Synchronous Languages 101

A Story[1]

Laure Gonnord and Lionel Morel

University of Lyon/ LIP - INSA Lyon / CITI

March 2016

[1] By 3rd generation degenerates

Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

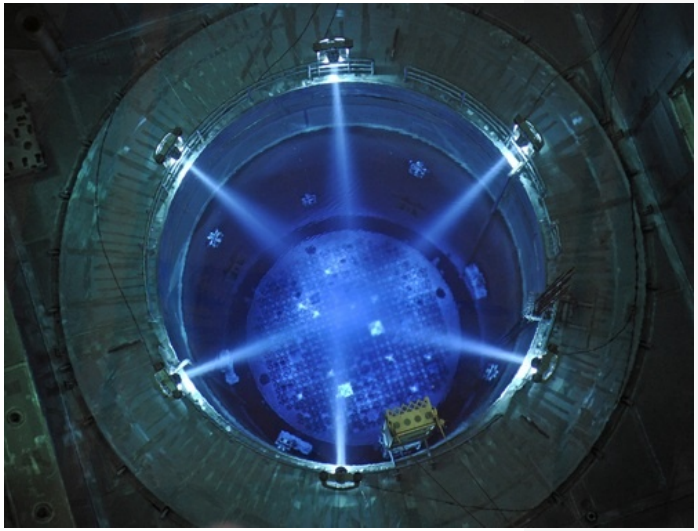
The LUSTRE language

Compilation of Synchronous Languages (Lustre)

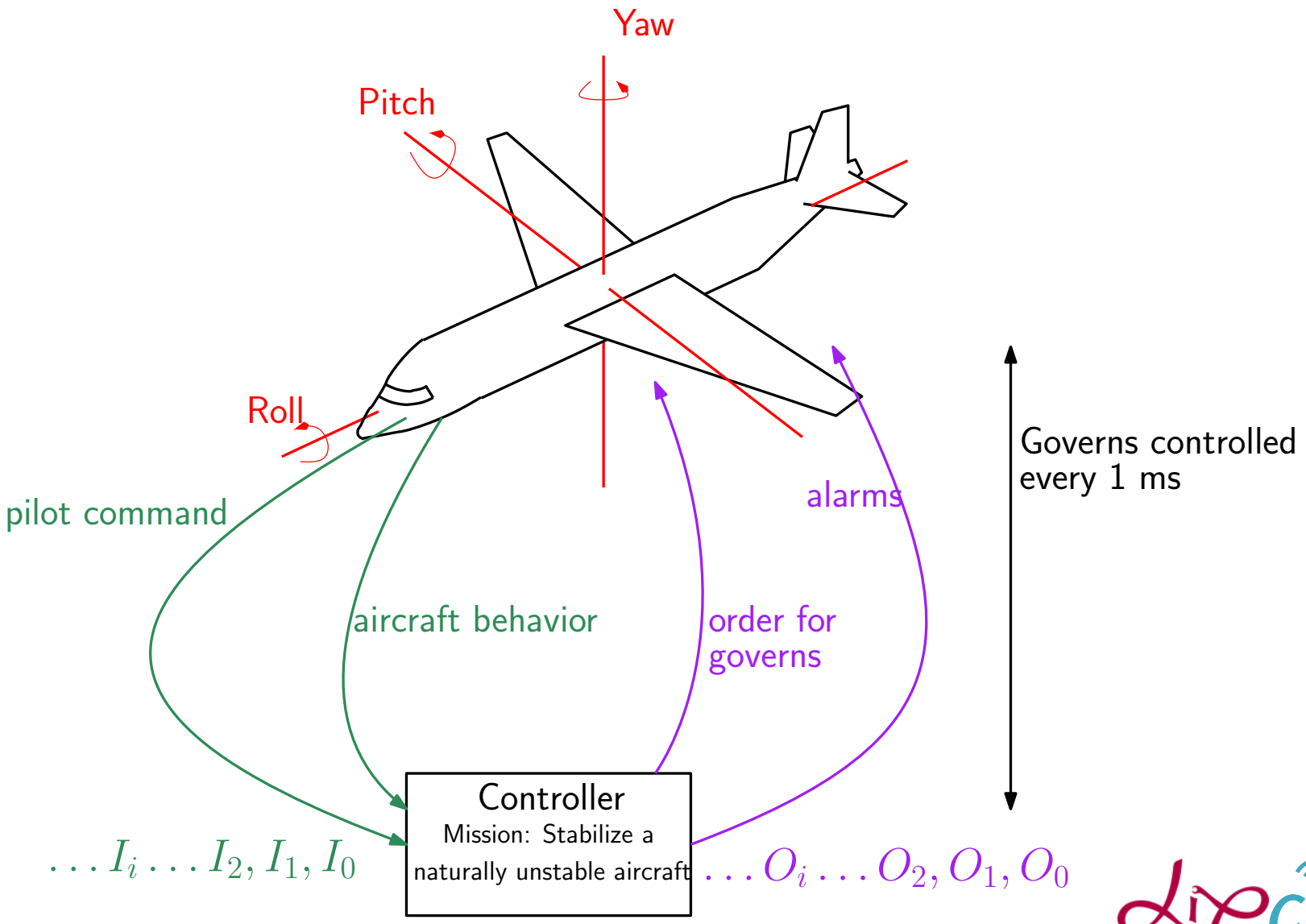
Conclusion

What about [non-synchronous] dataflow?

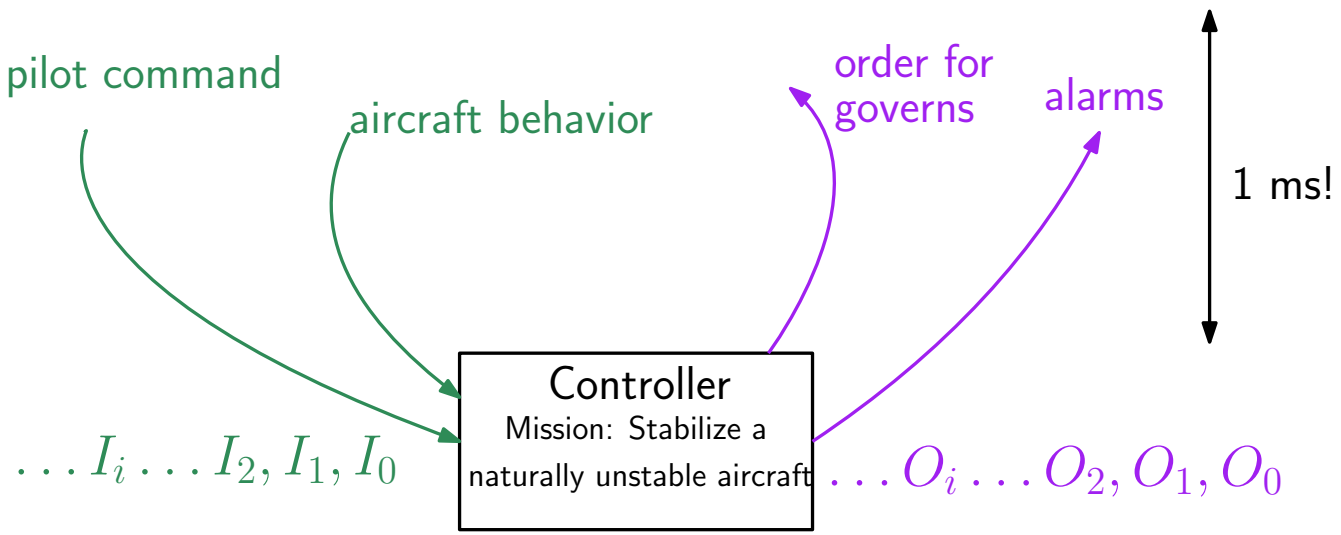
Safety-Critical Embedded Systems



Reactive System: Example



Programming



Functional correction

Compute the correct output values.

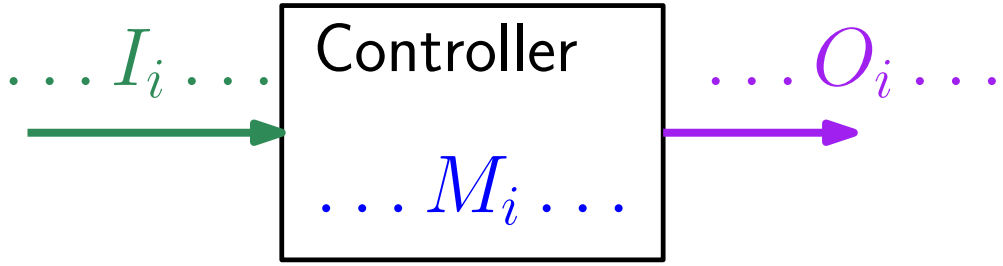
Temporal correction

Compute faster than the reactivity constraint.

Programming: the Functional Part

Remarks

- O_i depends *only* on the $I_1, I_2, ..., I_i$;
- Computations performed with *bounded memory* M_i.



Programming is ...

- ... identifying inputs and outputs;
- ... defining:
  - The **output function** $O_i = f(I_i, M_i)$;
  - The **transition function** $M_{i+1} = g(I_i, M_i)$

Programming: the Temporal Part

Difficulties

- Inputs do not arrive simultaneously, so:
  - When do we start computing ?
  - How to interpret a late input: Absent ? Delayed ?
- The controller usually comprises several activities

  $\Rightarrow$ how to schedule them?
- WCET estimation is hard!

The Asynchronous Approach

Parallel activities ⇒ Concurrent multi-task implementation.

```
int T1(){ // prio = P1
  while(true){
    readSensor();
    setActuator1();
    wait(100ms);
  }
}

int T2(){ // prio = P2
  while(true){
    readSensor();
    lock(l);
    updateState();
    unlock(l);
  }
}
```

```
int T3(){ // prio = P3
  while(true){
    whenever(event){
      lock(l);
      readState();
      updateState();
      unlock(l)
      setActuator2();
    }
  }
}
```

```
void scheduler(){
  if(Round_Robin){
    ...
  }
  if(FIFO){
    ...
  }
  if(PRIO){
    nextT = f(P1,P2,P3);
}
```

- Problem #1: **Inter-task communications**
- Problem #2: **Scheduling**

⇒ **Globally non-deterministic!!**

Plan

Requirements

- Design correctness: because safety is crucial!
- Need to reason formally about the operations of the system

$\Rightarrow$ Design methodologies and tools built on solid mathematical foundations

Let's Design Languages That ...

- Natively support **deterministic functional concurrency**
- **Programming style** natural for engineers
- Build on a **simple formal model**
- Can be compiled down to executable code
- Follow **synchronous execution** philosophy

```
Initialize Memory
for each input event do
  Compute Outputs
  Update Memory
end
```

```
Initialize Memory
for each clock tick do
  Read Inputs
  Compute Outputs
  Update Memory
end
```

The Synchronous Approach

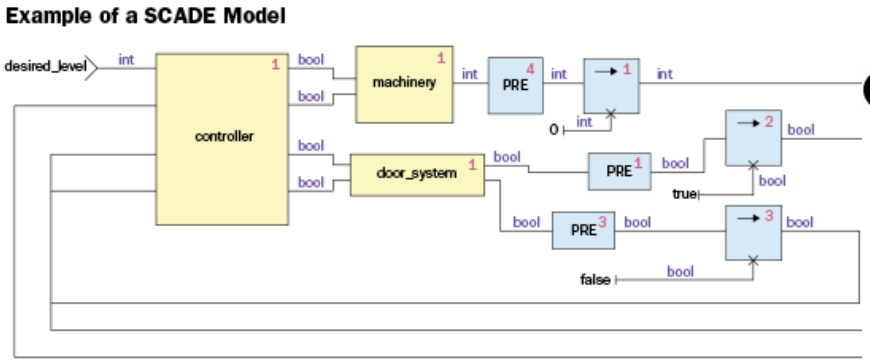
Real-time is replaced by a simplified, abstract, *logical time*.

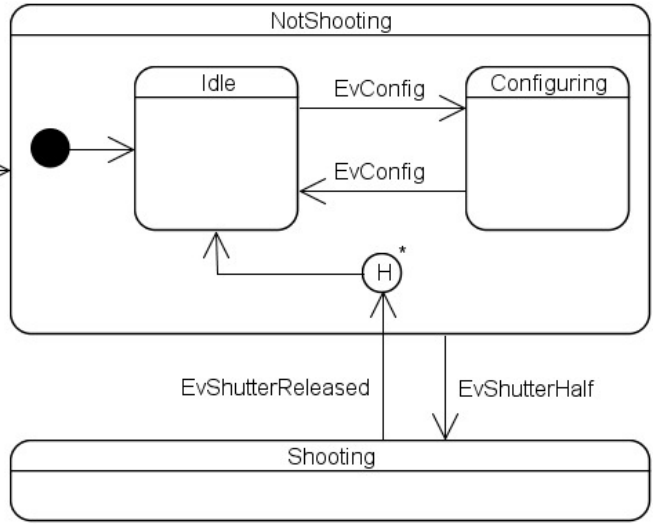
- *Instant*: one reaction of the system;
- Logical time: sequence of *instants*;
- The program describes what happens during each instant;
- *Synchronous hypothesis*: computations **complete before the next instant**. If so:
  - ⇒ We can ignore time inside an instant, only the order matters;
  - ⇒ We are only interested in how instants are chained together.

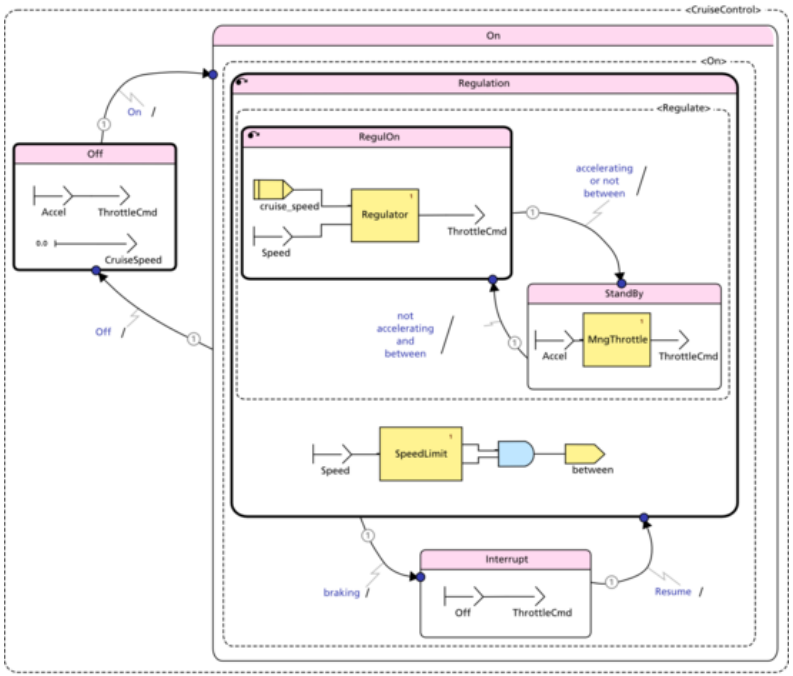
Lustre[Caspi1987] Esterel[Berry1984] Signal[LeGuernic1991]

Programming Styles Natural for Engineers

Essentially: block diagrams and FSMs.







Synchronous Languages vs Others

Advantages

- **Semantics defined formally** ⇒ enables formal proofs and provable compilation;
- **High abstraction** level ⇒ less work for the programmer, more for the compiler;
- **Bounded** memory and execution time;
- Barely needs an OS : **scheduling is defined in the language's semantics**

Disadvantages

- Produced code less efficient than hand-written code;
- Synchronous hypothesis hard to ensure (WCET not solved, distributed systems);
- Not well-suited for multi-rate systems.

Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The LUSTRE language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?

Flows and Clocks

- LUSTRE is a **data-flow** language: every expression and variable is a flow;
- **Flow**: infinite sequence of values + clock;
- **Clock**: defines when a flow is present (has a value).

Example

time	0	1	2	3	4	...
x	3	4	5	2	6	...
y	True			False	True	...

nb: for today, we forget about clocks

Point-Wise Extension of Classic Operators

Example

time	0	1	2	3	...
c	True	False	True	False	...
x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
x+y	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	...
`if c then x else y`	x_0	y_1	x_2	y_3	...

Delay Operator

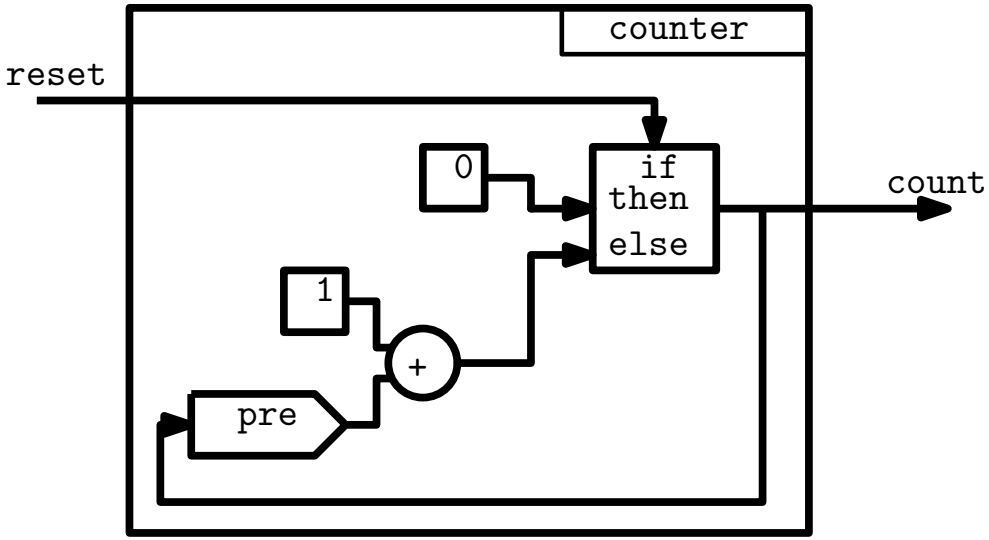
- The `pre(x)` operator denotes the **previous value** of a flow `x`;
- Undefined for the first instant;
- Usually combined with the initialization operator `->`: `x->y` is x on the first instant, y otherwise.

Example

time	0	1	2	3	...
x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
`pre x`		x_0	x_1	x_2	...
`y -> pre x`	y_0	x_0	x_1	x_2	...

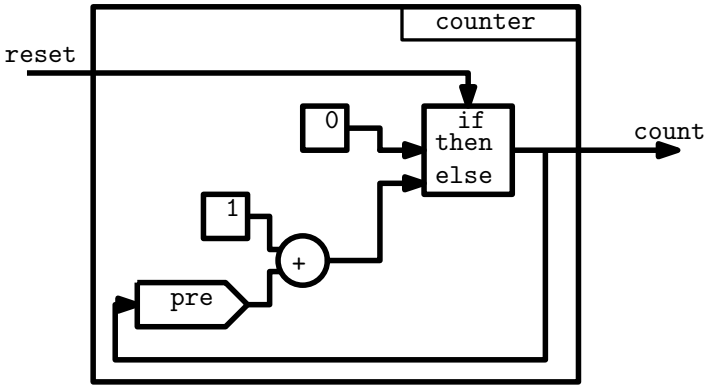
Example: A Resettable Counter

```
node counter(reset:bool) returns (count:int)
let
  count = 0 -> if reset then 0 else pre(count)+1;
tel
```



Example: A Resettable Counter

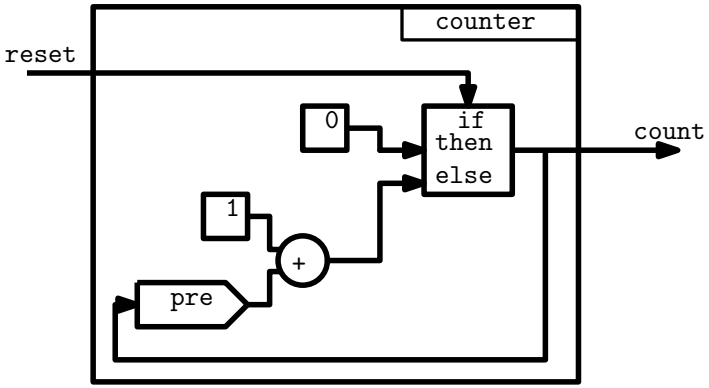
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset						...
count						...

Example: A Resettable Counter

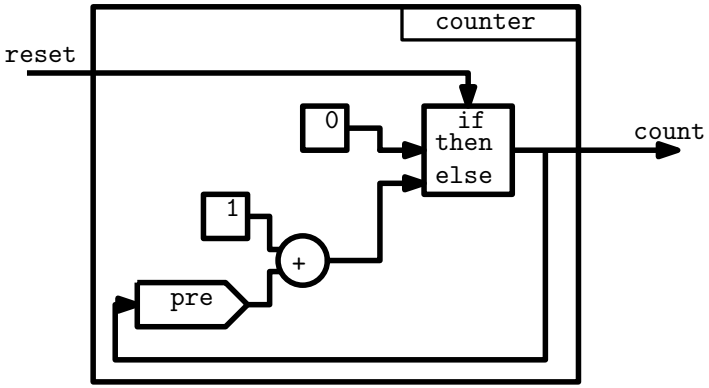
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False					...
count	0					...

Example: A Resettable Counter

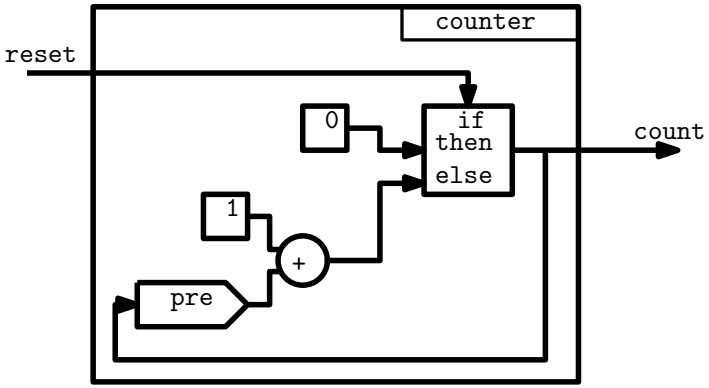
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False				...
count	0	1				...

Example: A Resettable Counter

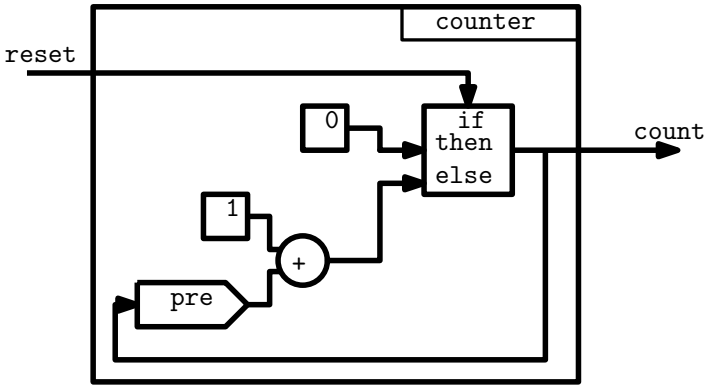
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False	False			...
count	0	1	2			...

Example: A Resettable Counter

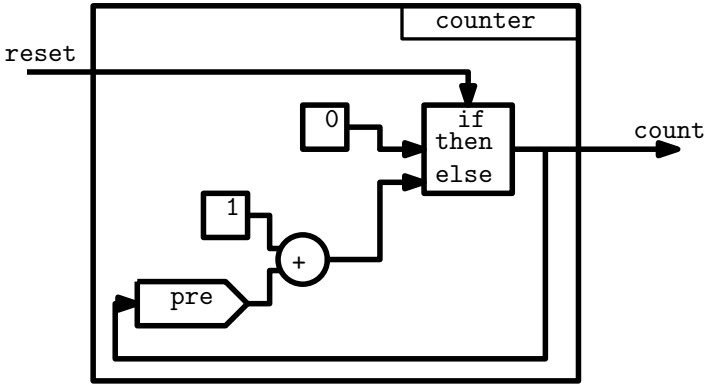
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False	False	True		...
count	0	1	2	0		...

Example: A Resettable Counter

```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False	False	True	False	...
count	0	1	2	0	1	...

Running Example

Let's:

- count modulo 9
- and display counter on a 7-seg
- reset if a button is pressed
- blink a led one over two "cycles".

```
node cpt(reset:bool) returns
     (sevseg:  int; led_on: bool) ;
let
   sevseg = 0  -> if (reset or pre(sevseg = 9))
                  then 0
                  else pre(sevseg)+1;
   led_on = true -> not pre(led_on);
tel
```

Running Example: Desired Result

Demo!

Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The LUSTRE language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?

Development Cycle for LUSTRE

✔ Write the synchronous program;

- Check **program** validity:
  - Causality (no cycle)
  - Initialisation analysis (pre)
  - **Clock calculus** ▶No access to absent values.

▶these checks are done at compile time

```
x = y;
y = x; -- forbidden
```

LUSTRE in the Development Cycle

- ✔ Write the synchronous program;
- ✔ Check program validity
- **Compile** ▶generation of C code

Sequential Code Generation - Goal

Generate a (C) program of the form:

```
init(memory)
each period do
  read(inputs);
  outputs=f(M,inputs);
  memory=g(M,inputs)
  write(outputs)
done
```

step()

- Goal here : generate `init`, `f`, `g`, infinite loop

Simple Syntax-Based Code Generation

Each node is compiled into a separate procedure:

- flow definition ▶ variable assignment;
- pointwise operator ▶ classical operator;
- `pre`, `->` ▶ memories;
- `when` ▶ tests (if).
- sequentialization of the equation system.

Compilation of Pre, Example

```
node foo(i:int) returns (dec:int)
let
  dec = 0 -> i;
tel
```

compiles into (**new init variable**):

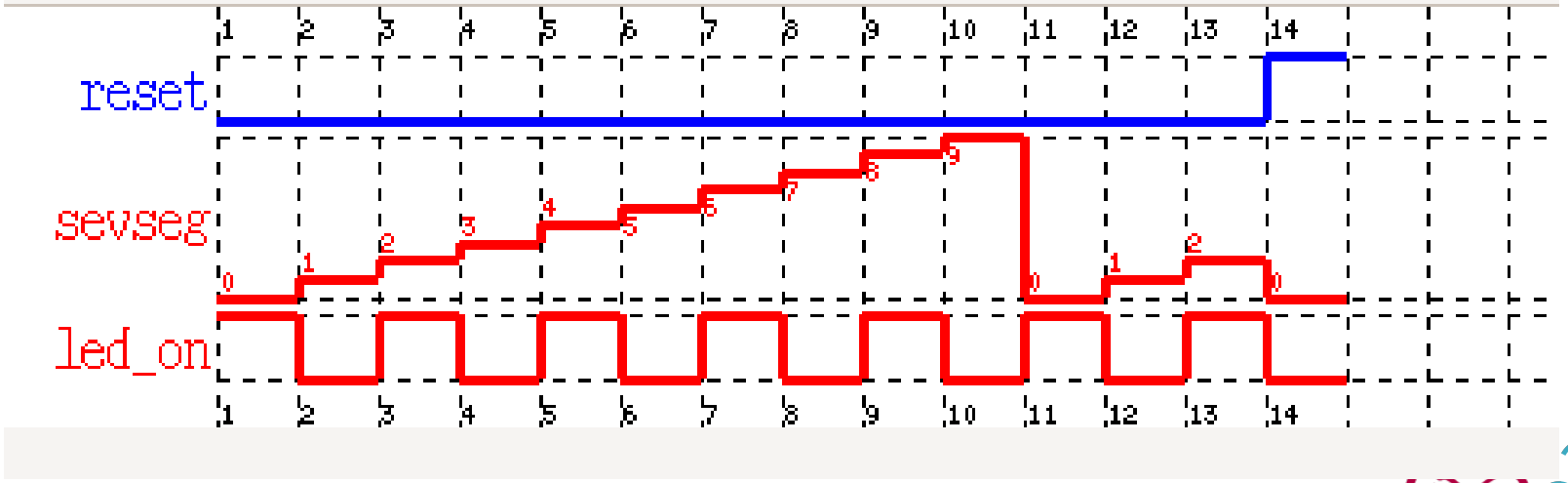
```
if init then dec = 0 else dec = i;
```

Compilation of our Example as a Simulation Loop

```
node cpt(reset:bool) returns
     (sevseg:  int; led_on: bool) ;
let
   sevseg = 0  -> if (reset or pre(sevseg = 9))
                  then 0
                  else pre(sevseg)+1;
   led_on = true -> not pre(led_on);
tel
```

- the compilation generates a .c and a main for simulation.

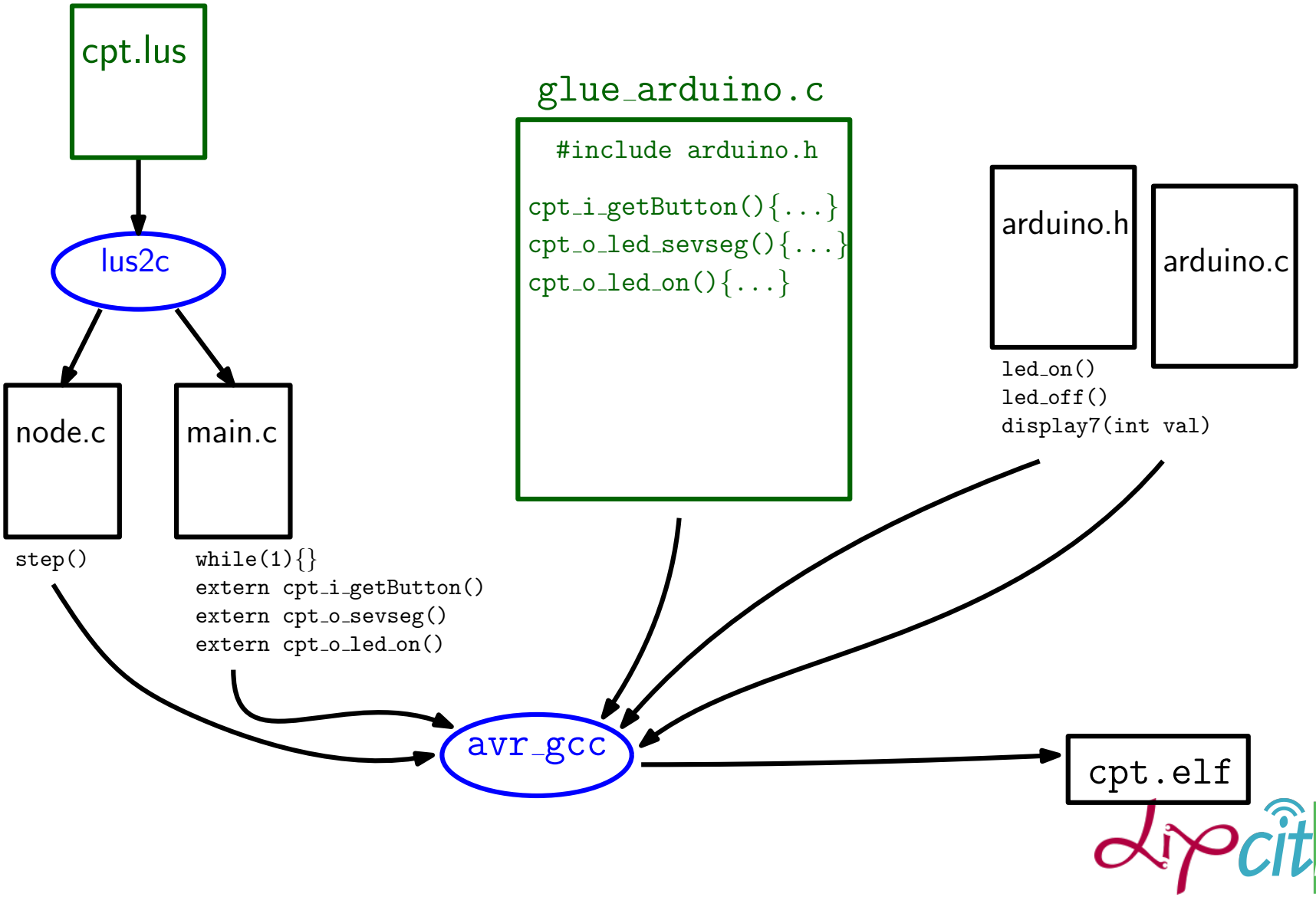
Demo (state + main).



Lustre in the Development Cycle

- ✔ Write the synchronous program (formal, high abstraction level);
- ✔ Compile : it generates C code (medium abstraction level);
- Write the **integration program**:
  - Read inputs on sensors;
  - Call the synchronous program;
  - Apply outputs to actuators.
- One more compilation: synchronous code + integration code ⇒ generates assembly (low abstraction level).

Running example : Complete Compilation Process



Glue Code: Input and Output Functions

Demo: glue code and interface.

Beyond Basic-Lustre

Around LUSTRE, a huge ecosystem:

- Formal verification, test, controler synthesis.
- Some work on data structures (arrays, iterators).
- Extensions with automata (Scade).
- A multithreaded implementation.
- Globally Asynchronous Locally Synchronous systems.

Plan

Conclusion 1/2

- Lustre (resp. synchronous languages): niche language ▶success
- Before : the (specialized) programmers already wrote the same kinds of programs as specifications (but without proof or automatic code generation)
- After : clarity, safety is easier ▶engineer is "happy". ▶she has Scade[2]

[2] Esterel Technologies

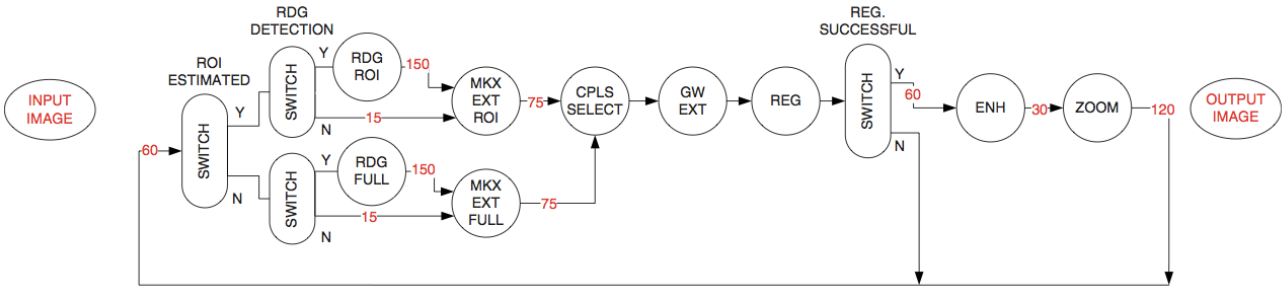
Conclusion 2/2

Synchronous programming is not dead: numerous research topics around it!

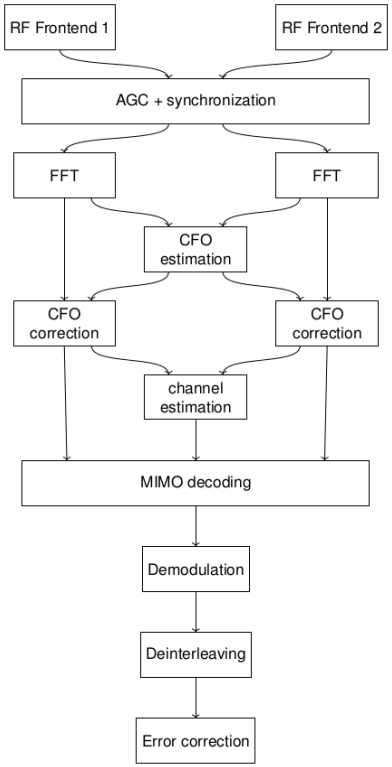
- Hybrid (discrete / dense time) models
- WCET estimation
- Targetting multi-core platforms
- Certification of compilers
- Relaxing synchrony to address streaming applications

Plan

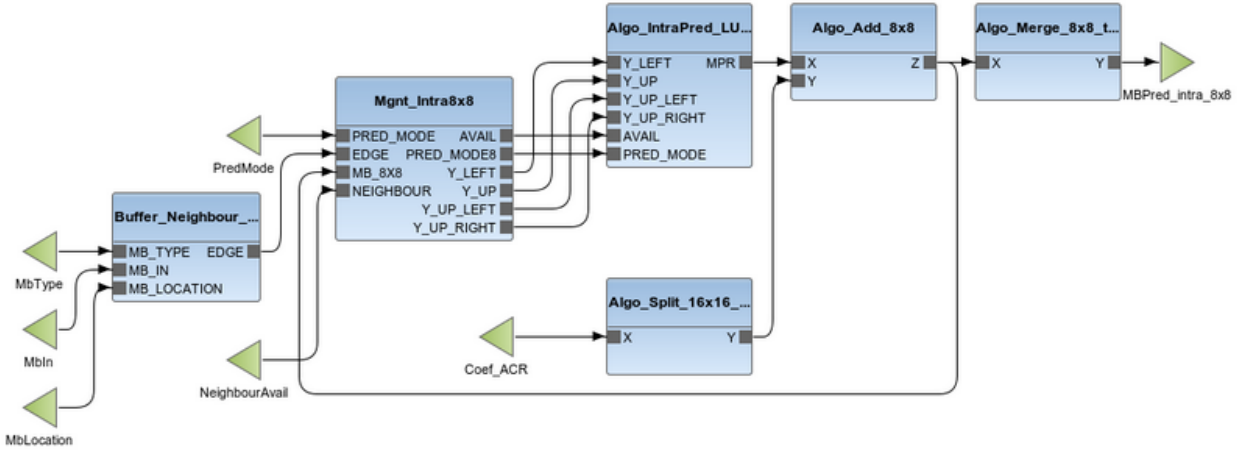
Dataflow Applications Examples



Medical image processing [Albers2012]

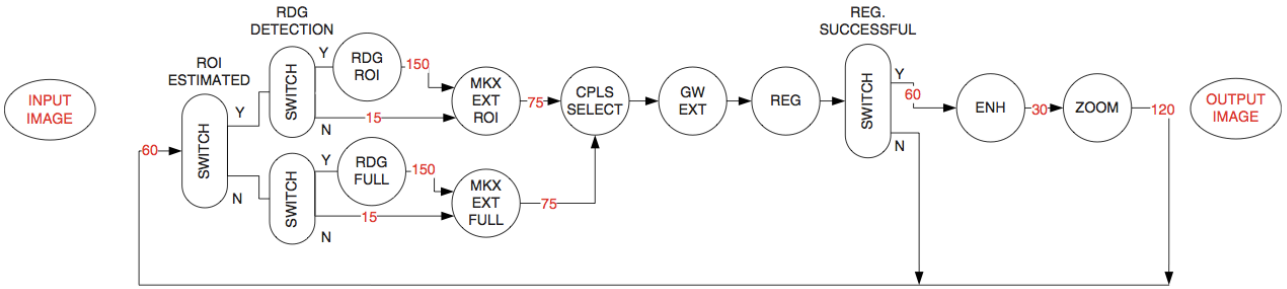


Software Defined Radio [Dardaillon2014]

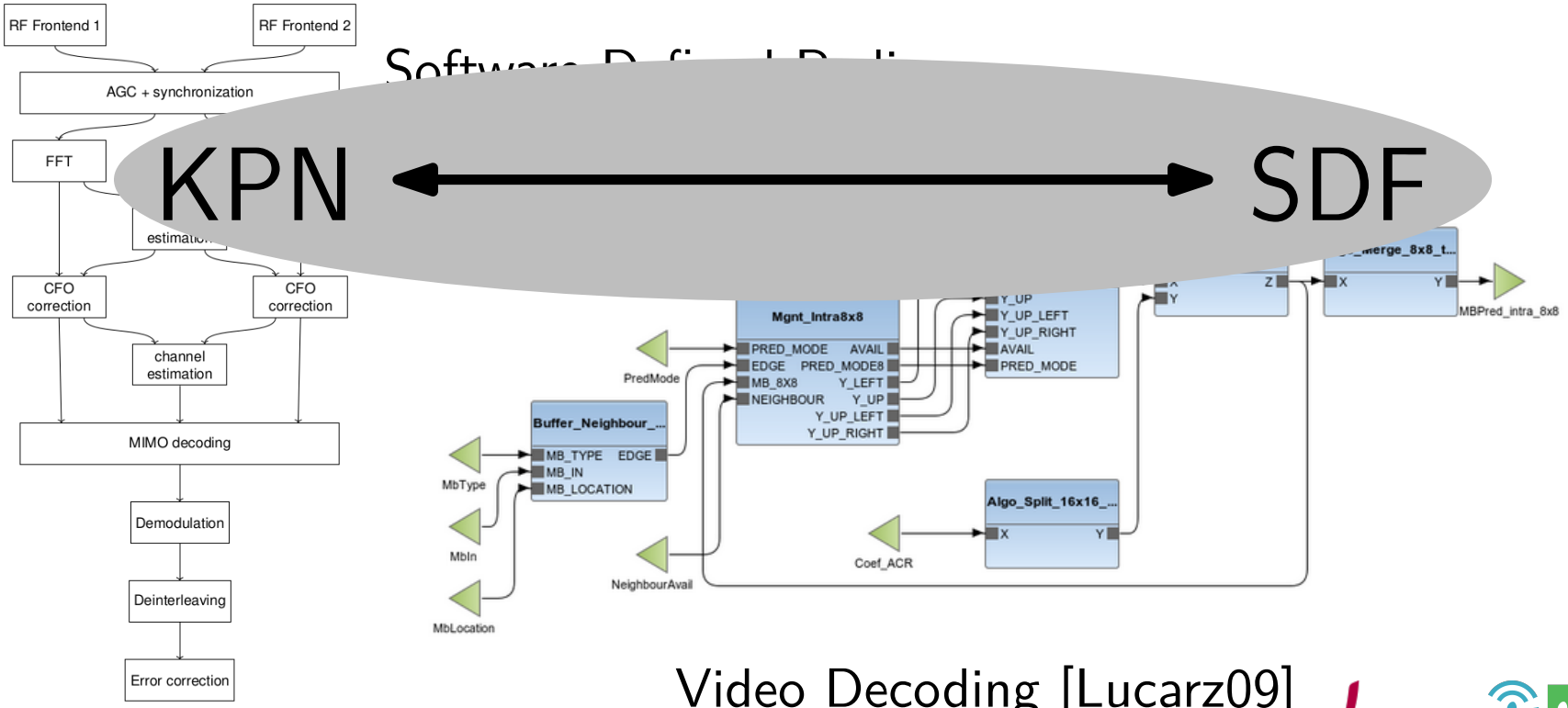


Video Decoding [Lucarz09]

Dataflow Applications Examples

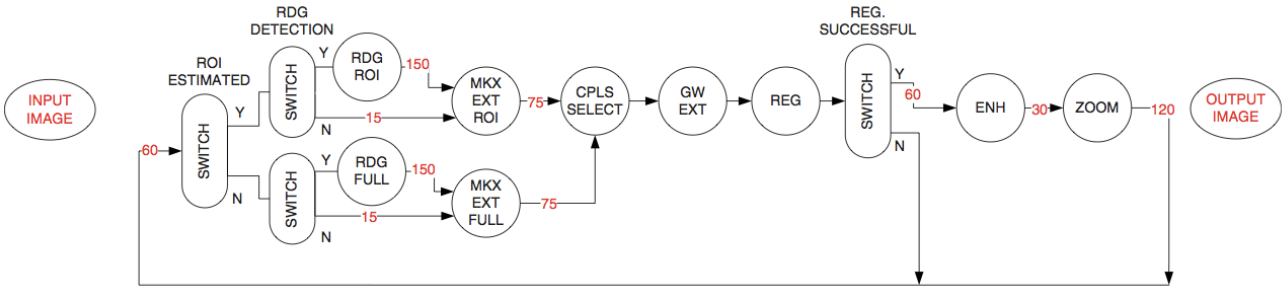


Medical image processing [Albers2012]

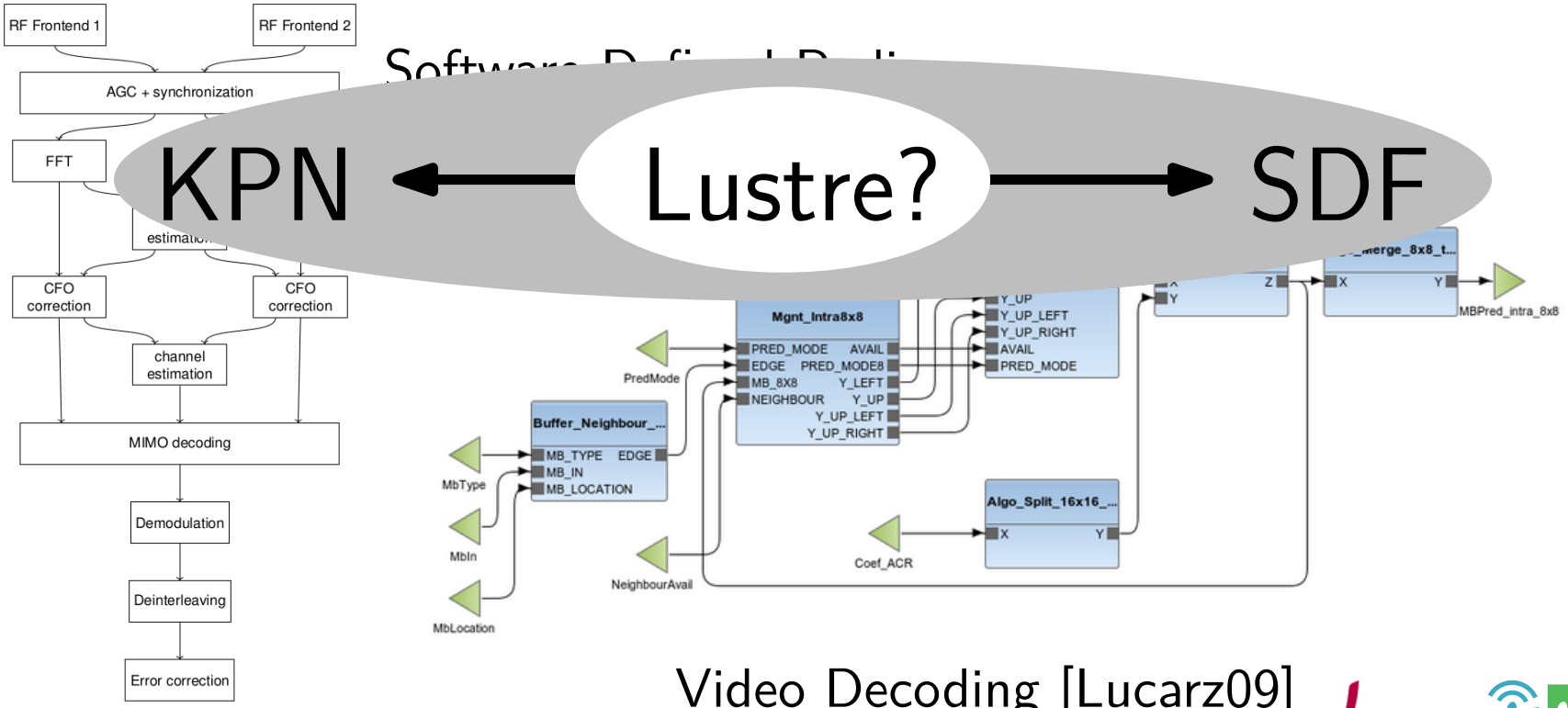


Video Decoding [Lucarz09]

Dataflow Applications Examples

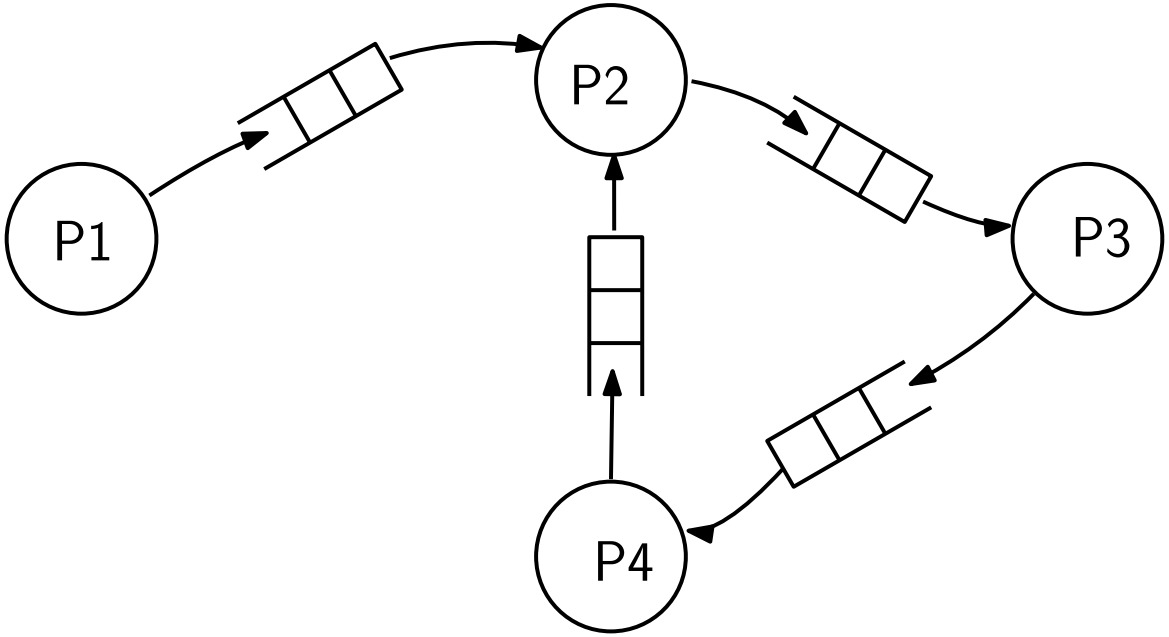


Medical image processing [Albers2012]



Video Decoding [Lucarz09]

Kahn Process Networks[3]



- Processes communicate through unbounded FIFOs
- Non-blocking writes / Blocking reads $\rightarrow$ deterministic
- Necessary whenever you can't predict production rates statically (eg compression)

[3] [Kahn1974]

Lustre vs KPN

Main differences

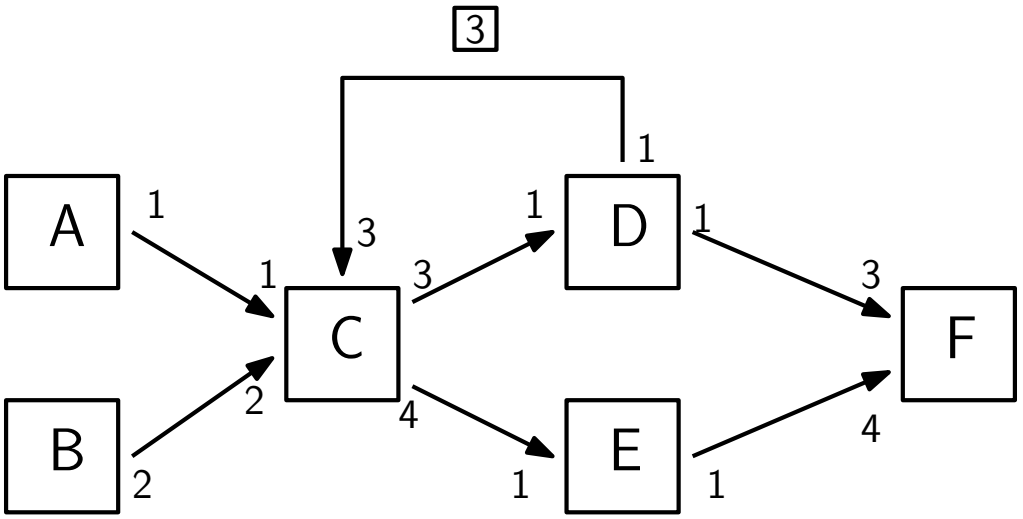
- Lustre's flows are ... not buffers.
  - Just data dependencies
- If you need FIFO buffers, you need to encode them explicitely
  - Including synchronization
- This is error-prone and very inefficient

Lustre vs KPN

OK, but KPNs assume un-bounded memory, so in practice ...

- ... Live with it and dimension your buffers "by experience" [RVC-Cal]
- ... Choose a trade-off, eg give clocks and have the type system dimension buffers for you, eg with Lucy-n [Plateau2010]

[Synchronous] Static DataFlow[4]



- Restricted Kahn Processes
- Number of tokens produced/consumed is statically known

⇒ Statically schedulable

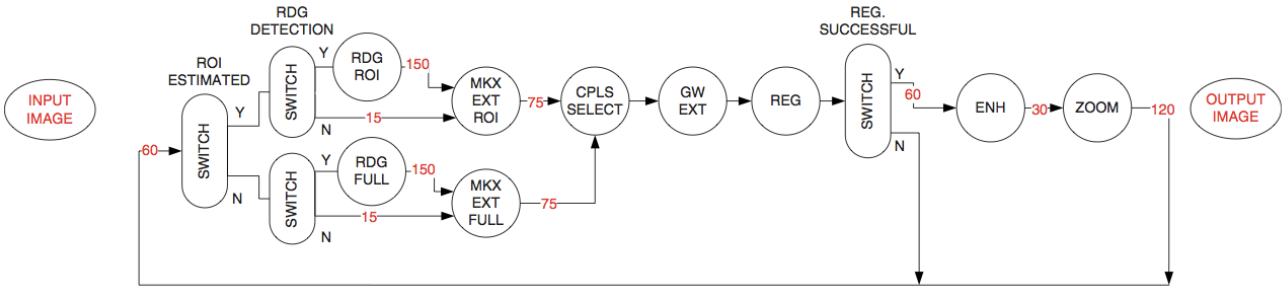
[4] [Lee1987]

Lustre vs SDF

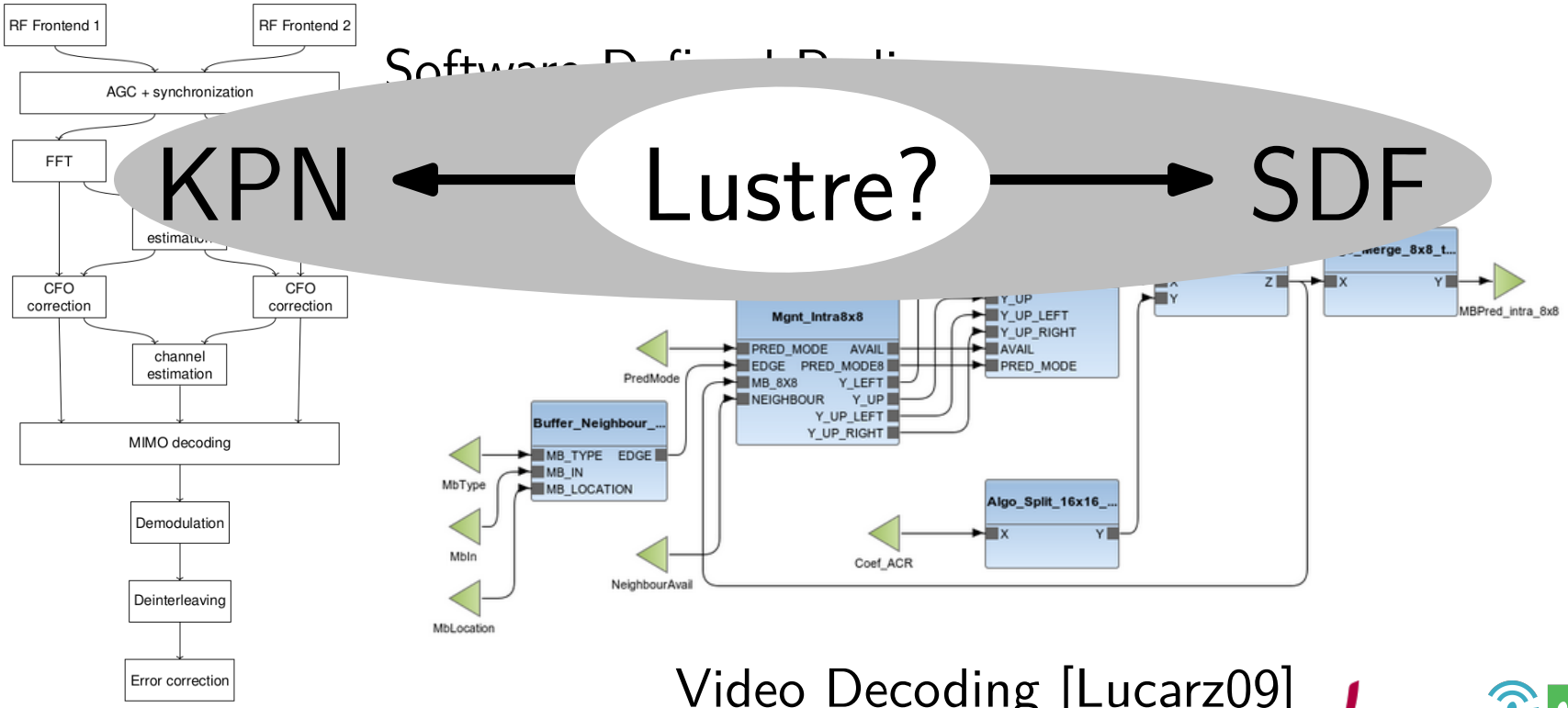
Main differences

- Fifos need to be hand-coded in Lustre
- SDF doesn't allow for absence of values (multi-clocked graphs)
  - Put differently: only multi-periodic designs can be constructed in SDF.

The End



Medical image processing [Albers2012]



Video Decoding [Lucarz09]

Credits

- *Julien FORGET (Cristal)* teaching notes
- *Abdoulaye GAMATIE (LIRMM)*, Synchronous Programming of Real-Time Systems with the Signal language, course at Telecom Lille 1, 2012;
- *Pascal RAYMOND (Verimag)*, various teaching notes.
- *Florence MARANINCHI (Verimag)*, Arduino inspiration.

Bibliography I

- A H R Albers and P H N de With.
  Task complexity analysis and qos management for mapping dynamic video-processing tasks on a multi-core platform.
  Journal of Real-Time Image Processing, 7(3):185–202, 2012.
- Gérard Berry and L. Cosserat.
  The Esterel synchronous programming language and its mathematical semantics.
  Research Report RR-0327, INRIA, 1984.

Bibliography II

- P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice.
  Lustre: A declarative language for real-time programming.
  In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.
- Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin, and Henri-Pierre Charles.
  A compilation flow for parametric dataflow: Programming model, scheduling, and application to heterogeneous mpsoc.

  In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded*

Bibliography III

Systems, CASES '14, pages 8:1–8:10, New York, NY, USA, 2014. ACM.

- G. Kahn.
  The semantics of a simple language for parallel programming.
  In *Information processing*. North-Holland, 1974.
- P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire.
  Programming real-time applications with signal.
  Proceedings of the IEEE, 79(9):1321–1336, Sep 1991.
- Edward A Lee and David G Messerschmitt.
  Synchronous data flow.
  Proceedings of the IEEE, 75(9):1235–1245, 1987.

Bibliography IV

- I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Deforges.
  Reconfigurable video coding on multicore.
  Signal Processing Magazine, IEEE, 26(6):113 –123, november 2009.
- Florence Plateau.
  Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée.
  PhD thesis, Université Paris-Sud 11, January 2010.
- Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet.
  Orcc: Multimedia development made easy.
  In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 863–866. ACM, 2013.