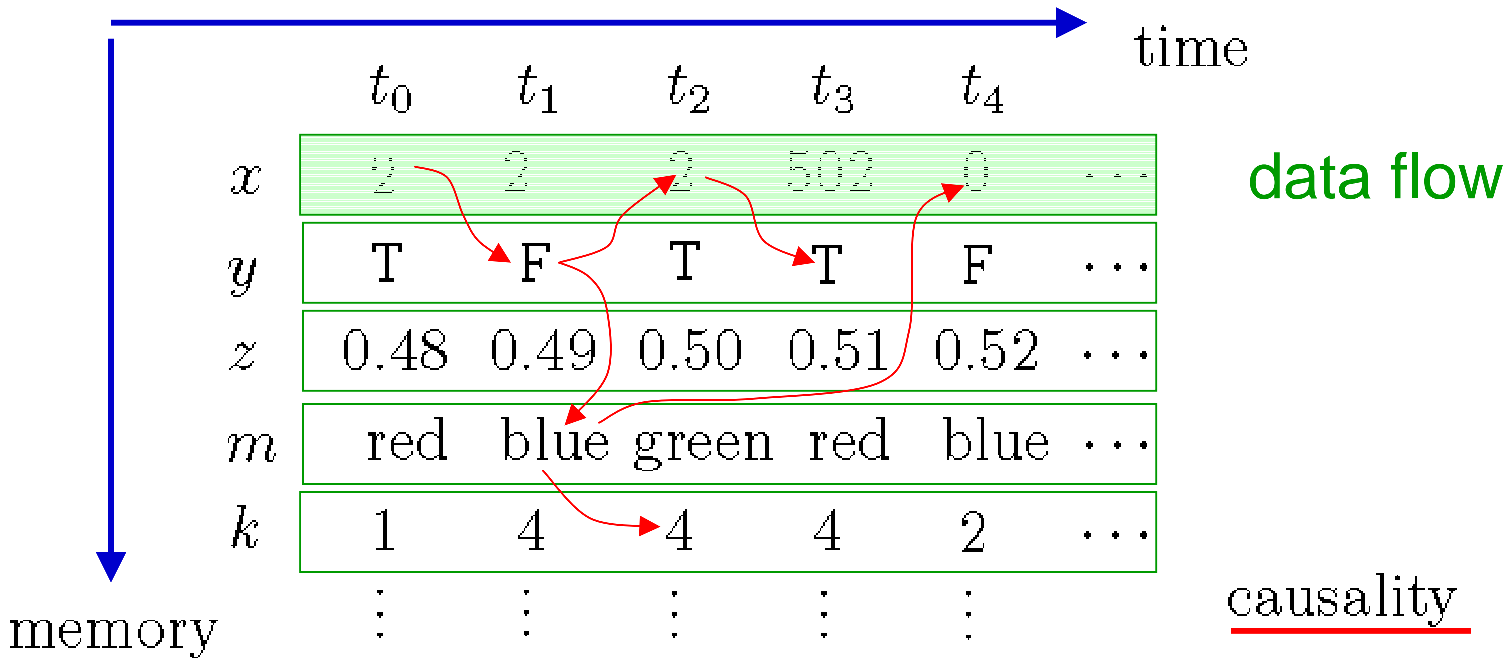
- Stream processing functions of Haskell are Kahn processes: Lazy rewriting is deterministic and sequential !
- Kahn processes can be coded in Haskell. Kahn processes exhibit parallelism of expression not of execution!

The Role of Laziness

Orthogonality in Time and Space

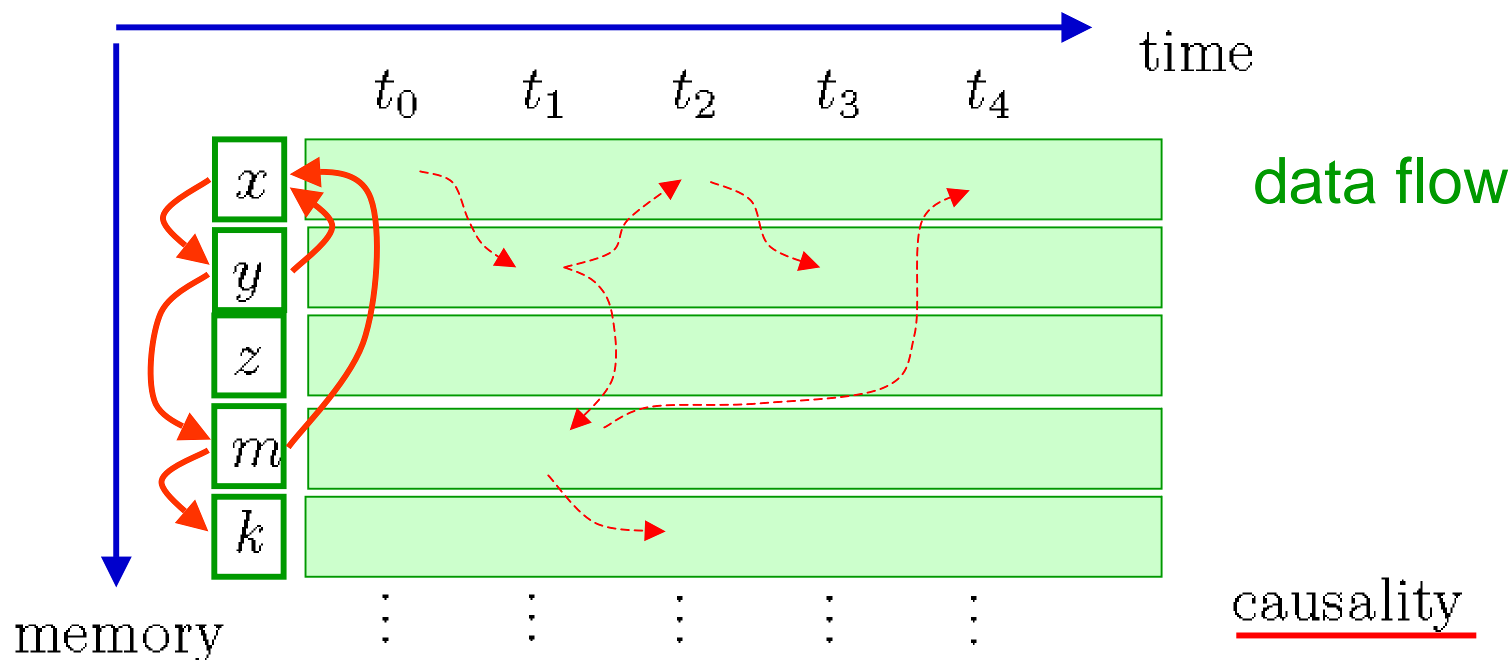


Data Flow



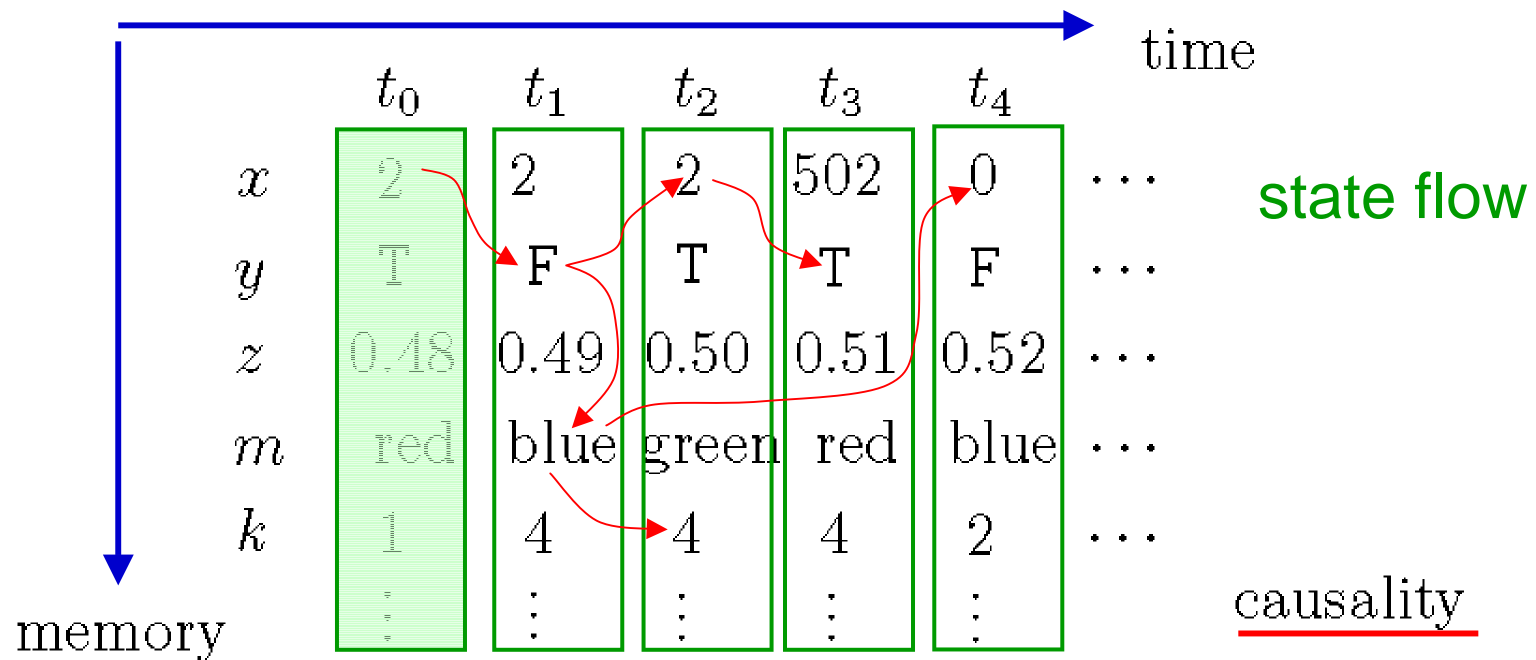
Data Flow

Q: How do we treat the cyclic DF dependencies ?



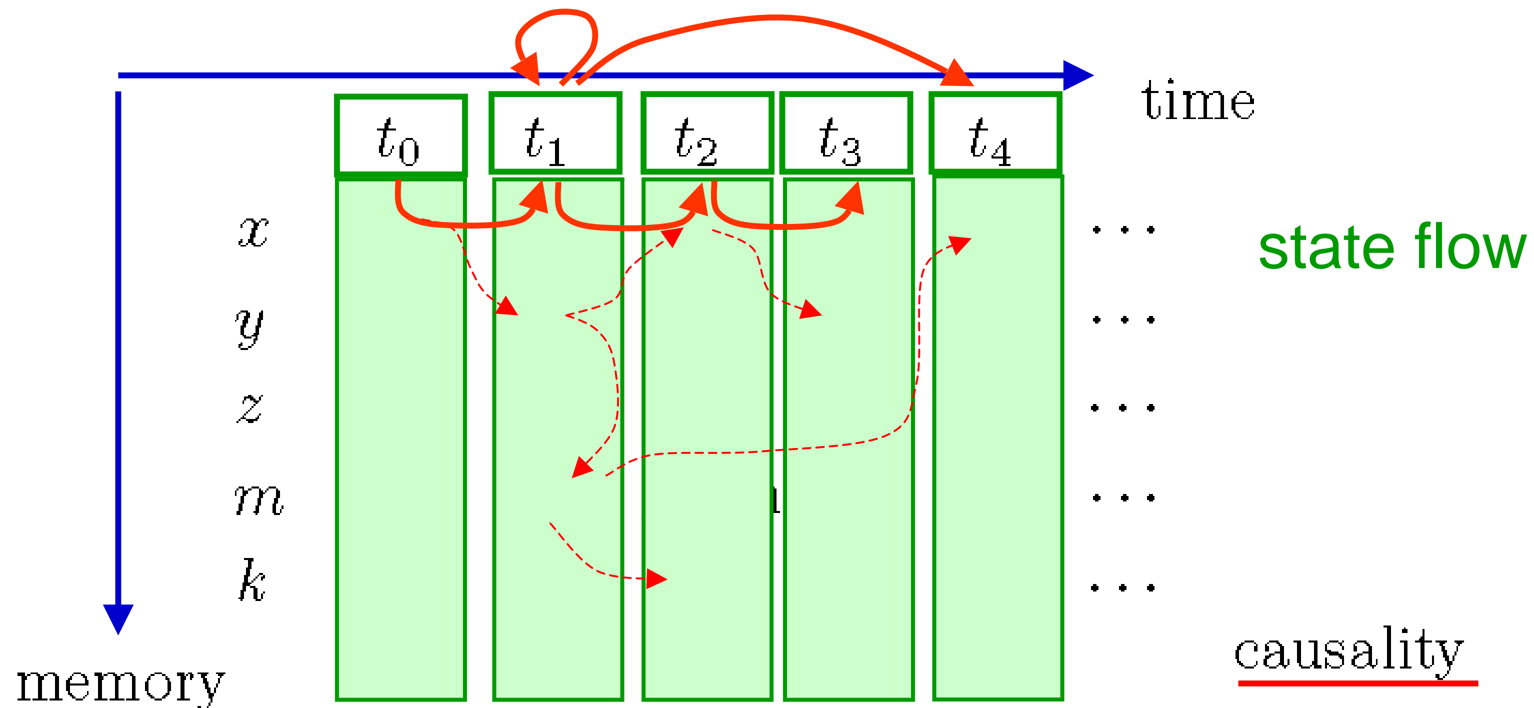
A: Fixpoints on prefix-ordering, lazy recursion on streams!

State Flow



State Flow

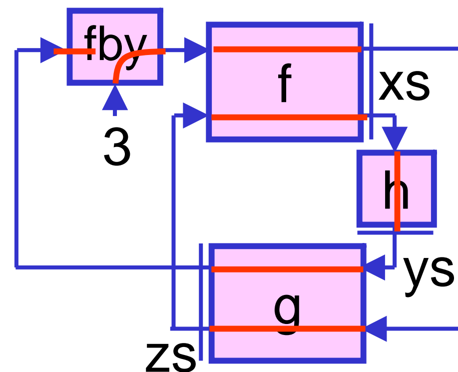
Q: How do we treat the cyclic SF dependencies ?

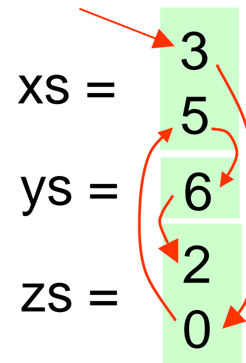


A: Scott information-ordering, lazy recursion on state

Fixed-point on Partial States

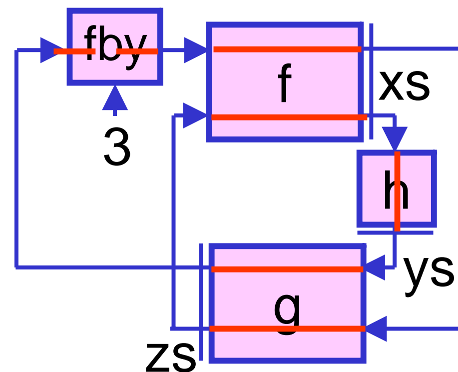
xs = f (3 fby (fst zs), snd zs)
ys = h (snd xs)
zs = g (ys, fst xs)

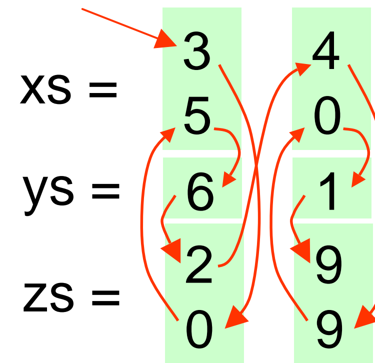




Fixed-point on Partial States

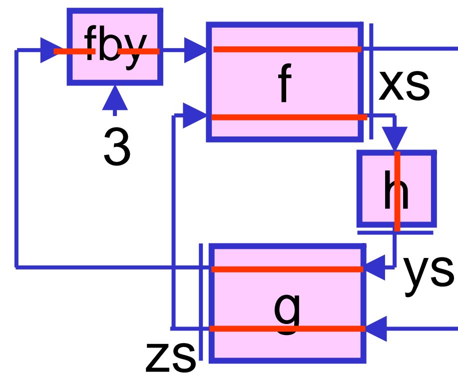
xs = f (3 fby (fst zs), snd zs)
ys = h (snd xs)
zs = g (ys, fst xs)

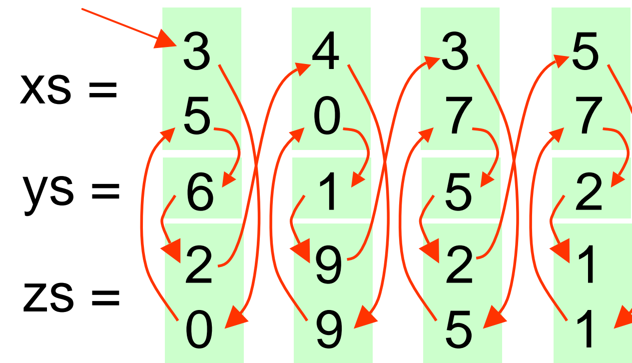




Fixed-point on Partial States

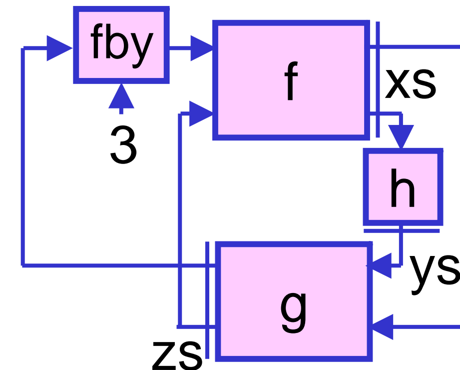
xs = f (3 fby (fst zs), snd zs)
ys = h (snd xs)
zs = g (ys, fst xs)





Where Eagerness is not Productive ...

xs = f (3 fby (fst zs), snd zs)
ys = h (snd xs)
zs = g (ys, fst xs)

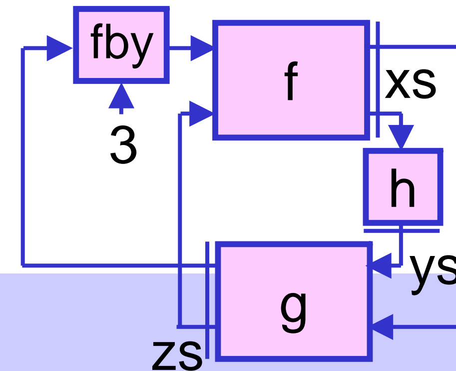


Eager Evaluation

fst xs → fst (f(3 fby (fst zs), snd zs))
→ fst (f(3:(fst zs), snd zs))
→ fst (f(3:(fst zs), snd(g(ys,fst xs))))
→ fst (f(3:(fst zs), snd(g(h(snd **xs**),fst xs))
→ fst (f(3:fst zs), snd(g(h(snd(f(3 fby (fst zs), snd zs))),fst xs)
→ … loop !

... Lazy Function Evaluation may be.

xs = f (3 fby (fst zs), snd zs)
ys = h (snd xs)
zs = g (ys, fst xs)



Lazy Evaluation

fst xs → fst(f(3 fby (fst zs), snd zs))

→ fst(($f_1 \times f_2$)(3 fby (fst zs), snd zs))

→ fst((λw.(f_1(fst w), f_2(snd w))(3 fby (fst zs), snd zs))

→ fst(f_1(fst(3 fby (fst zs), snd zs)), f_2(snd(3 fby (fst zs), snd zs)))

→ f_1(fst(3 fby (fst zs), snd zs))

lazy on state (data)

→ f_1(3 fby (fst zs))

lazy on streams

→ f_1(3:(fst zs))

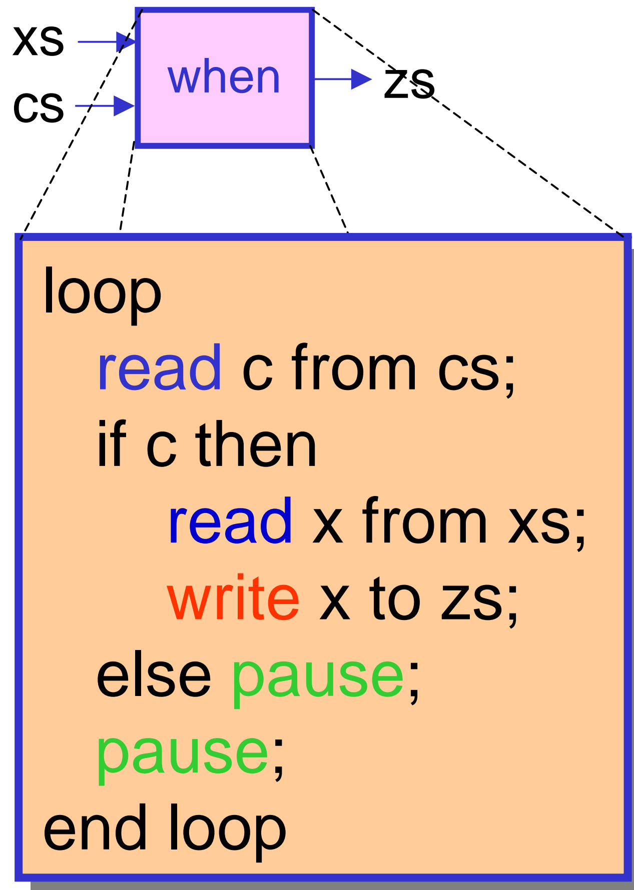
→ f_1(3) : f_1(fst zs)

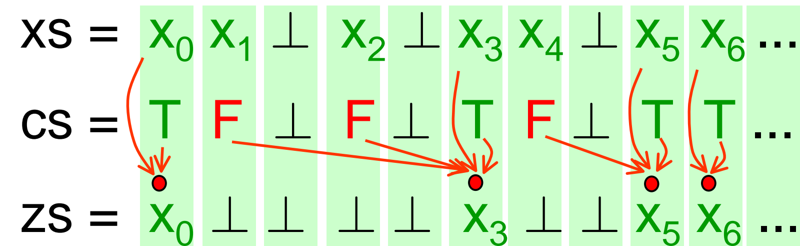
where H[v] = $f_1(g_1(h(f_2(g_2(v)))))$

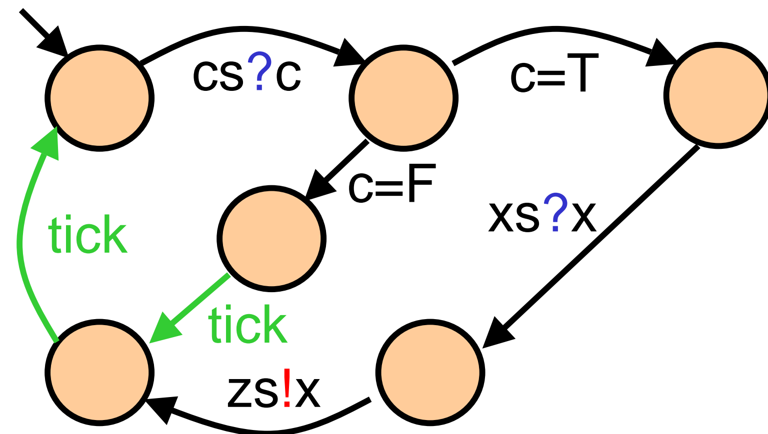
→ ... f_1(3) : H(f_1(3)) : H(H(f_1(3))) : H(H(H(f_1(3)))) : ...

SF $\neq$ Clocked DF

Clock Schedules are Implicit

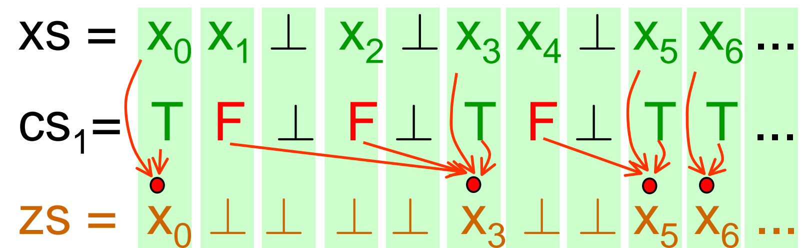


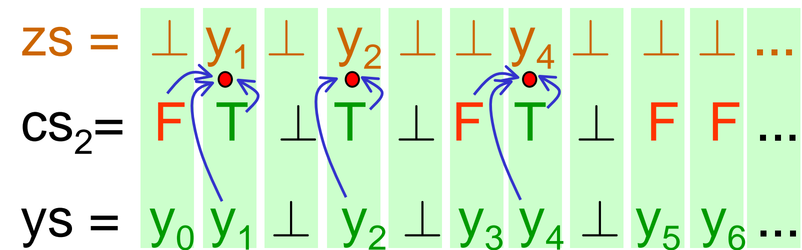




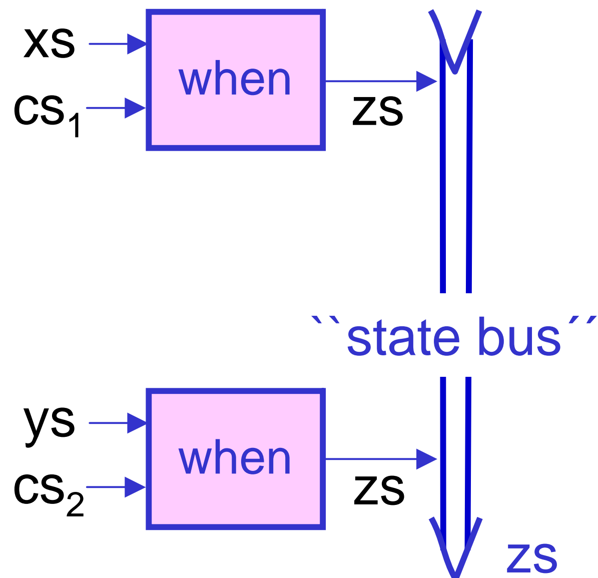
absence ⊥ is not communicated !

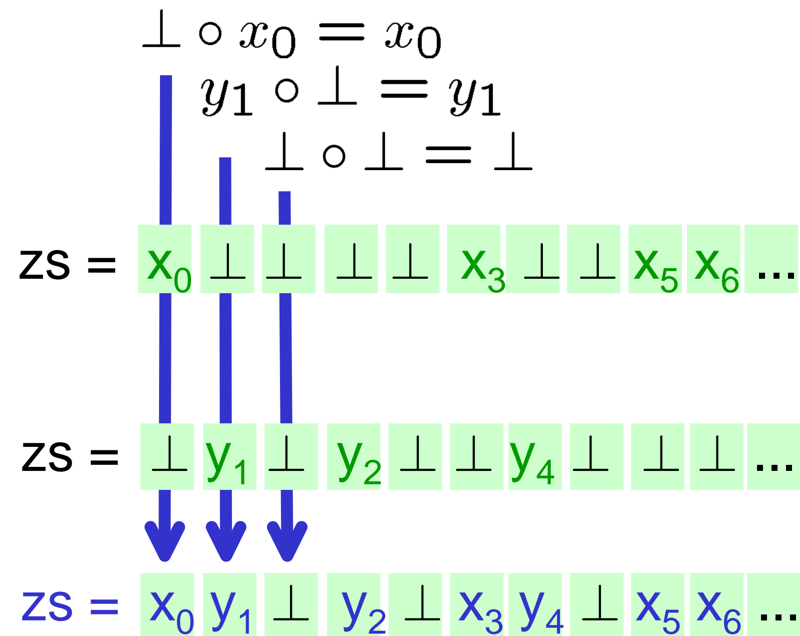
Absence = Identity Reaction on 'State Bus'





Absence = Identity Reaction on 'State Bus'



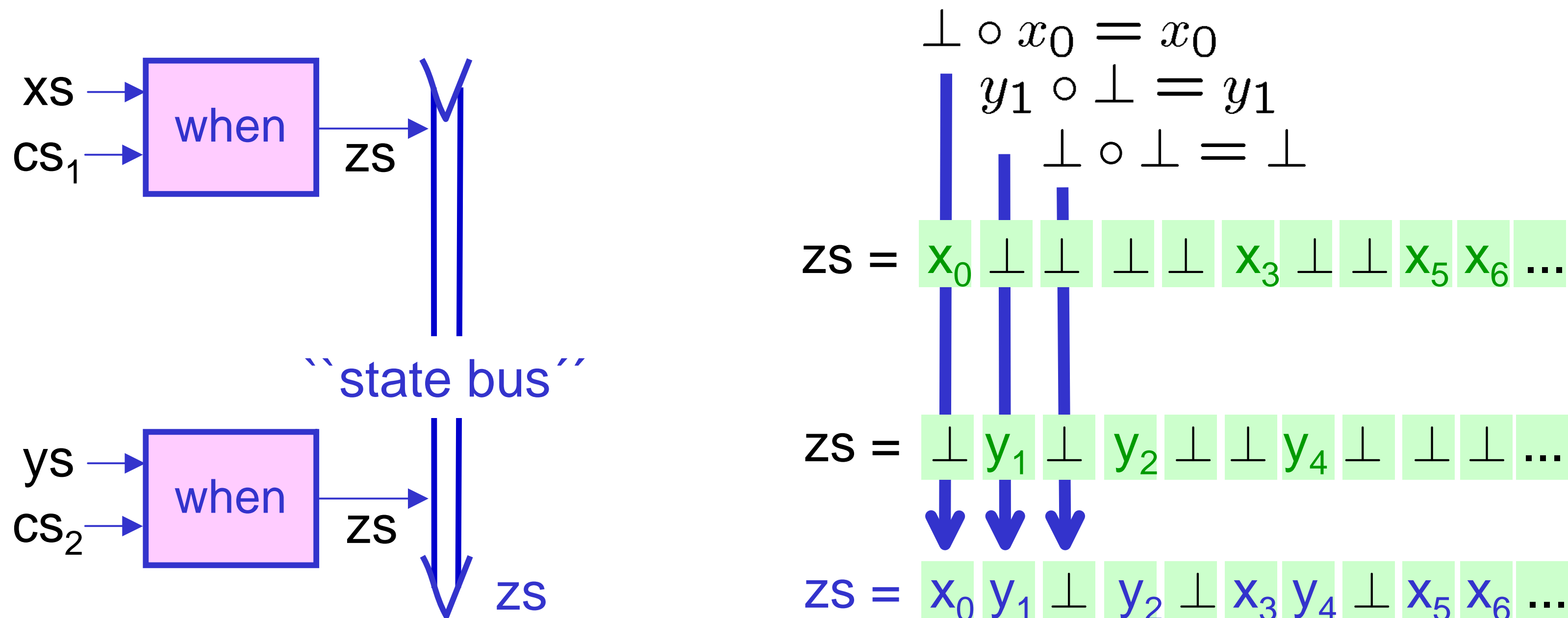


state bus values = response functions: $v \cong \lambda x.\, v \quad \bot \cong \lambda x.\, x$

parallel emits = function composition: $v_1 \mid\!>\!\mid v_2 \cong v_2 \circ v_1$

The last value dominates the bus $v_2 \circ v_1 = v_2$ if $v_2 \neq \bot$

Absence = Identity Reaction on ‘State Bus‘



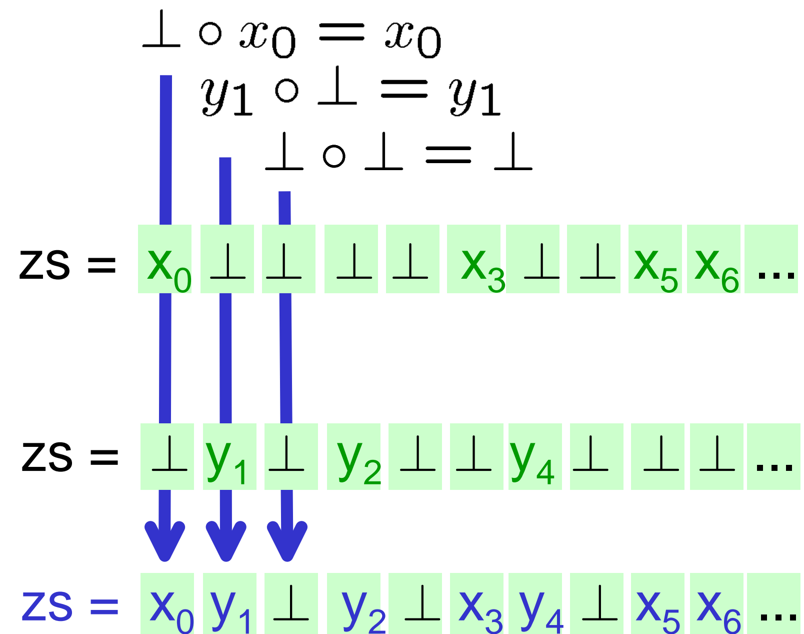
$x \circ y$ same as x default y (Signal) but deeply coded !

non-termination Ω DF $\neq$ identity $\perp$ SF $\neq$ absence $*$ DF

Absence = Identity Reaction on 'State Bus'

M. Fourman [1989]:

Response functions (cpo) to model bi-directional instantaneous communication.



x ○ y same as x default y (Signal) but deeply coded !

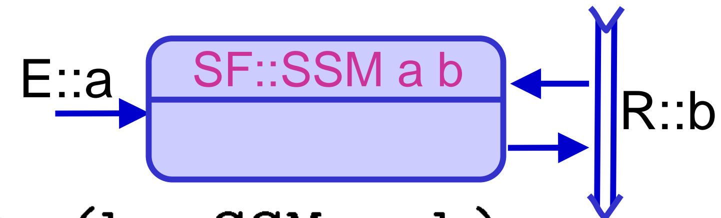
non-termination Ω ≠ identity $\bot$ ≠ absence $*$
DF SF DF

Synchronous Processes

SF/DF Reaction Relations

State Flow

$$E; R \vdash \mathsf{SF} \xrightarrow{R'} \mathsf{SF}'$$



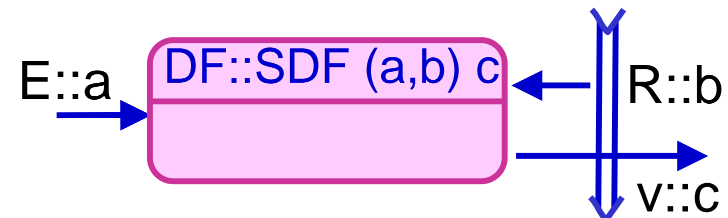

```
data SSM a b = (a,b) -> (b, SSM a b)
```

$R = \{\mathtt{sv}_1 = v_1, \mathtt{sv}_2 = v_2, \ldots, \mathtt{sv}_n = v_n\} :: b$ signal context

$E = \{x_1 = u_1, x_2 = u_2, \ldots, x_m = u_m\} :: a$ df variables

Data Flow

$$E; R \vdash \mathsf{DF} \xrightarrow{v} \mathsf{DF}'$$




```
data SDF (a,b) c = (a,b) -> (c, SDF (a,b) c)
```

SF/DF Process Language

State Flow

SF ::= s	state flow variable
| sv ? x ; SF	signal input
| sv ! DF ; SF	signal emission
| pause ; SF	wait
| present DF then SF else SF	branching
| do s = SF until DF then SF	weak preemption
| SF |>| SF	parallel
| sv := DF	binding DF to signal
| local sv in SF	local signal
| loop s. SF	iteration

sv $\in dom(R)$ state signal variable

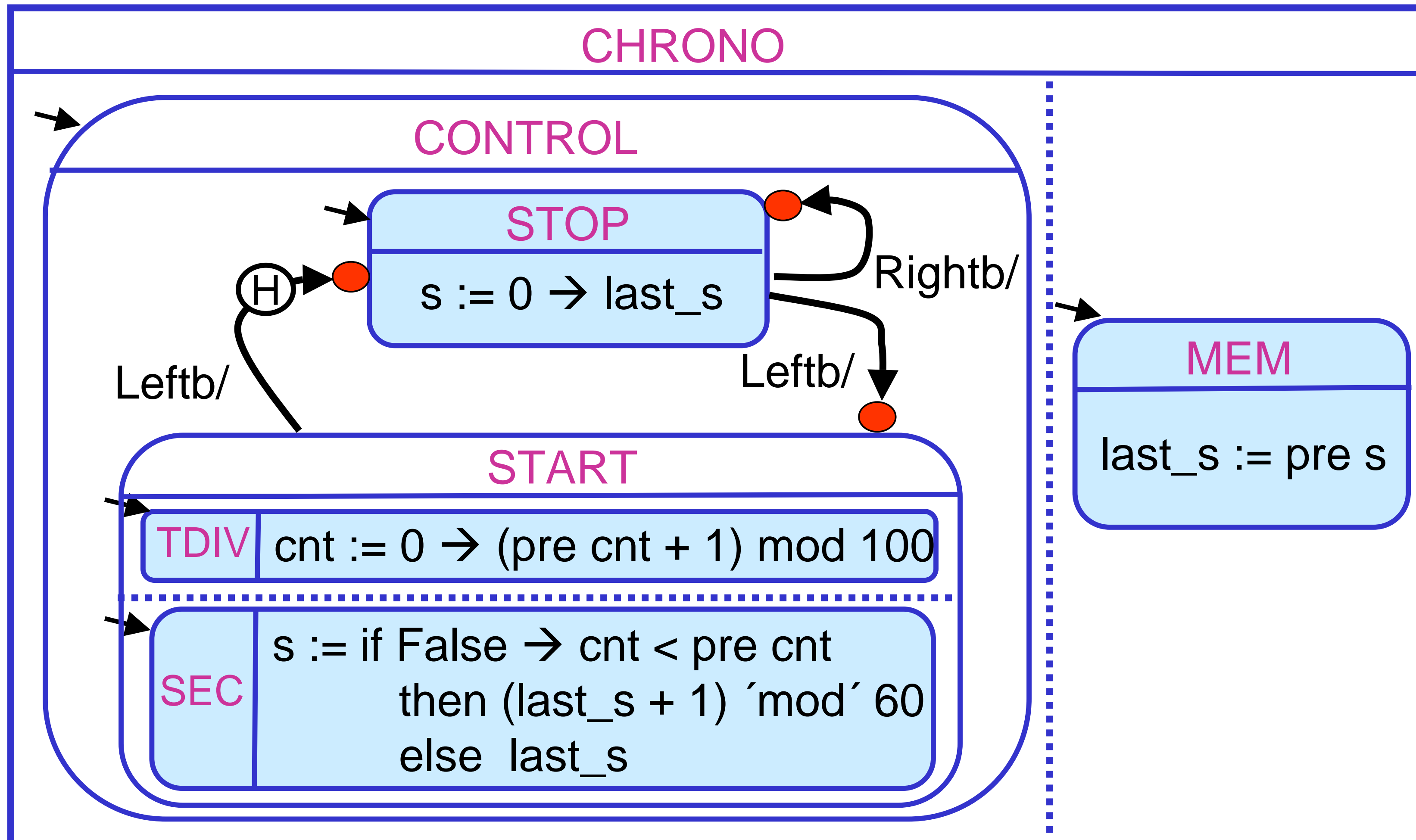
Data Flow

$\mathsf{DF} ::= \mathtt{rv}$	data flow variable
$\mid\ \mathtt{current}_v\ \mathtt{sv}\ \mathtt{in}\ \mathsf{SF}$	close-off state flow
$\mid\ op\ \mathsf{DF}\ \mathsf{DF} \cdots \mathsf{DF}$	static lifting of value operators
$\mid\ \mathtt{merge}\ \mathsf{DF}\ \mathsf{DF}\ \mathsf{DF}$	up-sampling
$\mid\ \mathsf{DF}\ \mathtt{when}\ \mathsf{DF}$	down-sampling
$\mid\ \mathtt{pre}\ \mathsf{DF}$	delay
$\mid\ \mathsf{DF} \rightarrow \mathsf{DF}$	initialisation
$\mid\ \mathsf{DF}\ \mathtt{fby}\ \mathsf{DF}$	initialised delay
$\mid\ \mathtt{rec}\ \mathtt{rv}.\ \mathsf{DF}$	feed-back, recursive data flow

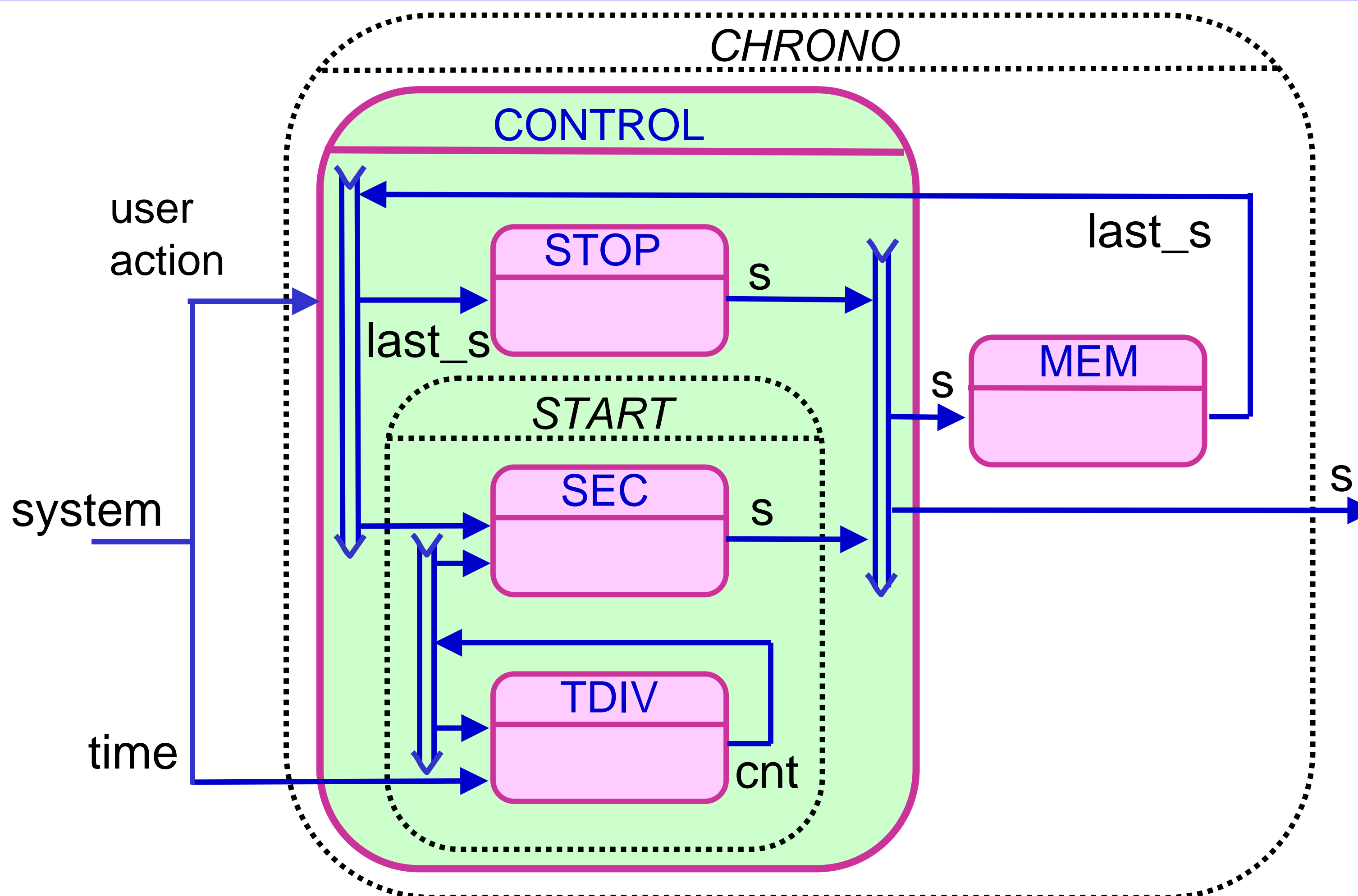
$\mathtt{rv} \in dom(E) \cup dom(R)$ interface read variables

Mode Automaton Example

Macro States & Mode Automaton



Data Flow

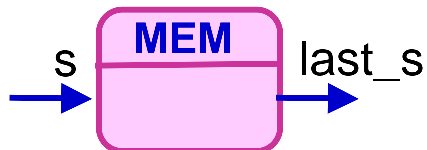


Reaction interface „E;R“

$$\texttt{mem} :: \mathsf{SSM}\{E::\texttt{Sys}\}\{\texttt{last_s}::\texttt{Int}, \texttt{s}::\texttt{Int}\}$$

MEM from Data Flow

```
mem =
  last_s := pre s
```



MEM as State Flow

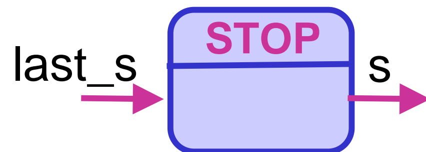
```
mem =
  loop end.
     s?v ;
     pause ;
     last_s!v ;
  end
```

DF Interface

$$\texttt{stop} :: \mathsf{SDF}\{E{::}\texttt{Sys}\}\{\texttt{last_s}{::}\texttt{Int}\} \rightarrow \mathsf{SDF}\{E{::}\texttt{Sys}\}\{\texttt{s}{::}\texttt{Int}\}$$

STOP as Data Flow

$$\texttt{stop}\,\texttt{last_s} = 0 \rightarrow \texttt{last_s}$$

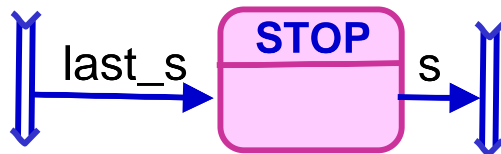


SF Reaction interface „E;R“

$$\texttt{stop} :: \mathsf{SSM}\{E{::}\texttt{Sys}\}\{\texttt{last_s}{::}\texttt{Int}, \texttt{s}{::}\texttt{Int}\}$$

STOP from Data Flow

$$\texttt{stop} =$$
$$\texttt{s} := 0 \rightarrow \texttt{last_s}$$



STOP as State Flow

$\texttt{loop}$ $\texttt{end.}$

$s!0$;

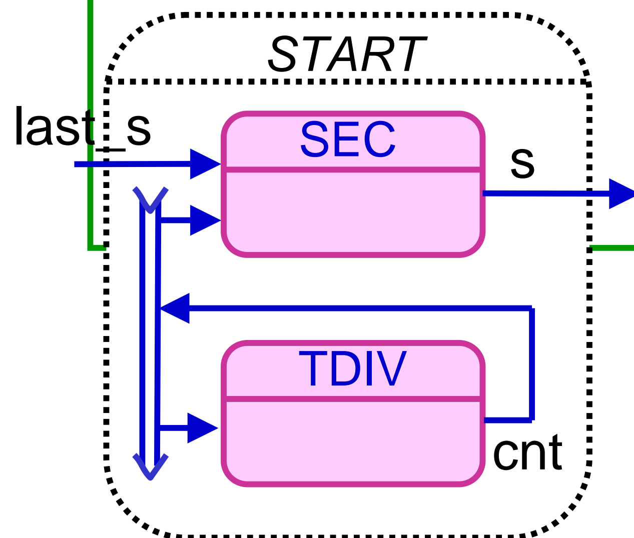
$\texttt{pause}$;

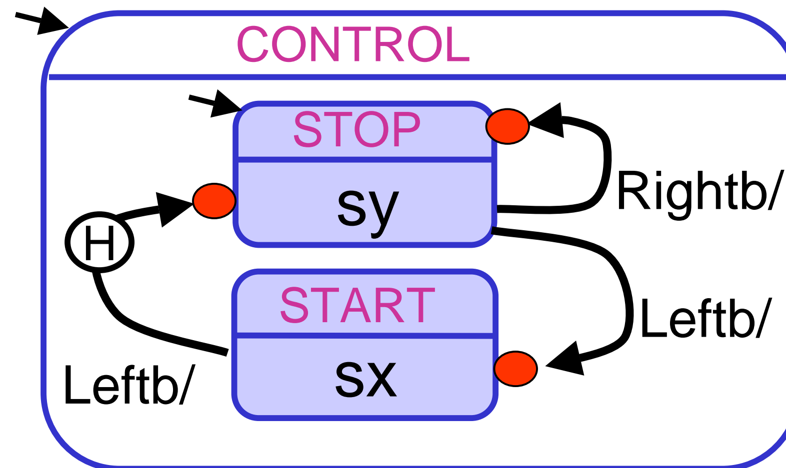
$\texttt{last_s}?v$;

$\texttt{s}!v$;

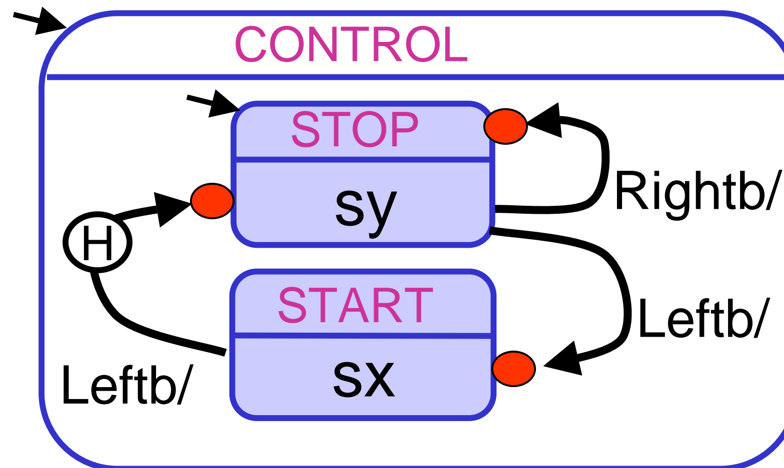
$\texttt{end}$

$$
\begin{aligned}
\texttt{start} &= \texttt{local}\,\texttt{cnt}\,\texttt{in}\,\texttt{sec}\ |{>}|\ \texttt{tdiv}\\
&\quad \texttt{where}\ \texttt{tdiv} =\\
&\qquad\qquad \texttt{cnt} := 0 \rightarrow (\texttt{pre}\,\texttt{cnt} + 1)\,\texttt{mod}\,100\\
&\qquad\quad \texttt{sec} =\\
&\qquad\qquad \texttt{s} := \ \texttt{if}\,\texttt{False} \rightarrow \texttt{cnt} < \texttt{pre}\,\texttt{cnt}\\
&\qquad\qquad\qquad \texttt{then}\,(\texttt{last_s} + 1)\,\texttt{mod}\,60\\
&\qquad\qquad\qquad \texttt{else}\,\texttt{last_s}
\end{aligned}
$$

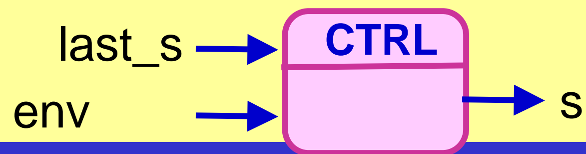


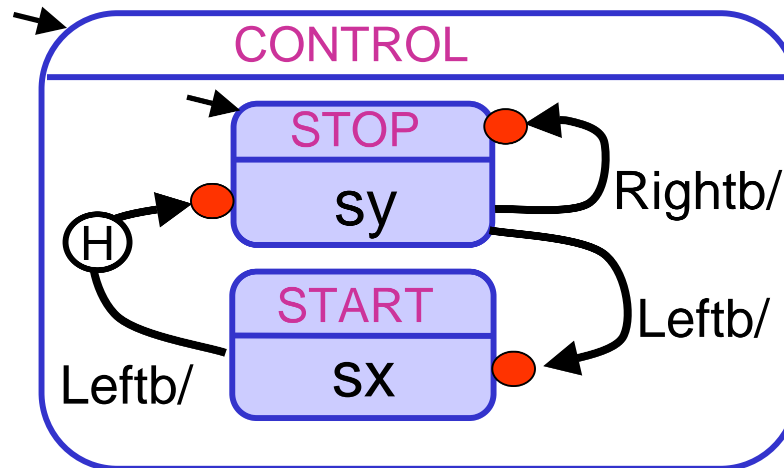



```
ctrl_stop sx sy =
   do p = sy until isBp(env)
   then present isLb(env)
            then ctrl_start start p
            else ctrl_stop sx stop
```




```
ctrl_stop sx sy = ...
ctrl_start sx sy =
    do p = sx until isLbp(env)
    then ctrl_stop p sy
```



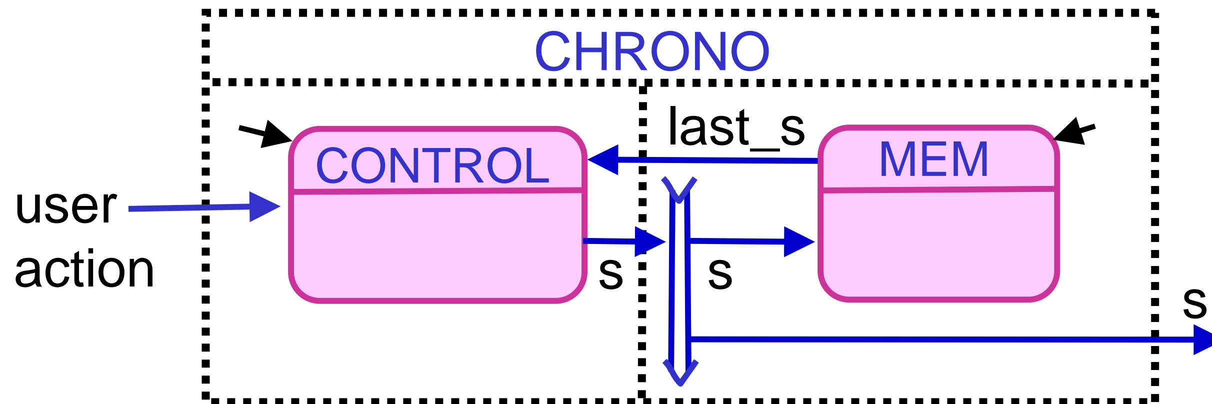


$\mathtt{ctrl_stop}\ \mathtt{sx}\ \mathtt{sy} = \ldots$

$\mathtt{ctrl_start}\ \mathtt{sx}\ \mathtt{sy} = \ldots$

CONTROL as Data Flow

$\mathtt{control} = \mathtt{ctrl_stop}\ \mathtt{start}\ \mathtt{stop}$

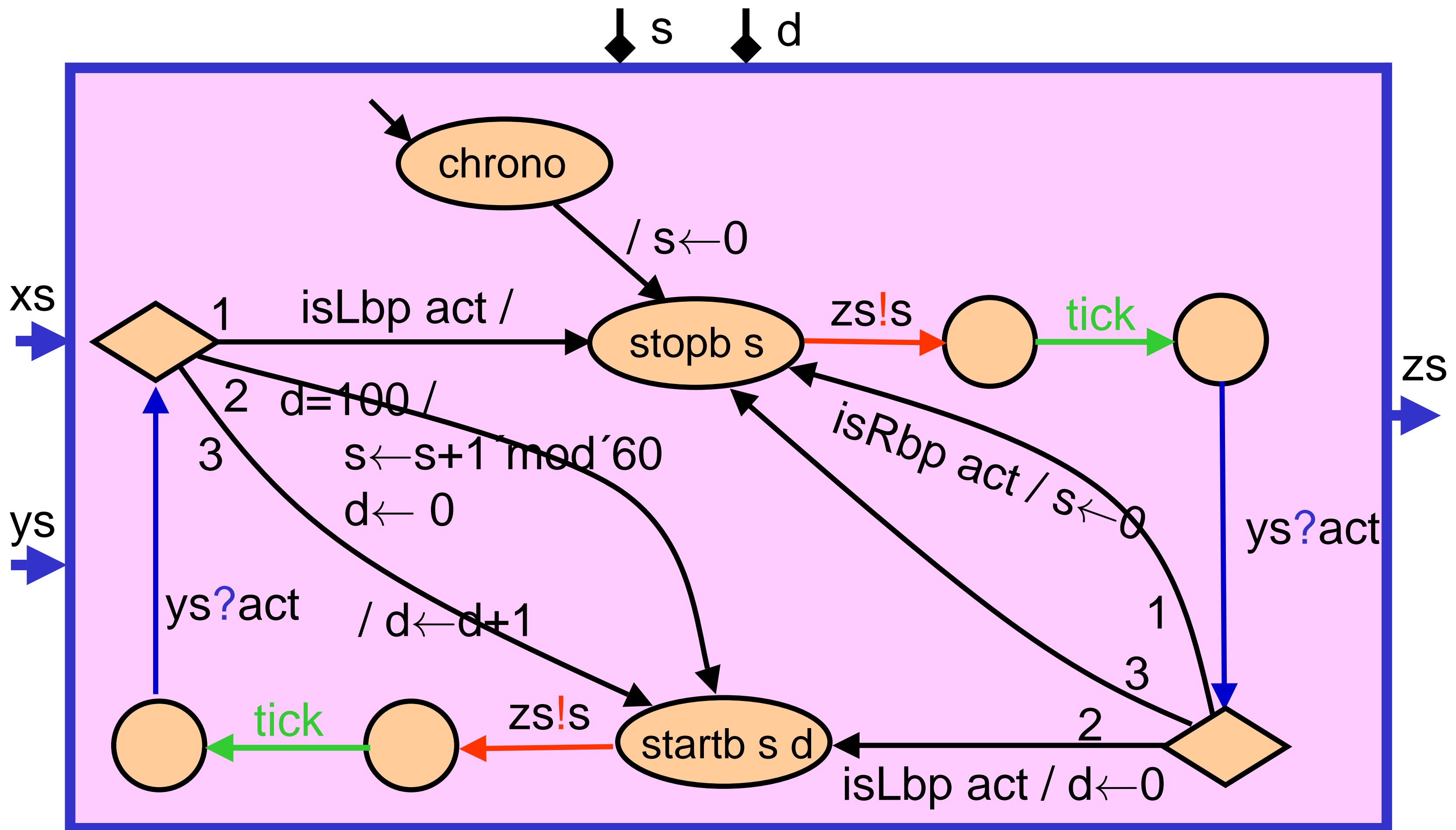


$\texttt{control}, \texttt{mem} :: \mathsf{SSM}\{E{::}\mathtt{Sys}\}\{\mathtt{last_s{::}Int}\ ,\ \mathtt{s{::}Int}\}$

$\texttt{chrono} :: \mathsf{SSM}\{E{::}\mathtt{Sys}\}\{\mathtt{s{::}Int}\}$

CHRONO as Data Flow

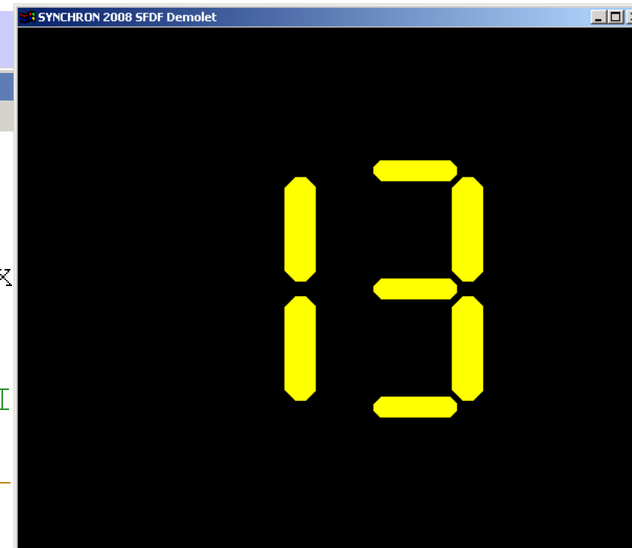
$\mathtt{chrono} = \mathtt{local\,last_s\,in\,(mem\ |{>}|\ control)}$


s
d
chrono
/ s←0
xs
1
isLbp act /
stopb s
zs!s
tick
zs
2 d=100 /
3
s←s+1 mod 60
d← 0
isRbp act / s←0
ys?act
ys
ys?act
/ d←d+1
1
3
tick
zs!s
startb s d
2
isLbp act / d←0

emacs@QUETZAL

File Edit Options Buffers Tools Haskell Help

```haskell
-- START:
start_x  :: SSM a (Mem Int)
start_x = hider (sec_x |>| (pweakrr tdiv_x

type CHRONO_STATE = SSM Environment (Mem I

pstopr_x :: CHRONO_STATE -> CHRONO_STATE -

pstopr_x startp_x stopp_x
   = pdountilm stopp_x dIsBp dispatch
       where dispatch Leftb  = \p -> pstartr_x start_init_x p
             dispatch Rightb = \_ -> pstopr_x startp_x stop_init_x


pstartr_x :: CHRONO_STATE -> CHRONO_STATE -> CHRONO_STATE

pstartr_x startp_x stopp_x
   = pdountil startp_x dIsLbp dispatch
       where dispatch = \p -> pstopr_x p stopp_x

-- CONTROL
control_x :: CHRONO_STATE
control_x = pstopr_x start_x stop_init_x
```



Haskell implementation

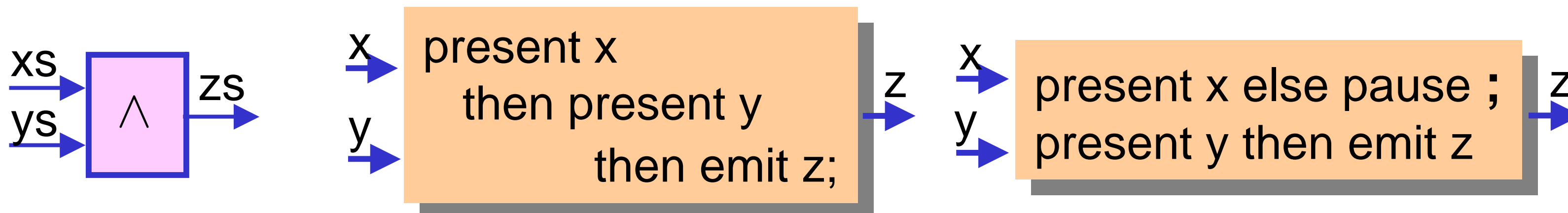
Start | C:\cygwin\home\... | Microsoft PowerP... | C:\WINNT\system... | C:\cygwin\home\... | emacs@QUETZAL | DE | 14:37

ESTEREL is non-sequential !

Concurrent AND, OR

Esterel Control Flow is concurrent (non-sequential) !

Concurrent AND



z is absent if at least one of x or y is absent

Concurrent OR

z is present if at least one of x or y is present

Conclusion

Summary

- Kahn‘s original model of SF-DF can be uniformely coded in lazy λ-calculus without clocks and explicit „absent“ values
- Kahn‘s prefix ordering is not sufficient for SF causality, need a form of Scott-ordering on state
- Kahn SF is weaker than (full) Esterel SF, the latter is non-sequential with true concurrency of execution
- Lusteral implements sequential „token ring“ broadcast of signals not concurrent „ethernet“ (like Esterel)

Related Work

- Nowak, Beauvais, Talpin [1999]: Signal in Coq
- Elliott [1997], Hudak [2000]: Functional reactive programming/animations (FRAN, FRP) in Haskell
- Kieburtz [1998]: Reactive Functional Programming
- Uustalu [2005]: DF with comonads and monads
- Pouzet, Hamón, Boulmé [$\approx$ 2001]: Lucid Synchrone in Coq
- Mandel [2004]: Reactive ML