Constraining end-to-end delays in multi-periodic Lustre programs

Timothy Bourke

Marc Pouzet

Michel Angot

Vincent Bregeon

Matthieu Boitrel

25 November 2021, Synchron, La Rochette

Context

- Standard practice: design an application as a set of periodically executed tasks that communicate through shared variables.
- Read data from sensors via a bus, compute through sequences of cyclic tasks, and write to actuators via the bus.

Context

- Standard practice: design an application as a set of periodically executed tasks that communicate through shared variables.
- Read data from sensors via a bus, compute through sequences of cyclic tasks, and write to actuators via the bus.
- Airbus project "All-in-Lustre"
  - *Current system*: each task is a Lustre node ($\approx 5\,000$) with separate constraints on order and latency.
  - *Desired system*: "All-in-Lustre": compose the nodes in a single Lustre program with new features for specifying periods and execution constraints.
  - Generate sequential code for cyclic execution on a single-processor platform.
  - Base period = 5ms. Tasks at 10ms, 20ms, 40ms, and 120ms.
  - Tasks are already chopped up into small pieces.

Slow flows

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

```
$ presseail example1.ail -v --glpk --print
```

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate $1/3$ (period $= 3$)
- Calculate each one once every three cycles

```
$ presseail example1.ail --glpk --compile 1 --print
```

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles
- The 5 calculations in this program are synchronous
  relative to the period
  even if they are not necessarily simultaneous
  relative to the base clock
- s3 = s1 + s2 is well clocked
  since s1 :: 1/3, s2 :: 1/3, and s3 :: 1/3.

```
$ presseail example1.ail --glpk --compile 1 --print
```

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles
- The 5 calculations in this program are synchronous
  relative to the period
  even if they are not necessarily simultaneous
  relative to the base clock
- s3 = s1 + s2 is well clocked
  since s1 :: 1/3, s2 :: 1/3, and s3 :: 1/3.
- Causality applies 'across' a period and 'within' an instant:
  $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ()$

```
$ presseail example1.ail --glpk --compile 1 --print
```

Declare and constrain resources

```
resource cpu : int

node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int)
    requires (cpu = 5);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    resource cpu <= 4;
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = filter(s2);
    () = write(s3);
tel
```

- Declare *named weights* to represent resources required per cycle
  - Simple proxies for worst-case execution time
  - Network bus accesses
- Use to constrain scheduling
- normally: resource balance cpu

Macro-scheduling of equations

- Label each equation, scheduling assigns a phase offset
  - Lustre with annotations as an ersatz intermediate language
  - label(filter_0) phase(1 % 3) s2 = filter(s1);
- Phase offsets are constrained by
  - Data dependencies in the source program
  - Resource constraints
  - Latency constraints...
- Phase offset (and latency) are implementation details
  - They are relative to the base rate, not the equation rate
  - Program semantics is independent of phase offsets

Macro-scheduling using Integer Linear Programming (ILP)

Usual Workflow

1. `$ presseail example2.ail --write-lp example2.lp`
   writes the scheduling constraints to a file
2. Call `cplex`
3. `$ presseail example2.ail --read-sol example2.sol --compile 1`
   reads the solution and generates code

Macro-scheduling using Integer Linear Programming (ILP)

Usual Workflow

1. `$ presseail example2.ail --write-lp example2.lp`
   writes the scheduling constraints to a file
2. Call `cplex`
3. `$ presseail example2.ail --read-sol example2.sol --compile 1`
   reads the solution and generates code

Testing simple examples

- `$ presseail example2.ail --glpk --compile 1`

```
Minimize
  rmax.equ

Subject to
  pw.def0.filter: pw.0.filter + pw.1.filter + pw.2.filter = 1
  pw.def1.filter: 2 pw.2.filter + pw.1.filter - p.filter = 0
  ...
  depd.wr.p.read.p.filter_5: p.filter - p.read >= 0
  ...
  rbnd.cpu_8: 5 pw.0.filter_1 + 5 pw.0.filter_0 + 5 pw.0.filter <= 8
  rbnd.cpu_7: 5 pw.1.filter_1 + 5 pw.1.filter_0 + 5 pw.1.filter <= 8
  rbnd.cpu_6: 5 pw.2.filter_1 + 5 pw.2.filter_0 + 5 pw.2.filter <= 8

Bounds
  0 <= p.read < 3
  0 <= p.filter < 3
  ...
General
  p.read p.filter ...
Binary
  pw.0.read pw.1.read pw.2.read pw.0.filter ...
End
```

Changing speeds: 1

```
resource cpu : int

node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int)
    requires (cpu = 5);

node main(s0 : int) returns (s4 : int)
var s1, s2, s3 : int :: 1/3;
let
    resource cpu <= 8;
    s1 = filter(s0 when (0 % 3));
    s2 = filter(s1);
    s3 = filter(s2);
    s4 = current(0, (2 % 3), s3);
tel
```

- x when c
  - c is for '(sampling) choice'
  - sub-sampling of a stream
  - fast-to-slow rate change
- current(0, c, x)
  - stutter stream elements
  - slow-to-fast rate change

  y = merge c x ((0 fby y) when not c)

```
$ presseail example3.ail --glpk --compile 1
```

Changing speeds: 2

r = w when (i % n)

- (i % n): take the ith of every n elements.
- n is the rate of w relative to r
  E.g., for w :: 1/4 and r :: 1/8, n is 2.
- It can be deduced from the clocks, but is useful for readability.
- It implies a lower bound on the scheduling of the equation.

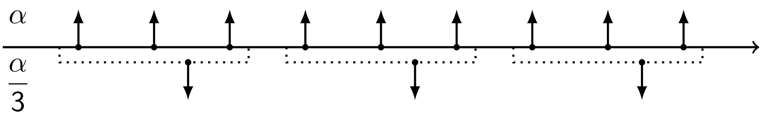
Changing speeds: 2

r = w when (i % n)

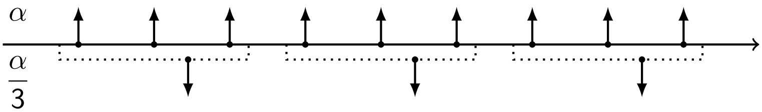
- (i % n): take the ith of every n elements.
- n is the rate of w relative to r
  E.g., for w :: 1/4 and r :: 1/8, n is 2.
- It can be deduced from the clocks, but is useful for readability.
- It implies a lower bound on the scheduling of the equation.

r = current(0, (i % n), w)

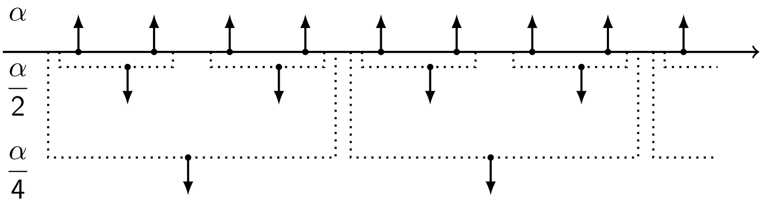
- (i % n): update r from the ith of every n elements of w.
- n is the rate of r relative to w
  E.g., for r :: 1/4 and w :: 1/8, n is 2.
- The first argument is a constant giving the default value.
  - Needed even for (0 % n)...
    r = current0(w)?
- It implies an upper bound on the scheduling of the equation.



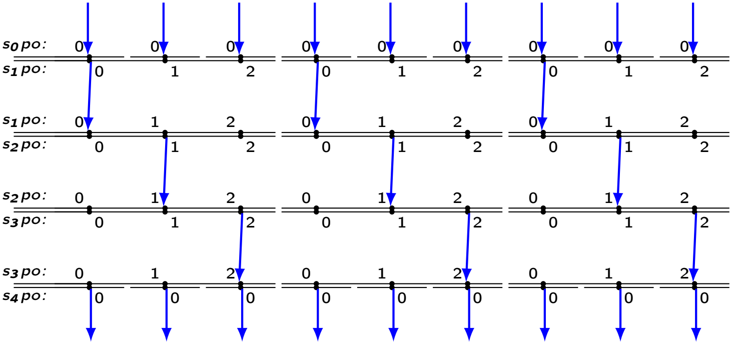
One slow tick is synchronous with three fast ones.



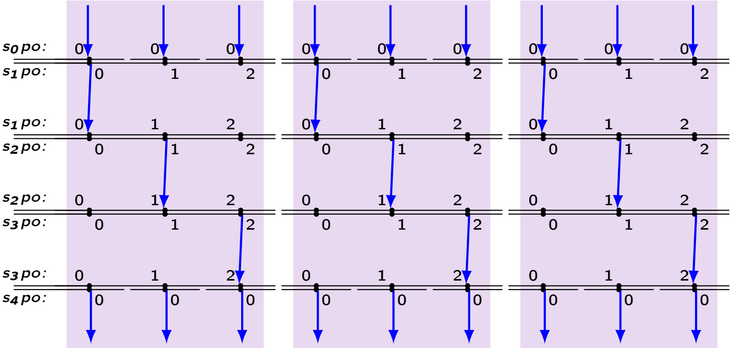
One slow tick is synchronous with three fast ones.



Implementation problems: assign computations to phases, buffer values




```
node main (s0 : int) returns (s4 : int);
var s1, s2, s3 : int :: 1/3;
let
  label(filter) phase(0 % 3) s1 = filter(s0 when (0 % 3));
  label(filter_0) phase(1 % 3) s2 = filter(s1);
  label(filter_1) phase(2 % 3) s3 = filter(s2);
  label(s4_2) phase(0 % 1) s4 = current(0, (2 % 3), s3);
tel
```




```
node main (s0 : int) returns (s4 : int);
var s1, s2, s3 : int :: 1/3;
let
  label(filter) phase(0 % 3) s1 = filter(s0 when (0 % 3));
  label(filter_0) phase(1 % 3) s2 = filter(s1);
  label(filter_1) phase(2 % 3) s3 = filter(s2);
  label(s4_2) phase(0 % 1) s4 = current(0, (2 % 3), s3);
tel
```

Changing speeds whenever

```
resource cpu : int

node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int)
    requires (cpu = 5);

node main(s0 : int) returns (s4 : int)
var s1, s2, s3 : int :: 1/3;
let
    resource cpu <= 8;
    s1 = filter(s0 when (? % 3));
    s2 = filter(s1);
    s3 = filter(s2);
    s4 = current(0, (? % 3), s3);
tel
```

- Manual choices in when and current over-constrains scheduling.
- Impractical for 100 000s variables!
- So write (? % n) for "don't care".
- Scheduling still respects casuality
  - y = x when (? % n) — x_0 before y.
  - y = current(0, (? % n), x) — x before y_{n-1}.
- What about determinism?
- Synchron 2018, F. Maraninchi "Non-determinism reference semantics"

```
$ presseail example4.ail --print --glpk --print
```

Code generation: 1

Generalize the clock-directed scheme

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

- `--compile n` generates n step functions
  - For the ith step function, step_i, `List.filter_map` equations by phase offset.
  - Generate dependency graph ignoring variables not in step_i —macro-scheduling guarantees they will already have been calculated.
  - Micro-schedule equations in step_i w.r.t. dependencies and phase offset/rate.
- Generate multiple Obc step methods, buffer values in state variables.
- Optimize the Obc by joining adjacent `case` statements.

Specialized case construct

```
case (state(c_3) mod 3) {
  0: { skip }
  1: { state(s2) := filter(state(s1)) }
  2: { skip }
  else undefined
};
case (state(c_3) mod 3) {
  0: { state(s1) := filter(s0) }
  1: { skip }
  2: { skip }
  else undefined
};
```

```
case (state(c_3) mod 3) {
  0: { state(s1) := filter(s0) }
  1: { state(s2) := filter(state(s1)) }
  2: { skip }
  else undefined
};
```

Code generation: 2

The 'else undefined' simplifies optimisation under (implicit) invariants

```
case (state(c_3) mod 24) {
  7:  { state(x) := read_real() }
  23: { y := read_real() }
  else undefined }
```

$\Rightarrow$

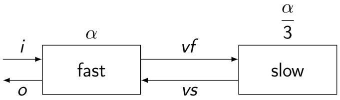
```
if (state(c_3) mod 24 = 7) {
    state(x) := read_real()
} else {
    y := read_real()
}
```

```
case (state(c_3) mod 24) {
  7:  { state(x) := read_real() }
  15: { skip }
  23: { y := read_real() }
  else undefined }
```

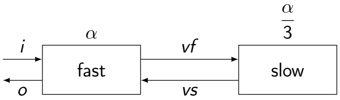
$\Rightarrow$

```
case (state(c_3) mod 24) {
  7:  { state(x) := read_real() }
  15: { skip }
  23: { y := read_real() }
  else undefined }
```

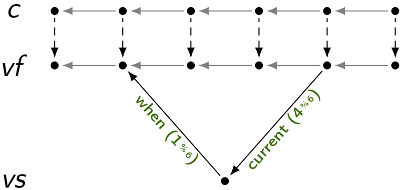
```
c = 0 fby (c + 1);
vf = current(0, (4 % 6), vs) + c;
vs = vf when (1 % 6) + 5;
```



c = 0 fby (c + 1);
vf = current(0, (4 % 6), vs) + c;
vs = vf when (1 % 6) + 5;

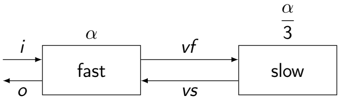


Causal dependencies

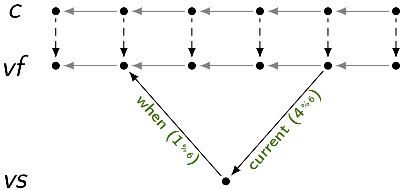


Causality, Scheduling,and Semantics

c = 0 fby (c + 1);
vf = current(0, (4 % 6), vs) + c;
vs = vf when (1 % 6) + 5;



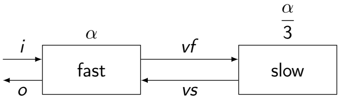
Causal dependencies



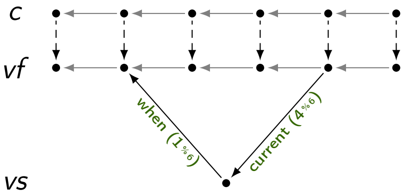
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Causality, Scheduling,and Semantics

c = 0 fby (c + 1);
vf = current(0, (4 % 6), vs) + c;
vs = vf when (1 % 6) + 5;



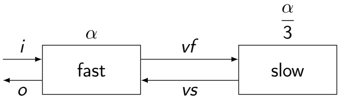
Causal dependencies



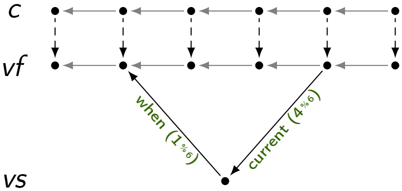
<table>
<tr><td>vf</td><td>0</td><td>1</td><td>2</td><td>3</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>28</td><td>29</td><td>· · ·</td></tr>
<tr><td>c</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>· · ·</td></tr>
<tr><td>vs</td><td colspan="6">6</td><td colspan="6">18</td><td>· · ·</td></tr>
</table>

Causality, Scheduling,and Semantics

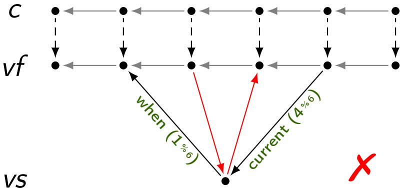
```
c = 0 fby (c + 1);
vf = current(0, (4 % 6), vs) + c;
vs = vf when (1 % 6) + 5;
```



Causal dependencies



Scheduling dependencies

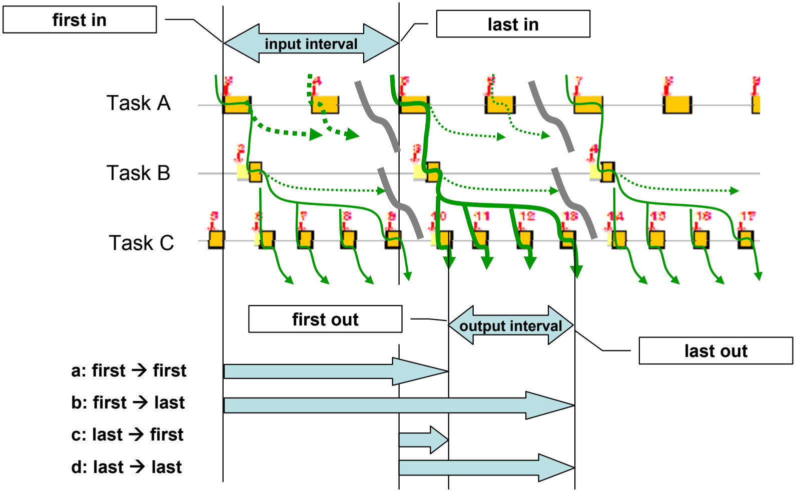


vf	0	1	2	3	10	11	12	13	14	15	28	29	...
c	0	1	2	3	4	5	6	7	8	9	10	11	...
vs			6						18				...

Bounding End-to-End Latency

```
latency_chain VAR_04101 8
 (data21510 -> data05224 -> data13157
  -> data26032 -> data03229 -> data31722
  -> data21555 -> data29595 -> data36187
  -> data13349 -> data06816 -> data01252
  -> data18196 -> data20921 -> data16645
  -> data11226 -> data29115 -> data23284
  -> data36163 -> data14490);
```

- Specify critical computation chains
- Bound the end-to-end latency in terms of the base clock
- Generate additional scheduling constraints



[Feiertag, Richter, Nordlander, and Jonsson (2008): A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics]

Flowgraph links

direct communications

`x`	direct, write-before-read, coforward
`last x`	direct, read-before-write, cobackward

Flowgraph links

direct communications

`x`	direct, write-before-read, coforward
`last x`	direct, read-before-write, cobackward

fast-to-slow communications

`x when (? % n)`	first-write-before-read, coforward
`(last x) when (? % n)`	read-before-last-write, cobackward

Flowgraph links

direct communications

`x`	direct, write-before-read, coforward
`last x`	direct, read-before-write, cobackward

fast-to-slow communications

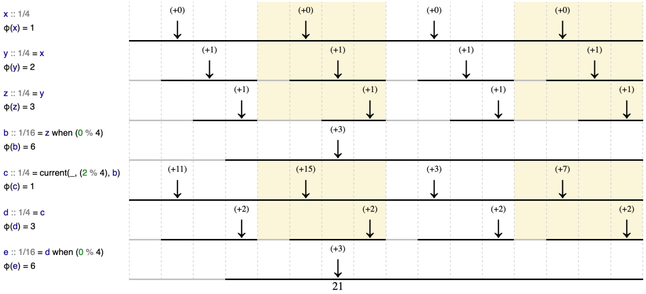
`x when (? % n)`	first-write-before-read, coforward
`(last x) when (? % n)`	read-before-last-write, cobackward

slow-to-fast communications

`current(c, (? % n), x)`	coforward or cobackward
`current(c, (? % n), last x)`	forbidden

Showlatency demo

latency_example <= 10 ✎



new variable: y definition: when (☑ CoF) sampling ratio: 4 add drop

latency bound: 10 update edit solve

minimize
obj: x

subject to
chainlat_39: -15 · ne_37 + r_26 + d - c + r_13 + r_10 + z - x <= 10
notenabled_38: ne_37 + d.0.b_24 <= 1
chainlat_36: -15 · ne_34 + r_26 + d - c + r_16 + r_10 + z - x <= 10

Related Work

- Prelude
- Lucy-n
- Harmonic 1-synchronous clocks / affine clocks
- anything missing?

Related Work: Prelude

- Language and compiler [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems] [Pagetti, Forget, Boniol, Cordovilla, and Lesens (2011): Multi-task Implementation of Multi-periodic Synchronous Programs]
- Specify task periods and offsets.
- Compose real-time primitives to express communication patterns.
- Semantic model based on tagged signals
- Generate and schedule a set of OS tasks
  - WCET, release times, deadlines
  - Adapt existing scheduling algorithms to respect data dependencies (causality).

Our work

- Task periods only—offsets as an implementation detail
- Every “task” completes within a cycle.
- No scheduling, just generate imperative code.

Related Work: Lucy-n

- Model [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems] and language [Mandel, Plateau, and Pouzet (2010): Lucy-n: a n-Synchronous extension of Lustre]
- Flexible scheduling patterns (`0010(010)`) and buffering
- Sophisticated type-based analysis for causality and buffer sizes
- Less focus on code generation

Our work

- Less flexible scheduling
- Buffering is implicit and very limited

Related Work: Iooss et al.

- "1-synchronous" programs [Iooss, Pouzet, Cohen, Potop-Butucaru, Souyris, Bregeon, and Baufreton (2020): 1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom]
- Two-element clocks: [*phase*, *period*]
  ($0^k 1 0^{n-k-1}$ or $0^k(10^{n-1})$, where n is the period and $0 \leq k < n$ is the phase
- Related to work on affine clocks
  - » [Curic (2005): Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints]
  - » [Smarandache, Gautier, and Le Guernic (1999): Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints]
- Several operators: `when`, `current`, `delay`, `delayfby`, `buffer`, `bufferfby`
- Prototype in Heptagon: introduces (lots of) `when`s and `merge`s

Our work

- Simpler clocks, fewer operators, implicit buffering
- Generate imperative code directly

Conclusion

- Simple prototype with ILP scheduling and basic code generation.
- Tested on Airbus example with 5000 nodes

Work in progress

- Reviewing literature on end-to-end timing properties of task chains.
- Adding support for explicit sample choices.
- Allowing variables in sample choices?

References I

- Biernacki, D., J.-L. Colaço, G. Hamon, and M. Pouzet (June 2008). "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. Tucson, AZ, USA: ACM Press, pp. 121–130.
- Cohen, A., M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet (Jan. 2006). "N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems". In: *Proc. 33rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2006)*. Charleston, SC, USA: ACM Press, pp. 180–193.
- Curic, A. (Sept. 2005). "Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints". PhD thesis. Grenoble, France: Université Joseph Fourier.
- Feiertag, N., K. Richter, J. Nordlander, and J. Jonsson (Nov. 2008). "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics". In: *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2008, co-located with RTSS 2008)*. Barcelona, Spain.

References II

- Forget, J., F. Boniol, D. Lesens, and C. Pagetti (Mar. 2010). "A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems". In: *Proc. 25th ACM Symp. Applied Computing (SAC'10)*. Ed. by S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung. Sierre, Switzerland: ACM, pp. 527–534.
- Iooss, G., M. Pouzet, A. Cohen, D. Potop-Butucaru, J. Souyris, V. Bregeon, and P. Baufreton (Mar. 2020). "1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom".
- Mandel, L., F. Plateau, and M. Pouzet (June 2010). "Lucy-n: a n-Synchronous extension of Lustre". In: *Proc. 10th Int. Conf. on Mathematics of Program Construction (MPC' 2010)*. Ed. by C. Bolduc, J. Desharnais, and B. Ktari. Vol. 6120. LNCS. Québec City, Canada: Springer, pp. 288–309.
- Pagetti, C., J. Forget, F. Boniol, M. Cordovilla, and D. Lesens (Sept. 2011). "Multi-task Implementation of Multi-periodic Synchronous Programs". In: *Discrete Event Dynamic Systems* 21.3, pp. 307–338.

References III

- Smarandache, I. M., T. Gautier, and P. Le Guernic (Sept. 1999). "Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints". In: *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99)*. Ed. by J. M. Wing, J. Woodcock, and J. Davies. Vol. 1709. LNCS. Toulouse, France: Springer, pp. 1364–1383.