Scheduling and compiling rate-synchronous programs with end-to-end latency constraints

Timothy Bourke

with Vincent Bregeon (Airbus) and Marc Pouzet

[Bourke, Bregeon, and Pouzet (2023): Scheduling and Compiling Rate-Synchronous Programs with End-to-End Latency Constraints]

Inria Paris — PARKAS Team
École normale supérieure, PSL University

17 October 2023, Systèmes réactifs synchrones MPRI 2-23-1

Context

- Set of periodic tasks communicating through variables:
  - » read data from sensors via a bus,
  - » compute via sequences of tasks, and
  - » write to actuators via the bus.

Context

- Set of periodic tasks communicating through variables:
  - » read data from sensors via a bus,
  - » compute via sequences of tasks, and
  - » write to actuators via the bus.

- Lustre:
  synchronous-reactive dataflow language for "model-based design"
  [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- Scade:
  modernized, industrial version
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

Context

- Set of periodic tasks communicating through variables:
  - » read data from sensors via a bus,
  - » compute via sequences of tasks, and
  - » write to actuators via the bus.

- Lustre:
  synchronous-reactive dataflow language for "model-based design"
  [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- Scade:
  modernized, industrial version
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

Airbus project "All-in-Lustre"

- *Original system*: $\approx 5\,000$ Lustre nodes
  $+$ separate constraints on order
- *Desired system*: a single Lustre program with features for periods and end-to-end latencies.
- Nodes at 10ms, 20ms, 40ms, and 120ms.
- *Implementation*: sequential code, period $= 5$ms

Context

- Set of periodic tasks communicating through variables:
  - » read data from sensors via a bus,
  - » compute via sequences of tasks, and
  - » write to actuators via the bus.

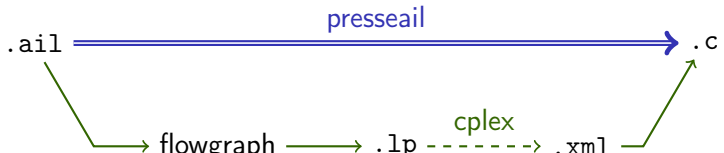
- Lustre:
  synchronous-reactive dataflow language for "model-based design"
  [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- Scade:
  modernized, industrial version
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

Airbus project "All-in-Lustre"

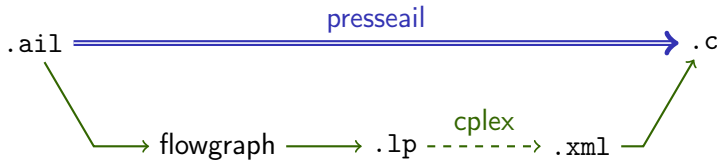
- *Original system*: $\approx 5\,000$ Lustre nodes
  $+$ separate constraints on order
- *Desired system*: a single Lustre program with features for periods and end-to-end latencies.
- Nodes at 10ms, 20ms, 40ms, and 120ms.
- *Implementation*: sequential code, period $= 5$ms
- Workload already chopped up into small pieces.
- Each node loops in < 5ms.

Overview: compilation using Integer Linear Programming (ILP)



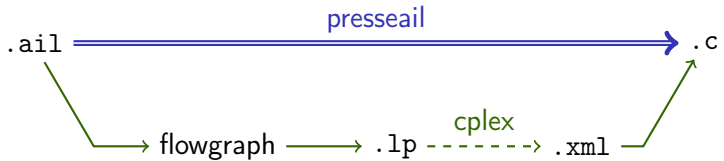
Overview: compilation using Integer Linear Programming (ILP)

- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock



Overview: compilation using Integer Linear Programming (ILP)

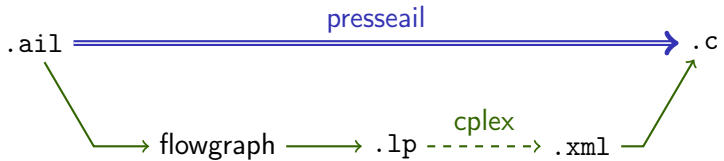
- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock
- One or more `step` functions
- Called cyclically at the base rate



Overview: compilation using Integer Linear Programming (ILP)

- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock

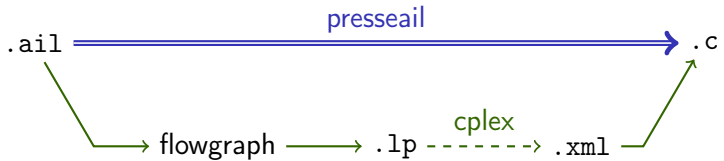
- One or more `step` functions
- Called cyclically at the base rate



- Vertex = node
- Arc from producer to consumer
- Independent of source language

Overview: compilation using Integer Linear Programming (ILP)

- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock
- One or more `step` functions
- Called cyclically at the base rate



- Vertex = node
- Arc from producer to consumer
- Independent of source language

- Data dependencies
- Load balancing
- End-to-end latency

Slow flows

```
resource cpu : int;

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 once every three cycles

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

Slow flows

```
resource cpu : int;

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 once every three cycles

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

Slow flows

```
resource cpu : int;

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 once every three cycles
- The 5 calculations here 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

```
resource cpu : int;

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 once every three cycles
- The 5 calculations here 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
```

Changing speeds: 1

```
resource cpu : int

node filter(x : int)
  returns  (y : int);

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

- `x when c`
  - » c is for '(sampling) choice'
  - » sub-sampling of a stream
  - » fast-to-slow rate change
- `current(x, c)`
  - » stutter stream elements
  - » must declare an initial `last` value
  - » slow-to-fast rate change

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

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

Changing speeds: 2

```
r = w when (i % n)
```

- `(i % n)`: take the `i`th 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 `i`th 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(w, (i % n))
```

- `(i % n)`: repeat the initial `last` value `i` times, then repeat each `w` value `n` times.
- `n` is the rate of `r` relative to `w`
  E.g., for `r :: 1/4` and `w :: 1/8`, `n` is 2.
- It implies an upper bound on the scheduling of the equation.

Changing speeds: 2

`r = w when (i % n)`

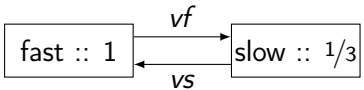
- `(i % n)`: take the `i`th of every `n` elements.
- `n` is the rate of `w` relative to `r`
  E.g., for `w :: 1/4` and `r :: 1/8`, `n` is 2.
- It can be deduced from the clocks, but is useful for readability.
- It implies a lower bound on the scheduling of the equation.

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

- `(i % n)`: repeat the initial `last` value `i` times, then repeat each `w` value `n` times.
- `n` is the rate of `r` relative to `w`
  E.g., for `r :: 1/4` and `w :: 1/8`, `n` is 2.
- It implies an upper bound on the scheduling of the equation.

- Write `(? % n)` if we don't care when values are sampled/updated.
- The schedule decides when sampling/updating occur; fixed at compile time.

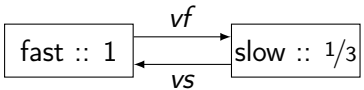
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

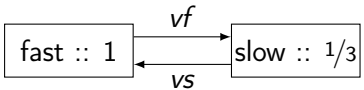
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

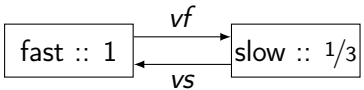
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

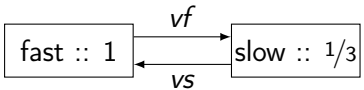
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

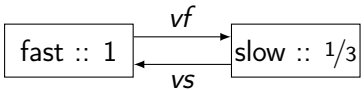
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

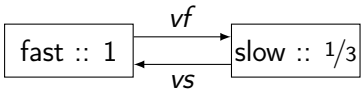
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

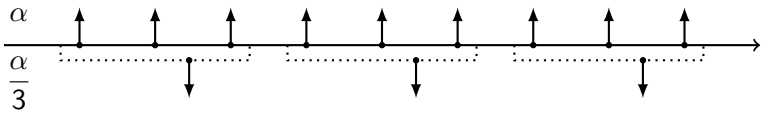
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
      n : int :: 1 last = 0;
     vs : int :: 1/3 last = 0;
let
  vf = n + current(vs, (2 % 3));
   n = (last n) + 1;
  vs = (vf when (1 % 3)) + 5;
tel
```

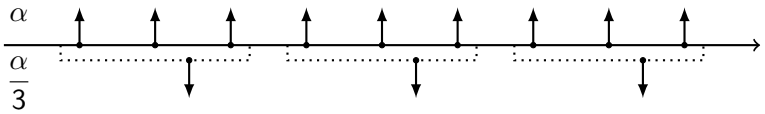
vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

Rate-synchronous Model

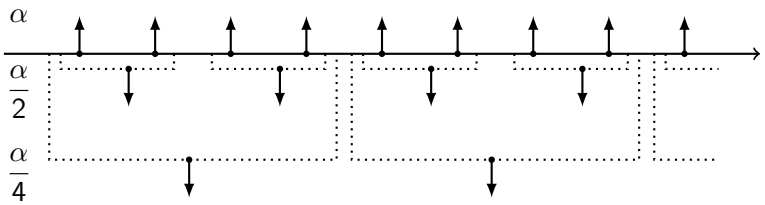


One slow tick is synchronous with three fast ones.

Rate-synchronous Model



One slow tick is synchronous with three fast ones.



- Calculations are synchronous relative to their periods
  but not necessarily simultaneous relative to execution cycles
- The compiler assigns computations to phases, buffering values if necessary

Syntax

$$
\begin{aligned}
eq &::= x \texttt{ = } e \mid x^* \texttt{ = } f\texttt{(}e^*\texttt{)} \\
e &::= c \mid x \mid \diamond\ e \mid e \oplus e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{last } x \\
&\quad \mid x \texttt{ when } s \mid \texttt{(last } x\texttt{) when } s \mid \texttt{current(}x\texttt{, } s\texttt{)} \\
s &::= \texttt{(}c \texttt{ \% } c\texttt{)} \mid \texttt{(? \% } c\texttt{)} \\
p &::= (d\ \texttt{;})^* \\
d &::= \texttt{resource } x \texttt{ : } ty \\
&\quad \mid \texttt{node } f\texttt{(}\ (x \texttt{ : } ty\ \texttt{;})^*\texttt{)} \texttt{ returns (}\ (x \texttt{ : } ty\ \texttt{;})^*\texttt{)} \texttt{ requires (}\ (x \texttt{ = } c\ \texttt{;})^*\texttt{)} \\
&\quad \mid \texttt{node } f\texttt{(}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^*\texttt{)} \texttt{ returns (}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^*\texttt{)} \\
&\quad\ \ \texttt{var } (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^* \texttt{ let } (((pragmas\ eq) \mid cst)\ \texttt{;})^+ \texttt{ tel} \\
pragmas &::= [\texttt{label(}x\texttt{)}]\ [\texttt{phase(}c \texttt{ \% } c\texttt{)}] \\
cst &::= \texttt{resource balance } x \\
&\quad \mid \texttt{resource } x\ rel\ c \\
&\quad \mid \texttt{latency } (\texttt{exists} \mid \texttt{forward} \mid \texttt{backward})\ rel\ c\ \texttt{(}\ x\ \texttt{,}\ x\ (\texttt{,}\ x)^*\texttt{)} \\
rel &::= \texttt{<=} \mid \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{>=}
\end{aligned}
$$

Valid programs are defined by clock typing

$$\frac{e_1 :: 1/n \qquad e_2 :: 1/n}{e_1 \oplus e_2 :: 1/n}$$

$$\frac{x :: 1/n}{\texttt{last}\ x :: 1/n}$$

$$\frac{x :: 1/m}{x\ \texttt{when}\ (\cdot \%\, n) :: 1/mn}$$

$$\frac{x :: 1/mn}{\texttt{current}(x, (\cdot \%\, n)) :: 1/m}$$

- No phase offsets in clock types, unlike
  - » Prelude: `rate(100, 0)`
    [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  - » Lucy-n: `(010), 00(00100)`
    [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems]
  - » 1-Synchronous: `[0, 2]`
    [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]
  - » Simulink: `[Ts, To]`
- Dataflow semantics: independent of phase offsets
- Generated code: phase offsets implement data dependencies.

Valid programs are defined by clock typing

$$\frac{e_1 :: 1/n \qquad e_2 :: 1/n}{e_1 \oplus e_2 :: 1/n}$$

$$\frac{x :: 1/n}{\texttt{last}\ x :: 1/n}$$

$$\frac{x :: 1/m}{x\ \texttt{when}\ (\cdot \%\, n) :: 1/mn}$$

$$\frac{x :: 1/mn}{\texttt{current}(x, (\cdot \%\, n)) :: 1/m}$$

- No phase offsets in clock types, unlike
  - » Prelude: `rate(100, 0)`
    [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  - » Lucy-n: `(010), 00(00100)`
    [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems]
  - » 1-Synchronous: `[0, 2]`
    [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]
  - » Simulink: `[Ts, To]`
- Dataflow semantics: independent of phase offsets
- Generated code: phase offsets implement data dependencies.

Stream-based Semantics

$$[\![e_1 \oplus e_2]\!](i) = [\![e_1]\!](i) \oplus [\![e_2]\!](i)$$

$$[\![\texttt{last}\ x]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i = 0 \\ [\![x]\!](i-1) & \text{otherwise} \end{cases}$$

$$[\![x\ \texttt{when}\ (s\,\%\,n)]\!](i) = [\![x]\!](n \cdot i + s)$$

$$[\![\texttt{current}(x, (s\,\%\,n))]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i < s \\ [\![x]\!](\lfloor {}^{i-s}/_{n} \rfloor) & \text{otherwise} \end{cases}$$

- Recursive equations on streams: $\mathbb{N} \to \mathbb{V}$
- $x_{\text{-}1}$ is the initial `last` value
- No explicit presence or absence
- Cf. Prelude (tagged-signal model)

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 = 4);

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`

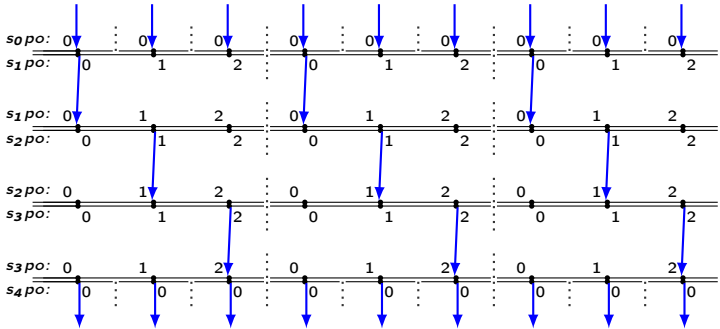
Declare and constrain resources

```
resource cpu : int

node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int)
  requires (cpu = 4);

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
```

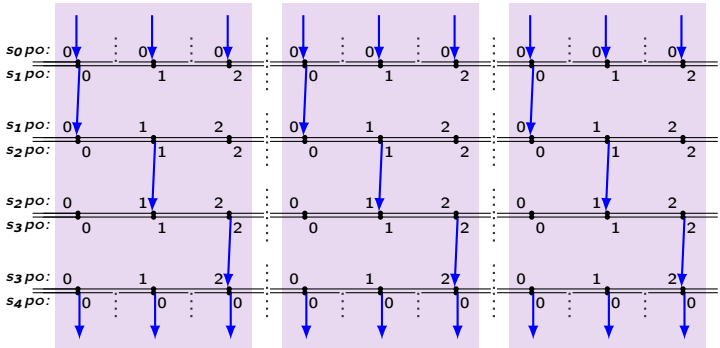
- Declare *named weights* to represent resources required per cycle
  - » Simple proxies for worst-case execution time
  - » Network bus accesses
- Use to constrain scheduling
- normally: `resource balance cpu`
- Trade-off *resource balancing* against *latency*: `latency_chain <= 0 (s0 -> s1 -> s2 -> s3);`

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




```
node main (s0 : int) returns (s4 : int)
var s1, s2 : int :: 1 / 3,
    s3 : int :: 1 / 3 last = 0;
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(s3, (2 % 3));
tel
```




```
node main (s0 : int) returns (s4 : int)
var s1, s2 : int :: 1 / 3,
    s3 : int :: 1 / 3 last = 0;
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(s3, (2 % 3));
tel
```

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`

ILP Problem

```
Maximize
 obj: x1 + 2 x2 + 3 x3 + x4
Subject To
 c1: - x1 + x2 + x3 + 10 x4 <= 20
 c2: x1 - 3 x2 + x3 <= 30
 c3: x2 - 3.5 x4 = 0
Bounds
 0 <= x1 <= 40
 2 <= x4 <= 3
General
 x4
End
```

- LP File Format Sections
  - 1. `Minimize` / `Maximize`: objective function
  - 2. `Subject to`: list of constraints
  - 3. `Bounds`: optional bounds on variables
  - 4. `General`: list of integer variables, **variables default to 'continuous'**
  - 5. `Binary`: $0 \leq$ integer variables ≤ 1
  - 6. `End`
- Constraints:
  - *(optional)* `label:`
  - sum of terms <= constant *(also >=, etc.)*
  - each term pairs a constant with a variable

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

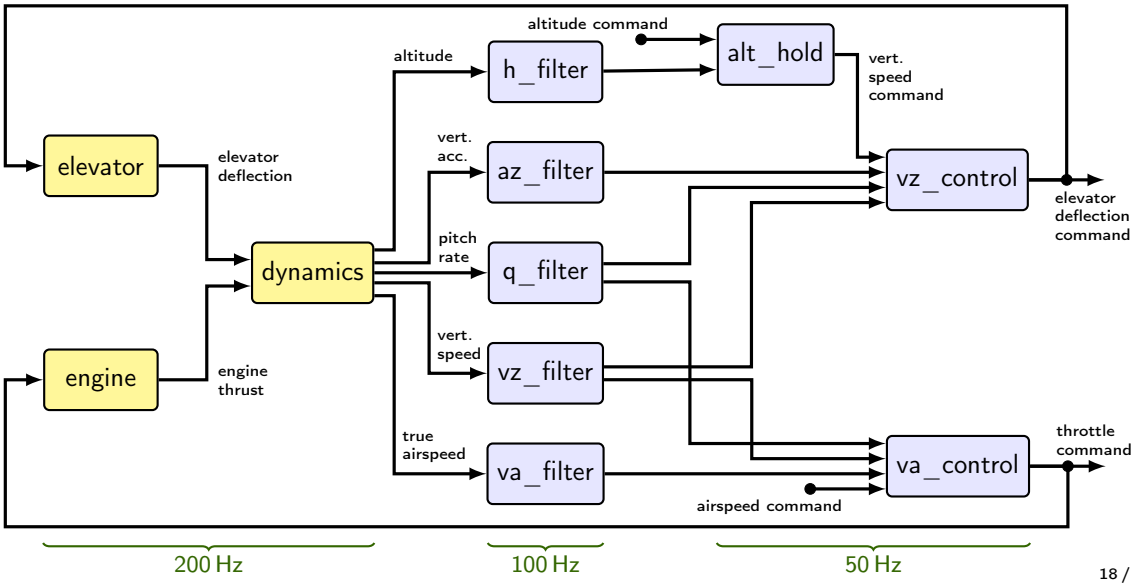
Prelude: Multi-periodic Sync. Prog. [Forget, Boniol, Lesens, and Pagetti (2008): A Multi-Periodic Synchronous Data-Flow Language]

- Language [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems] and compiler [Pagetti, Forget, Boniol, Cordovilla, and Lesens (2011): Multi-task implementation of multi-periodic synchronous programs]
- Extend Lustre with task periods/phases and WCET.
- Compose real-time primitives to express communication patterns.
- Generate and schedule a set of real-time tasks
  - » WCET, release times, deadlines
  - » Adapt existing scheduling algorithms to respect data dependencies

- "Don't Care" [Wyss, Boniol, Forget, and Pagetti (2012): A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming],
  Let the compiler decide if `c dc x` (`c fby? x`) is
  - » `c fby x`
  - » `x`

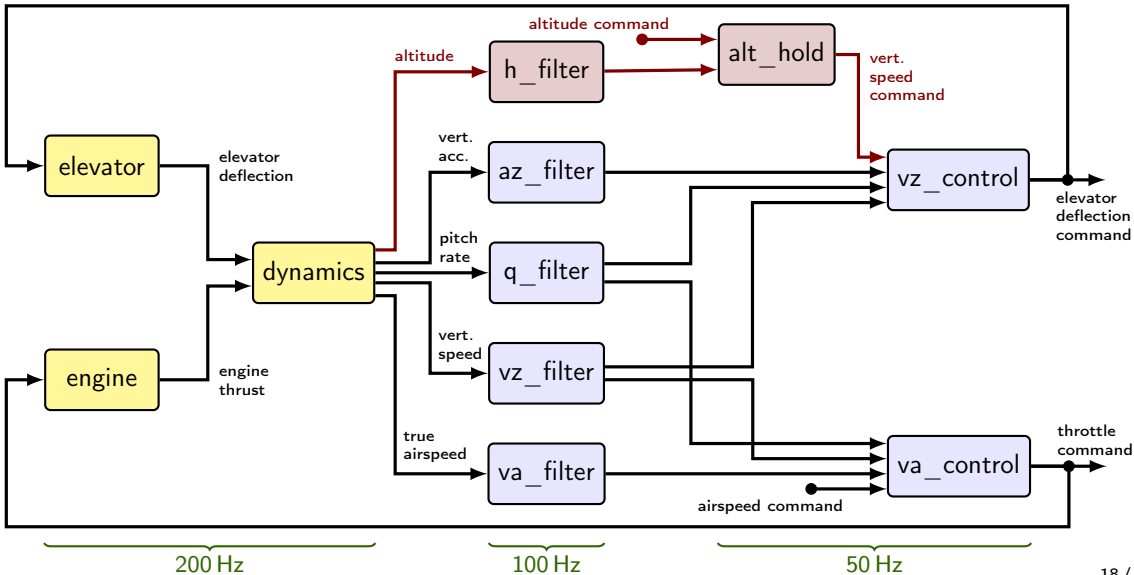
The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]



The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]



The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]

```
resource ops : int
node alt_hold (h_c, h_f : float) returns (vz_c : float) requires (ops = 201);
...
const H200 : rate = 1 / 2 (* base clock = 400Hz *)
const H100 : rate = 1 / 4
const H50  : rate = 1 / 8
const H10  : rate = 1 / 40

node assemblage1( h_c : float :: H10 last = 0.;   (* altitude command *)
                  va_c : float :: H10 last = 0.)  (* airspeed command *)
returns (d_th_c : float :: H50 last = 1.6402;     (* throttle command *)
         d_e_c  : float :: H50 last = 0.0186)     (* elevator deflection command *)
var  h_f : float :: H100; (* altitude *)
    vz_c : float :: H50;  (* vertical speed command *)
    ...
let
   h_f =  h_filter( h when (? % 2) );
  vz_c = alt_hold( current(h_c, (? % 5)), h_f when (? % 2) );
  ...
  resource balance ops;
  latency assemblage exists <= 2
    (dynamics, h_filter, alt_hold, vz_control, elevator);
tel
```

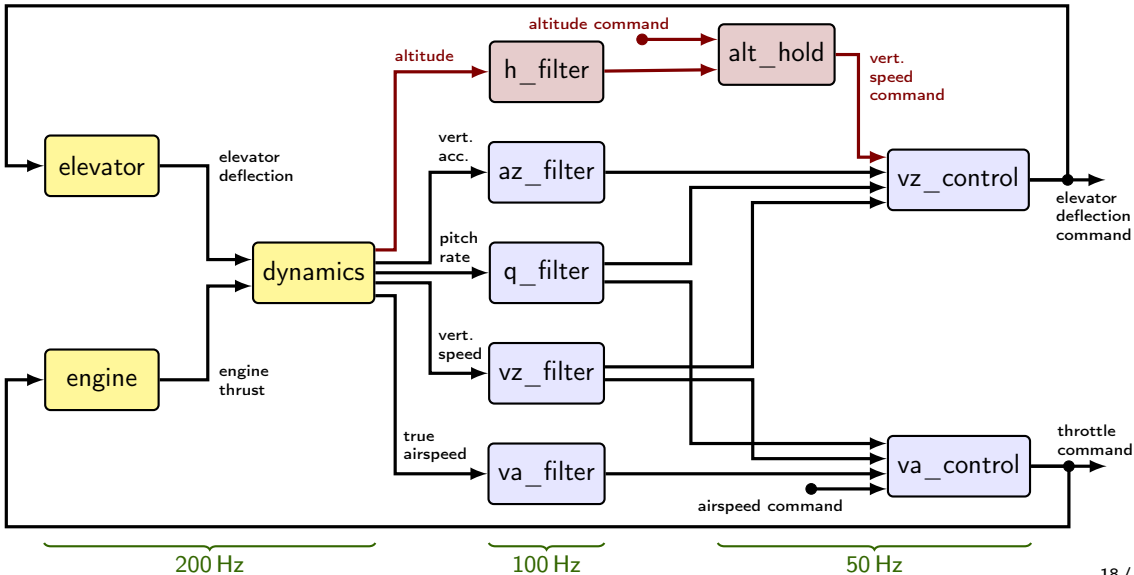
elevator deflection command

throttle command

200 Hz | 100 Hz | 50 Hz

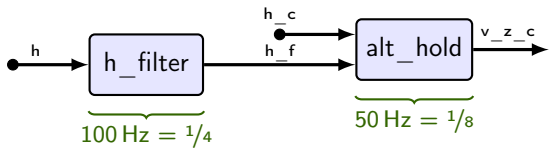
The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]



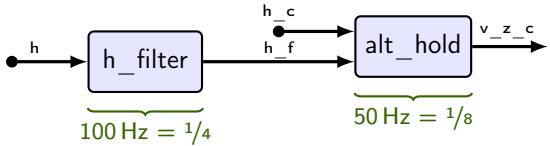
Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
 h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



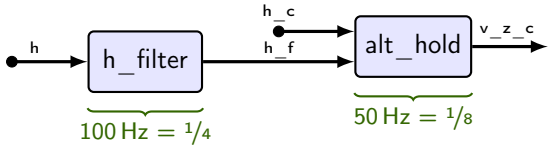
Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
 h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



Compilation: Model ⇒ Scheduling ⇒ Generated Code

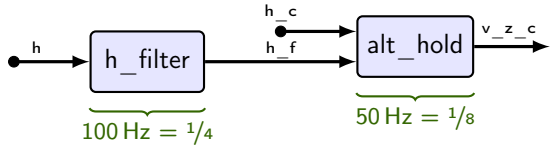
```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```




```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```

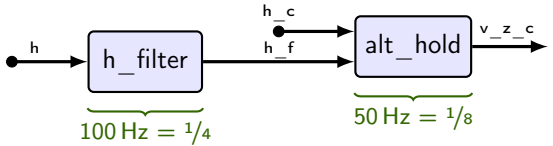


- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



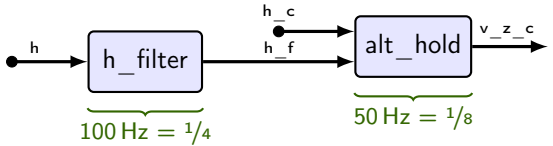
- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

f (concomitance)

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```

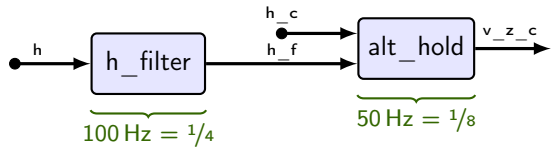


- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();       // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 0: va_control(); break;
    case 6: alt_hold();       // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```

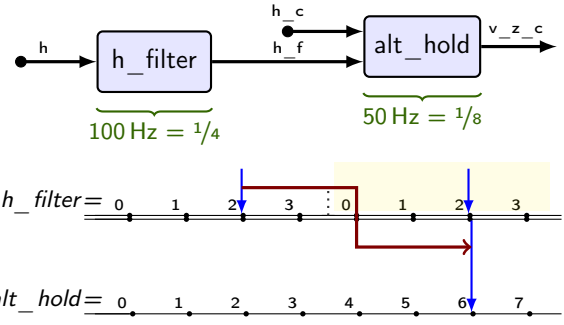


- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

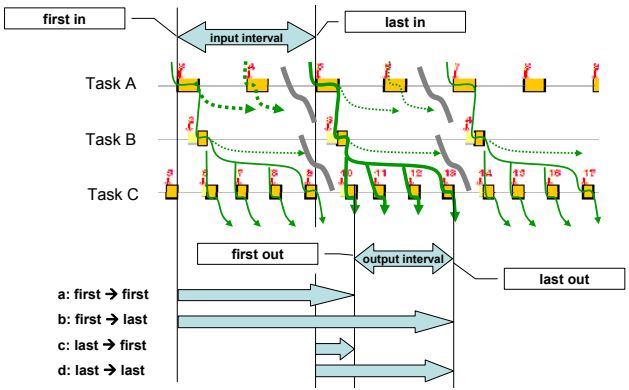
```
static int c_30 = 0;

void step0()
{
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    c_30 = (c_30 + 1) % 8;
}
```

b (concomitance)

```
node assemblage(h_c, va_c : float rate(100, 0))
returns (d_th_c, d_e_c : float)
var vz_c : float;
    d_e, th, h, az, va, q, vz : float;
    vz_f, va_f, h_f, az_f, q_f : float;
let
  (* 200Hz *)
  d_e = elevator(d_e_c *^ 4);
  th = engine(d_th_c *^ 4);
  (va, az, q, vz, h) = dynamics(1.6402 fby th, 0.0186 fby d_e);
  (* 100Hz *)
   h_f =  h_filter( h /^ 2);
  az_f = az_filter(az /^ 2); ...
  (* 50Hz *)
  vz_c = alt_hold(h_c *^ 5, h_f /^ 2);
  d_e_c = vz_control(vz_c, vz_f /^ 2, q_f ^/ 2,
                     az_f /^ 2);
  d_th_c = va_control(va_c *^ 5, va_f /^ 2,
                      q_f /^ 2, vz_f /^ 2);

  (* dynamics -> h_filter -> alt_hold -> vz_control -> elevator <= 2 *)
  (* scheduled as real-time tasks with WCET, precedence, deadlines *)
tel
```

```
node assemblage(h_c, va_c : float :: 1/40 last = 0.)
returns (d_th_c, d_e_c : float :: 1/8 last = (1.6402, 0.0186))
var vz_c : float :: 1/8;
    d_e, th, h, az, va, q, vz : float :: 1/2;
    vz_f, va_f, h_f, az_f, q_f : float :: 1/4;
let
  (* 200Hz = 1/2 *)
  d_e = elevator(current(d_e_c, (? % 4)));
  th = engine(current(d_th_c, (? % 4)));
  (va, az, q, vz, h) = dynamics(th, d_e);
  (* 100Hz = 1/4 *)
   h_f =  h_filter( h when (? % 2));
  az_f = az_filter(az when (? % 2)); ...
  (* 50Hz = 1/8 *)
  vz_c = alt_hold(current(h_c, (? % 5)), h_f when (? % 2));
  d_e_c = vz_control(vz_c, vz_f when (? % 2), q_f when (? % 2),
                     az_f when (? % 2));
  d_th_c = va_control(current(va_c, (? % 5)), va_f when (? % 2),
                      q_f when (? % 2), vz_f when (? % 2));

  latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
  resource balance ops;
tel
```

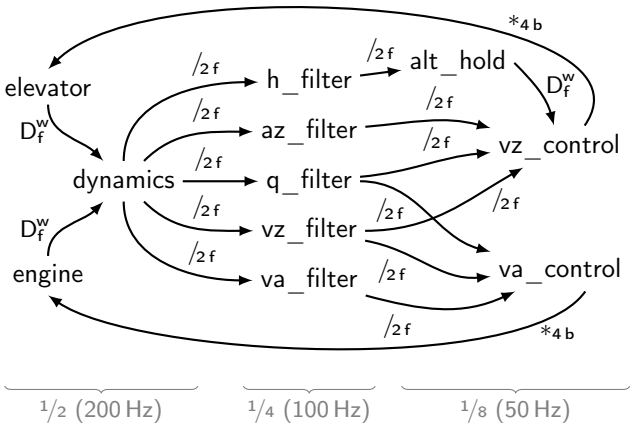
End-to-End Latency



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

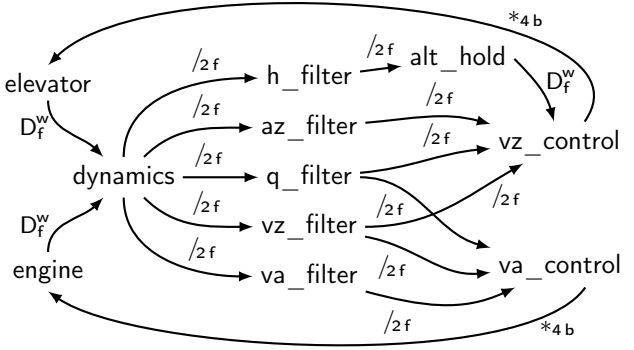
- first-to-first = reaction time = `forward`
- last-to-last = data age = `backward`
- at least one backward path = `exists`
- Lots of other related work
- We ignore execution time and jitter

Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - » rate transitions
  - » concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

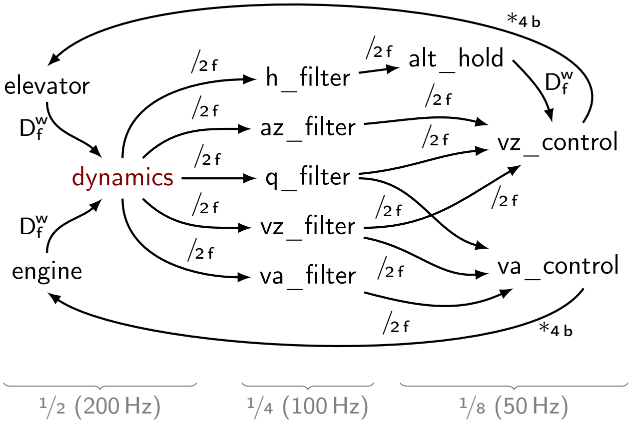
Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - » rate transitions
  - » concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
```

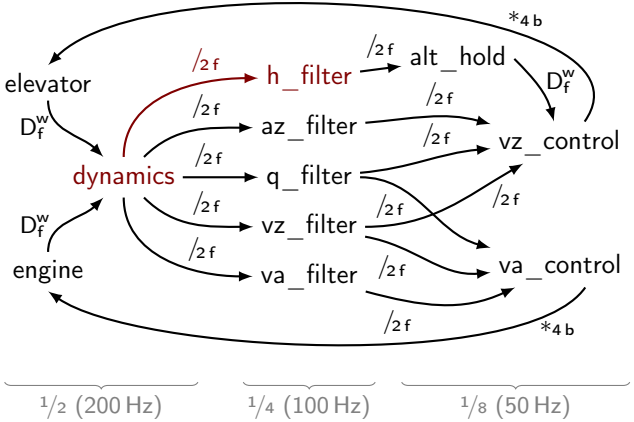
Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - rate transitions
  - concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
```

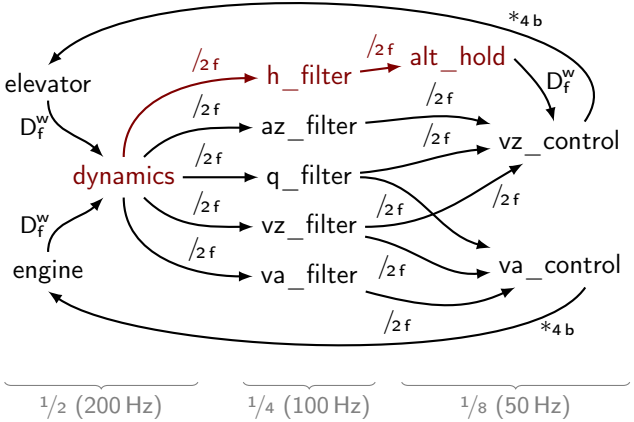
Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - » rate transitions
  - » concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
```

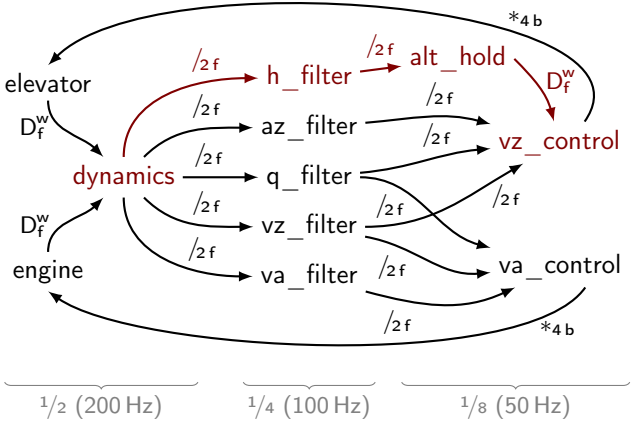
Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - » rate transitions
  - » concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

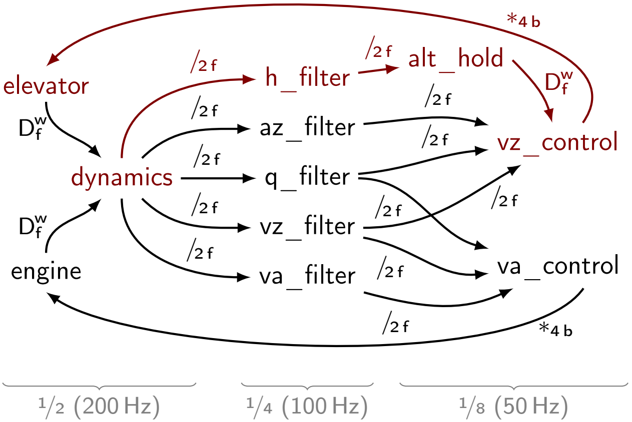
```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
```

Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - » rate transitions
  - » concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

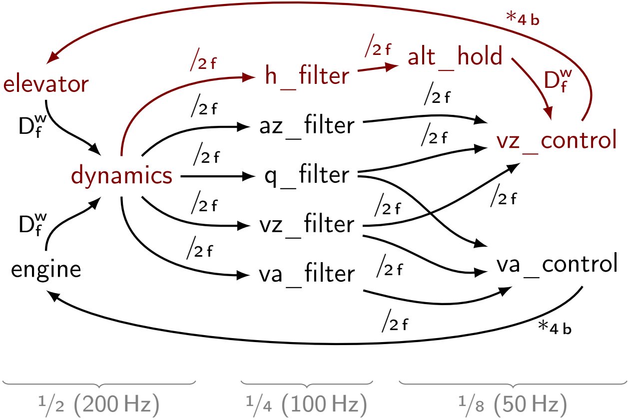
```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
```



- Generate flowgraph from program, annotations:
  - rate transitions
  - concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
```

Flowgraphs and Latency chains



- Generate flowgraph from program, annotations:
  - » rate transitions
  - » concomitance (order within cycle)
- Identify and eliminate cycles
- Transform path into an ILP constraint to constrain the schedule

```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
latency exists <= 2 (d       , h       , a       , v         , e       );
```

Flowgraph links

x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} eq_r$	
`last` x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{b}}} eq_r$	
unconstrained	$eq_w \xrightarrow{\mathsf{D}^{?}_{\mathsf{f}}} eq_r$	
x `when` $(\cdot \,\%\, n)$	$eq_w \xrightarrow{/_n \mathsf{f}} eq_r$	
(`last` x) `when` $(\cdot \,\%\, n)$	$eq_w \xrightarrow{/^{\mathsf{L}}_n \mathsf{b}} eq_r$	
`current`$(x, (\cdot \,\%\, n))$	$eq_w \xrightarrow{*_n \mathsf{f}} eq_r$	(by default)
	$eq_w \xrightarrow{*_n \mathsf{b}} eq_r$	(if 'fast-first')
`current`(`last` $x, (\cdot \,\%\, n)$)	*forbidden*	

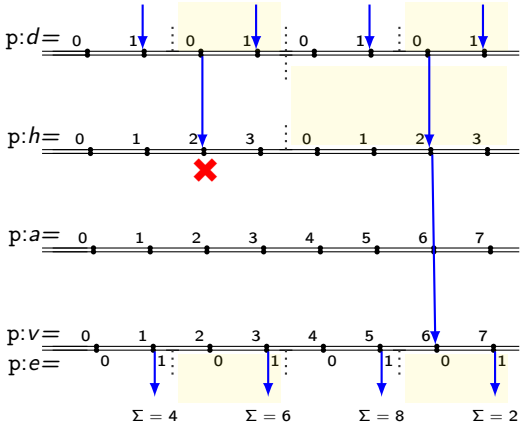
Schedule

	ops	*phase*	
`elevator`	98	$1 \bmod 2$	(p:*e*)
`engine`	82	$0 \bmod 2$	
`dynamics`	1174	$1 \bmod 2$	(p:*d*)
`h_filter`	38	$2 \bmod 4$	(p:*h*)
`az_filter`	37	$2 \bmod 4$	
`q_filter`	37	$2 \bmod 4$	
`vz_filter`	37	$2 \bmod 4$	
`va_filter`	38	$2 \bmod 4$	
`alt_hold`	201	$6 \bmod 8$	(p:*a*)
`vz_control`	88	$6 \bmod 8$	(p:*v*)
`va_control`	90	$2 \bmod 8$	

- Our ROSACE implementation
  - » Cycle period = 2.5 ms (400 Hz)
  - » Allow load balancing of fastest components (200 Hz)
  - » The ops resource estimates the computations required
- Assign each component a phase relative to its period (in terms of base cycles)
- Balance ops per cycle
- Respect end-to-end latency

Schedule

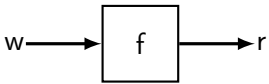
	ops	*phase*	
elevator	98	1 %2	(p:*e*)
engine	82	0 %2	
dynamics	1174	1 %2	(p:*d*)
h_filter	38	2 %4	(p:*h*)
az_filter	37	2 %4	
q_filter	37	2 %4	
vz_filter	37	2 %4	
va_filter	38	2 %4	
alt_hold	201	6 %8	(p:*a*)
vz_control	88	6 %8	(p:*v*)
va_control	90	2 %8	



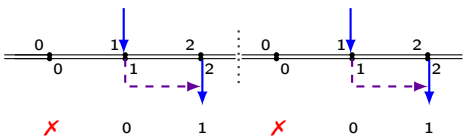

```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
latency exists <= 2 (d,        h,        a,        v,          e);
```

Direct Communications

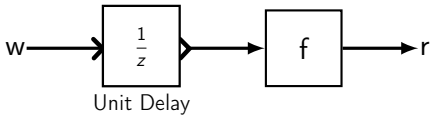
```
r = f(w)
```



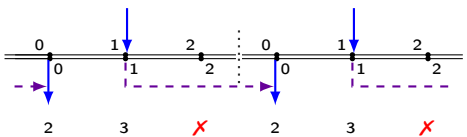
- D_f^w: Direct Write-before-read
  (forward concomitance)
- Dependency constraint: p:$w \leq$ p:r
- $0 \leq$ p:$r -$ p:$w < period$




```
r = f(last w)
```

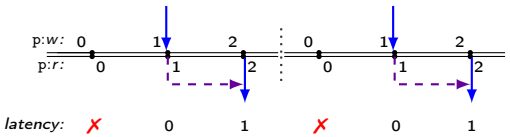


- D_b^r: Direct Read-before-write
  (backward concomitance)
- Dependency constraint: p:$r \leq$ p:w
- $0 <$ p:$r -$ p:$w + period \leq period$



Minimum Pairwise Latency: same period

Direct communication: `r = w`



$\mathtt{r}, \mathtt{w} :: 1/3$

$\mathrm{p}{:}w = 1$

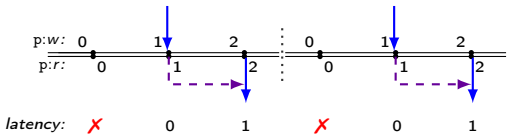
$\mathrm{p}{:}r = 2$

$lat. = 1$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{w}}_{\mathrm{f}}} eq_r$
- Write-before-read: $\mathrm{p}{:}w \leq \mathrm{p}{:}r$
- $0 \leq \mathrm{p}{:}r - \mathrm{p}{:}w < period$

Minimum Pairwise Latency: same period

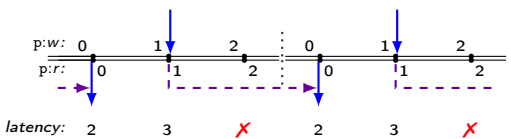
Direct communication: `r = w`



$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 2$
$lat. = 1$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{w}}_{\mathrm{f}}} eq_r$
- Write-before-read: $\mathrm{p}{:}w \leq \mathrm{p}{:}r$
- $0 \leq \mathrm{p}{:}r - \mathrm{p}{:}w < period$

Delayed communication: `r = last w`

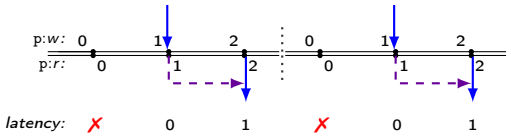


$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 0$
$lat. = 2$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{r}}_{\mathrm{b}}} eq_r$
- Read-before-write: $\mathrm{p}{:}r \leq \mathrm{p}{:}w$
- $0 < \mathrm{p}{:}r - \mathrm{p}{:}w + period \leq period$

Minimum Pairwise Latency: same period

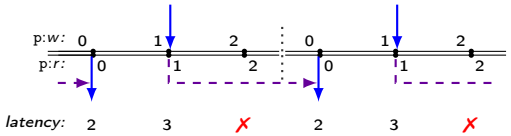
Direct communication: `r = w`



$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 2$
$lat. = 1$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{w}}_{\mathrm{f}}} eq_r$
- Write-before-read: $\mathrm{p}{:}w \leq \mathrm{p}{:}r$
- $0 \leq \mathrm{p}{:}r - \mathrm{p}{:}w < period$

Delayed communication: `r = last w`



$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 0$
$lat. = 2$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{r}}_{\mathrm{b}}} eq_r$
- Read-before-write: $\mathrm{p}{:}r \leq \mathrm{p}{:}w$
- $0 < \mathrm{p}{:}r - \mathrm{p}{:}w + period \leq period$

Unconstrained communication (`r = last? w`)

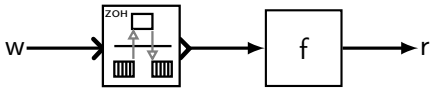
- $eq_w \xrightarrow{\mathrm{D}^{?}_{\mathrm{f}}} eq_r$: $(\mathrm{p}{:}r - \mathrm{p}{:}w + \mathrm{period}(w)) \bmod \mathrm{period}(w)$
- $eq_w \xrightarrow{\mathrm{D}^{?}_{\mathrm{b}}} eq_r$: $((\mathrm{p}{:}r - \mathrm{p}{:}w + \mathrm{period}(w) - 1) \bmod \mathrm{period}(w)) + 1$

Rate Transitions

`r = f(w when (1 % 3))`

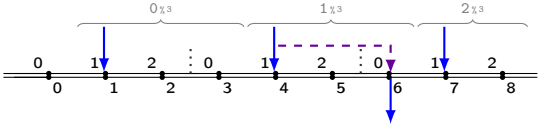
`(i % n)`: take value `i` of every `n`

`(? % n)`: take any of every `n` values



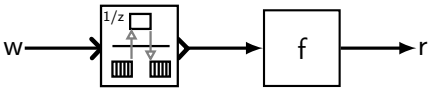
Fast-to-Slow Transition

- $/_{nf}$: Fast-to-slow
  (forward concomitance)



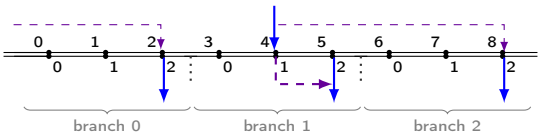
`r = f(current(w, (1 % 3)))`

`(i % n)`: `i` initial values, then repeat `n` times



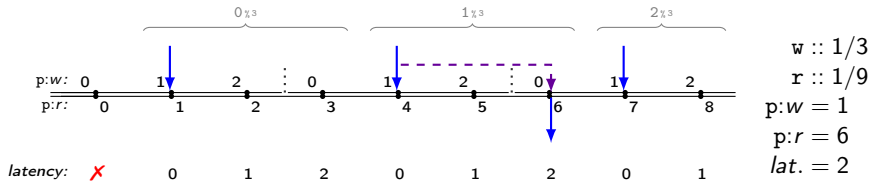
Slow-to-Fast Transition

- $*_{nf} \mid *_{nb}$: Slow-to-fast
  (forward or backward concomitance)



Minimum Pairwise Latency: fast-to-slow

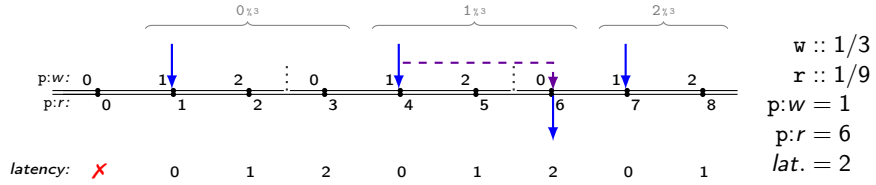
`r = w when (? % 3)` $\quad eq_w \xrightarrow{/_{nf}} eq_r \quad$ $(\mathrm{p}{:}r - \mathrm{p}{:}w) \bmod \mathrm{period}(w)$



`w` :: $1/3$
`r` :: $1/9$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 6$
$lat. = 2$

Minimum Pairwise Latency: fast-to-slow

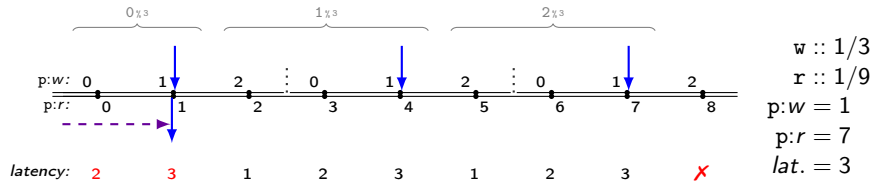
`r = w when (? % 3)` $eq_w \xrightarrow{/nf} eq_r$ $(\mathrm{p}{:}r - \mathrm{p}{:}w) \bmod \mathrm{period}(w)$



`w` :: 1/3
`r` :: 1/9
p:$w = 1$
p:$r = 6$
lat. $= 2$

`r = (last e) when (0 % 3)` $eq_w \xrightarrow{/nb} eq_r$

$((\mathrm{period}(w) + \mathrm{p}{:}r - \mathrm{p}{:}w - 1) \bmod \mathrm{period}(w)) + 1$

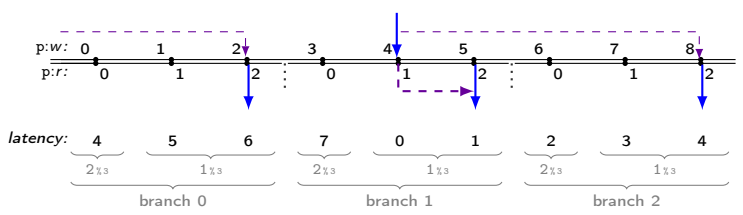


`w` :: 1/3
`r` :: 1/9
p:$w = 1$
p:$r = 7$
lat. $= 3$

Minimum Pairwise Latency: slow-to-fast

`r = current(w, (1 % 3))` $eq_w \xrightarrow{*_{nf}} eq_r$

$((branch \cdot \mathrm{period}(r) + \mathrm{period}(w) + \mathrm{p}{:}r - \mathrm{p}{:}w - 1) \bmod \mathrm{period}(w)) + 1$

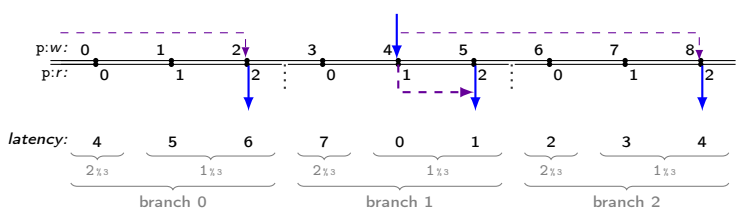


`w` :: 1/9
`r` :: 1/3
p:$w = 4$
p:$r = 2$
$lat. = 1, 4$

Minimum Pairwise Latency: slow-to-fast

`r = current(w, (1 % 3))` $\quad eq_w \xrightarrow{*_{nf}} eq_r$

$((branch \cdot \mathrm{period}(r) + \mathrm{period}(w) + \mathrm{p}{:}r - \mathrm{p}{:}w - 1) \bmod \mathrm{period}(w)) + 1$



`w` :: 1/9
`r` :: 1/3
p:$w = 4$
p:$r = 2$
$lat. = 1, 4$

`r = current(last w, (? % 3))`?

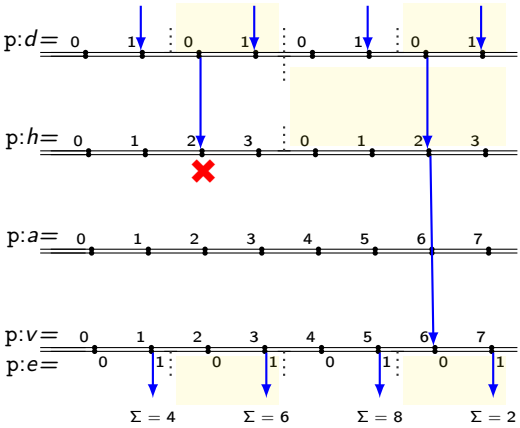
- Not allowed. Not enough 'memories'.
- Must be normalized to

```
r = current(t, (? % 3));
t = last w;
```

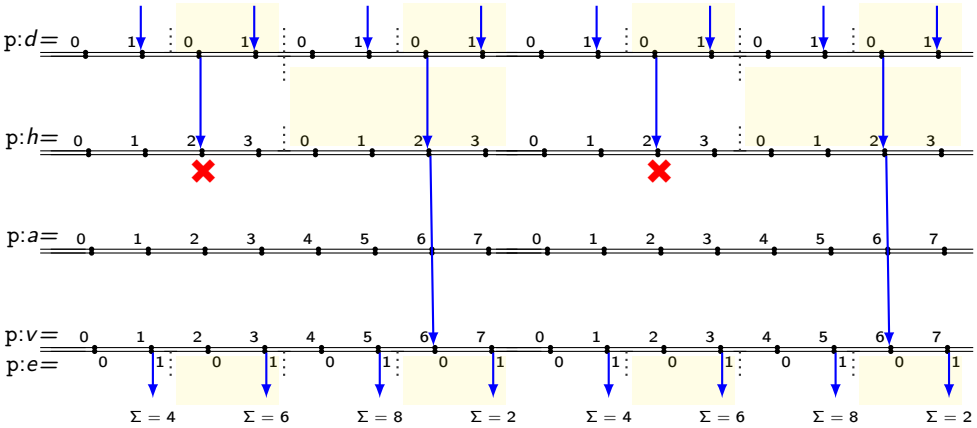
End-to-End Latency: the wrong way

- Define pairwise latencies for each link type.
- Chain them together into a sequence.
- Difficult to handle branching and dead ends.
- Difficult to explain.
- Complicated formulas.
- There's a better way...

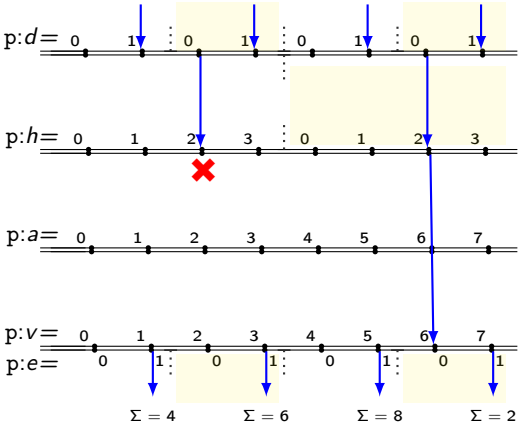
Constraining End-to-end Latency



Constraining End-to-end Latency

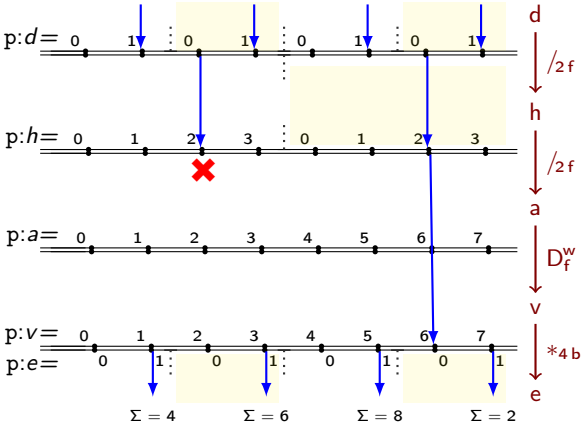


Constraining End-to-end Latency



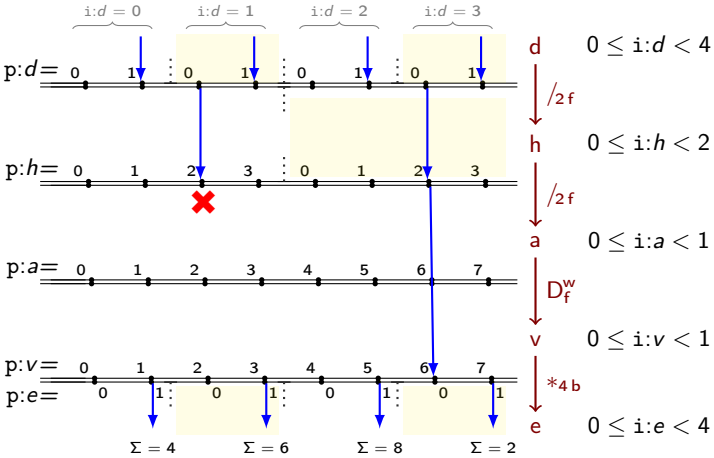
(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency



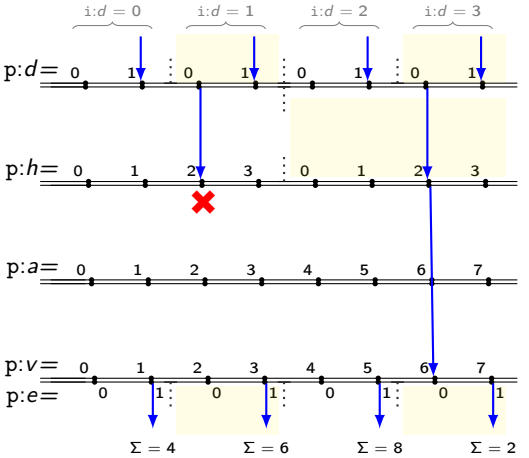
(View online at https://www.tbrk.org/dataflow/showlatency)

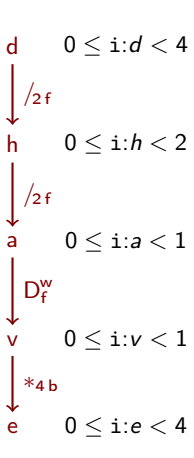
Constraining End-to-end Latency



(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

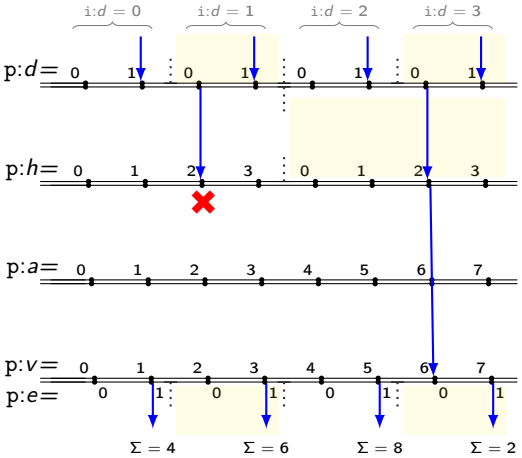
$0 \le \mathrm{i}{:}v < 1$

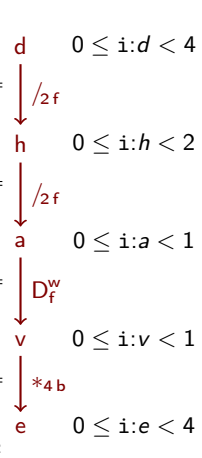
$0 \le \mathrm{i}{:}e < 4$

$$0 \le \mathtt{lat}_{d,h} < 2$$
$$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

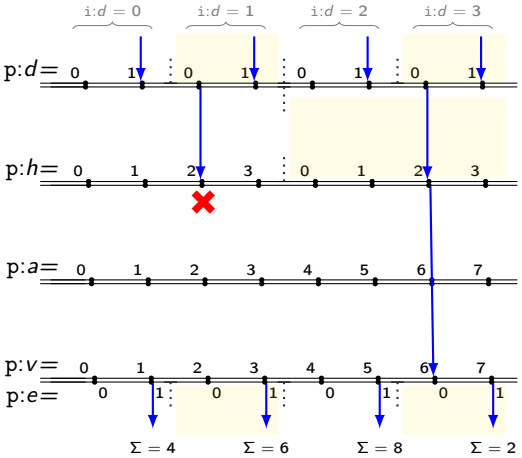
$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

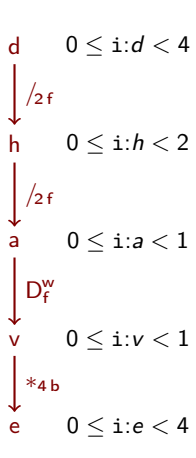
$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

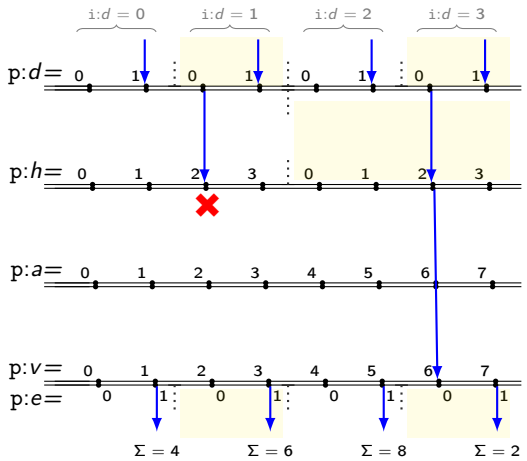
$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

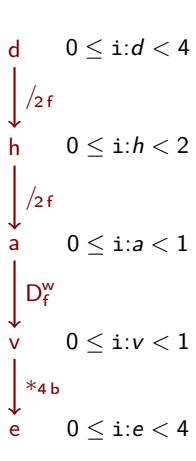
$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$$0 \le \mathrm{i}{:}d < 4$$

$$0 \le \mathrm{lat}_{d,h} < 2$$
$$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathrm{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$$

$$0 \le \mathrm{i}{:}h < 2$$

$$0 \le \mathrm{lat}_{h,a} < 4$$
$$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathrm{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$$

$$0 \le \mathrm{i}{:}a < 1$$

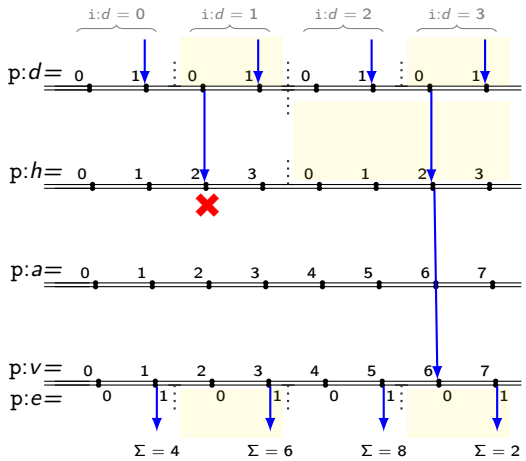
$$0 \le \mathrm{lat}_{a,v} < 8$$
$$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathrm{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$$

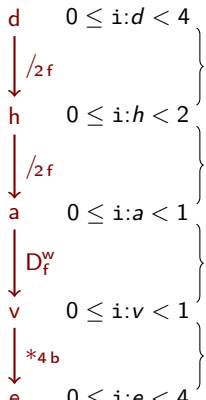
$$0 \le \mathrm{i}{:}v < 1$$

$$0 \le \mathrm{i}{:}e < 4$$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

$0 \le \mathtt{lat}_{a,v} < 8$

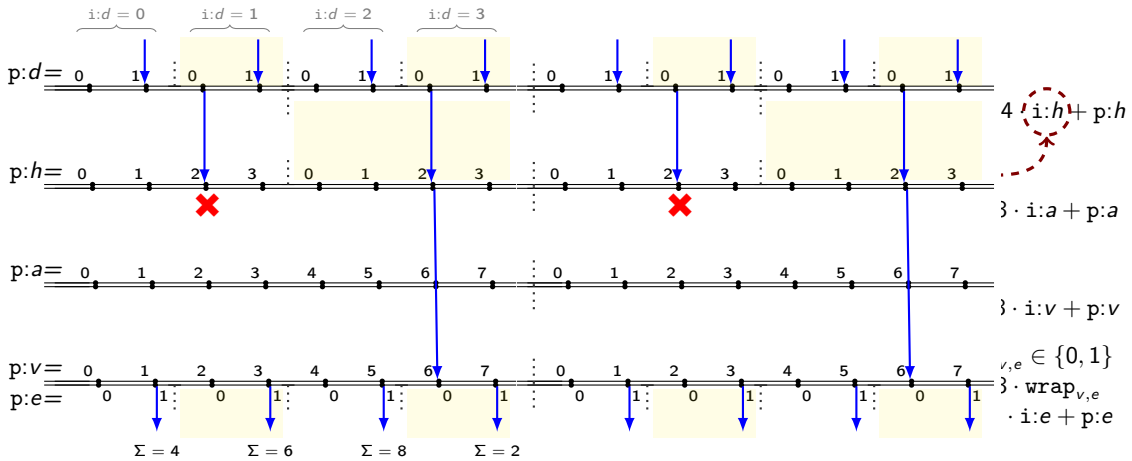
$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathtt{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$

$0 < \mathtt{lat}_{v,e} \le 8 \qquad \mathtt{wrap}_{v,e} \in \{0, 1\}$

$8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v + \mathtt{lat}_{v,e} - 8 \cdot \mathtt{wrap}_{v,e} = 2 \cdot \mathrm{i}{:}e + \mathrm{p}{:}e$

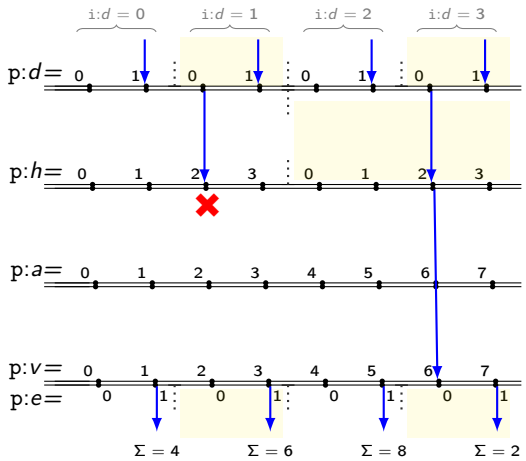
(View online at https://www.tbrk.org/dataflow/showlatency)

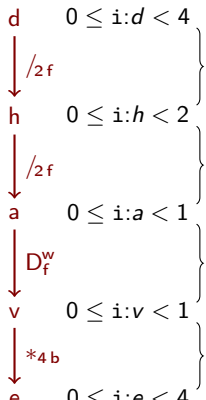
Constraining End-to-end Latency



(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

$0 \le \mathtt{lat}_{a,v} < 8$

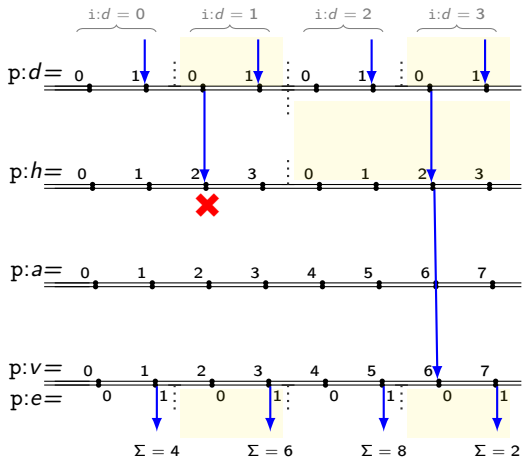
$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathtt{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$

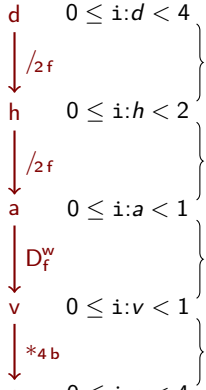
$0 < \mathtt{lat}_{v,e} \le 8 \qquad \mathtt{wrap}_{v,e} \in \{0, 1\}$

$8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v + \mathtt{lat}_{v,e} - 8 \cdot \mathtt{wrap}_{v,e} = 2 \cdot \mathrm{i}{:}e + \mathrm{p}{:}e$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

$0 \le \mathtt{lat}_{a,v} < 8$

$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathtt{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$

$0 < \mathtt{lat}_{v,e} \le 8 \qquad \mathtt{wrap}_{v,e} \in \{0, 1\}$

$8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v + \mathtt{lat}_{v,e} - 8 \cdot \mathtt{wrap}_{v,e} = 2 \cdot \mathrm{i}{:}e + \mathrm{p}{:}e$

$\mathtt{lat}_{d,h} + \mathtt{lat}_{h,a} + \mathtt{lat}_{a,v} + \mathtt{lat}_{v,e} \le 2$

(View online at https://www.tbrk.org/dataflow/showlatency)

Chains of constraints

`latency forward/backward` $\leq B\ (\ldots, w, r, \ldots)$

Chains of constraints

`latency forward/backward` $\leq B\ (\ldots, w, r, \ldots)$

For each link $w \xrightarrow{s,c} r$,

Chains of constraints

`latency` `forward`/`backward` $\leq B\ (\ldots, w, r, \ldots)$

For each link $w \xrightarrow{s,c} r$,

$$0 \leq \mathtt{i}{:}r < {}^{hp}/_{\mathrm{period}(r)}$$

$$\begin{array}{ll} 0 \leq \mathtt{lat}_{w,r} < L & \text{for } c = \mathsf{f} \\ 0 < \mathtt{lat}_{w,r} \leq L & \text{for } c = \mathsf{b} \end{array} \quad \left\{ \begin{array}{ll} \text{where} & L = \mathrm{period}(r) \text{ if } \texttt{forward} \\ \text{and} & L = \mathrm{period}(w) \text{ if } \texttt{backward} \end{array} \right.$$

$$0 \leq \mathtt{wrap}_{w,r} \leq 1 \qquad \begin{array}{l} \text{if } \texttt{forward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, *_n\} \\ \text{or if } \texttt{backward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, /_n\} \end{array}$$

$$\mathrm{period}(w) \cdot \mathtt{i}{:}w + \mathtt{p}{:}w + \mathtt{lat}_{w,r} - hp \cdot \mathtt{wrap}_{w,r} = \mathrm{period}(r) \cdot \mathtt{i}{:}r + \mathtt{p}{:}r.$$

- Each intermediate instance is both a reader and a writer, and thus constrained by two equations.
- `forward`: attach chains from the top
- `backward`: attach chains to the bottom

Manipulating flow graphs

Before scheduling

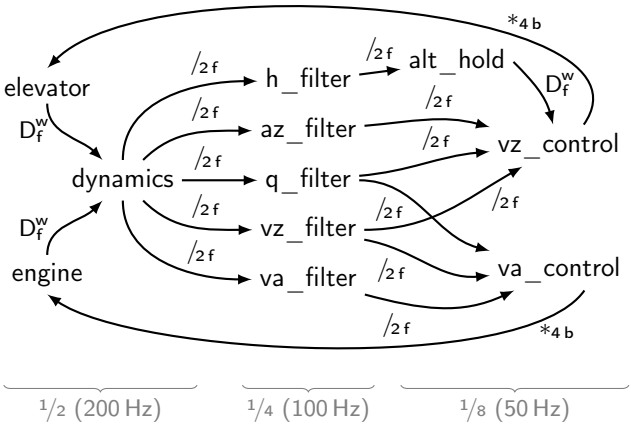
1. Construct flow graph from program
2. Remove potential cycles by flipping the microcausality
3. Use to generate ILP constraints (causality, end-to-end latency)

After scheduling: convert to dependency graph

1. Drop edges between equations that cannot execute in any phase.
2. Flip the cobackward (b) edges.
3. Use with standard algorithm to schedule equations within a step function.

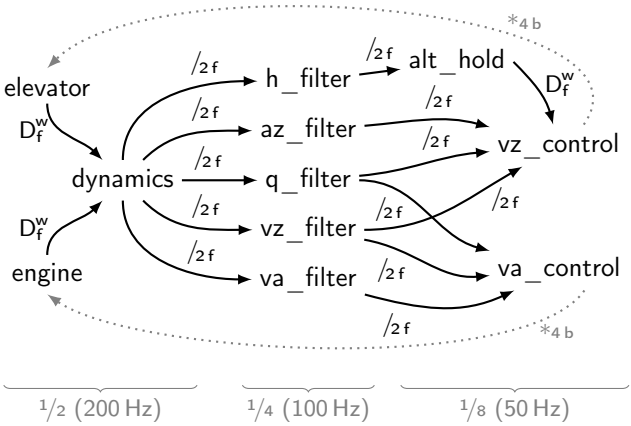
ROSACE example: flow graph → dependency graph

	ops	*phase*
`elevator`	98	$1_{\%2}$
`engine`	82	$0_{\%2}$
`dynamics`	1174	$1_{\%2}$
`h_filter`	38	$2_{\%4}$
`az_filter`	37	$2_{\%4}$
`q_filter`	37	$2_{\%4}$
`vz_filter`	37	$2_{\%4}$
`va_filter`	38	$2_{\%4}$
`alt_hold`	201	$6_{\%8}$
`vz_control`	88	$6_{\%8}$
`va_control`	90	$2_{\%8}$



ROSACE example: flow graph → dependency graph

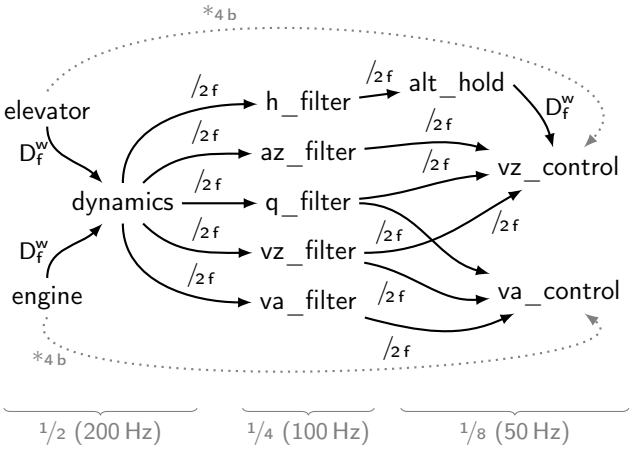
	ops	*phase*
`elevator`	98	$1_{\%2}$
`engine`	82	$0_{\%2}$
`dynamics`	1174	$1_{\%2}$
`h_filter`	38	$2_{\%4}$
`az_filter`	37	$2_{\%4}$
`q_filter`	37	$2_{\%4}$
`vz_filter`	37	$2_{\%4}$
`va_filter`	38	$2_{\%4}$
`alt_hold`	201	$6_{\%8}$
`vz_control`	88	$6_{\%8}$
`va_control`	90	$2_{\%8}$



- Remove all edges between equations that can never execute in the same phase.
- Reverse edges whose microcausality is *b*.

ROSACE example: flow graph → dependency graph

	ops	*phase*
`elevator`	98	$1_{\%2}$
`engine`	82	$0_{\%2}$
`dynamics`	1174	$1_{\%2}$
`h_filter`	38	$2_{\%4}$
`az_filter`	37	$2_{\%4}$
`q_filter`	37	$2_{\%4}$
`vz_filter`	37	$2_{\%4}$
`va_filter`	38	$2_{\%4}$
`alt_hold`	201	$6_{\%8}$
`vz_control`	88	$6_{\%8}$
`va_control`	90	$2_{\%8}$



- Remove all edges between equations that can never execute in the same phase.
- Reverse edges whose microcausality is b.

ROSACE example: generated code

```
static int c = 0;
static float h_c = 0, d_th_c = 1.6402, d_e_c = 0.0186, ...;
static float vz_c, ..., q_f;

void step0()
{
  if (c % 2 == 0) {
    engine();
    if (c % 4 == 2) {
      vz_filter(); h_filter(); va_filter(); q_filter(); az_filter();
    }
  } else {
    elevator(); dynamics();
  }
  switch (c) {
  case 2: va_control(); break;
  case 6: alt_hold(); vz_control(); break;
  }
  c = (c + 1) % 8;
}
```

Code generation

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.

Code generation: 2

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 }
```

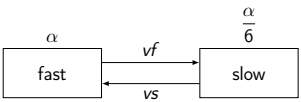
```
if (state(c_3) mod 24 = 7) {
    state(x) := read_real()
} else {
    y := read_real()
}
```

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

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

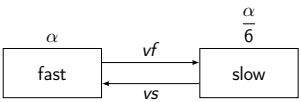
Causality, Scheduling, and Semantics

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

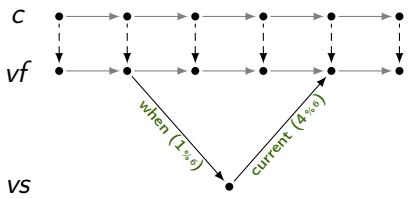


Causality, Scheduling, and Semantics

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

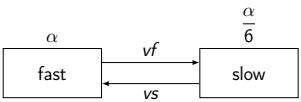


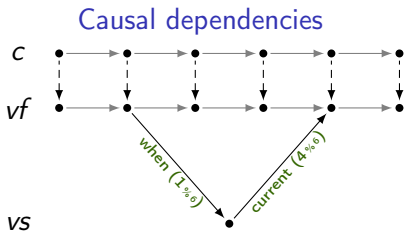
Causal dependencies



Causality, Scheduling, and Semantics

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

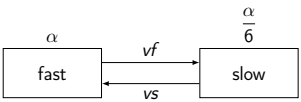




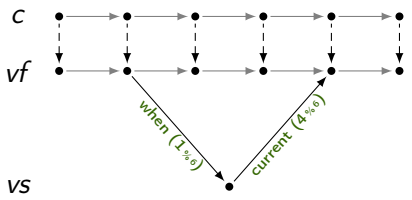
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Causality, Scheduling, and Semantics

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



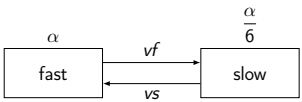
Causal dependencies



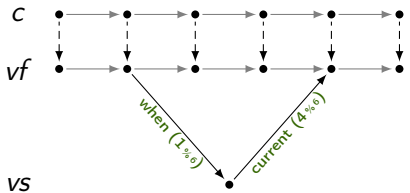
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Causality, Scheduling, and Semantics

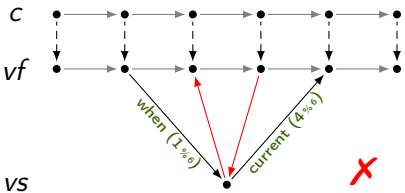
```
c = last c + 1;                  -- last c = -1
vf = current(vs, (4 % 6)) + c;
vs = vf when (1 % 6) + 5;        -- last vs = 0
```



Causal dependencies



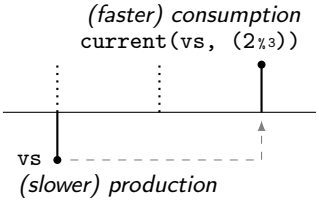
Scheduling dependencies



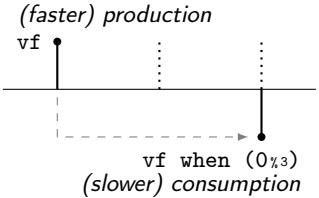
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Adding equations to relax constraints/add buffering

Hold slow around fast reads

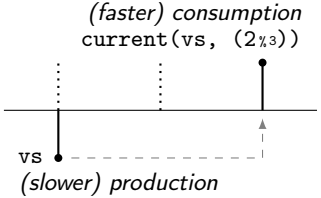


Hold fast around fast writes

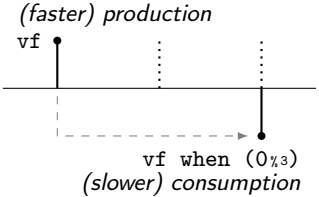


Adding equations to relax constraints/add buffering

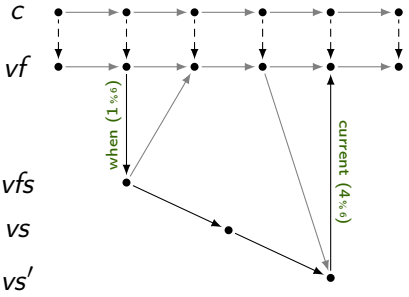
Hold slow around fast reads



Hold fast around fast writes




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



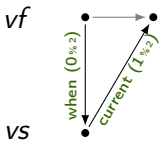
Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

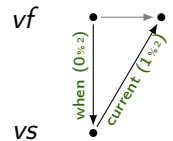
```
vf = current(vs, (1%2));
vs = vf when (0%2) + 1;
```



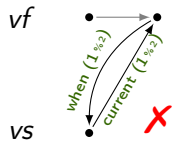
Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

```
vf = current(vs, (1%2));
vs = vf when (0%2) + 1;
```



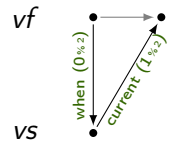

```
vf = current(vs, (1%2));
vs = vf when (1%2) + 1;
```



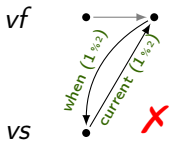
Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

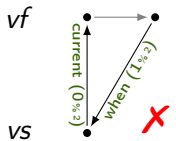
```
vf = current(vs, (1%2));
vs = vf when (0%2) + 1;
```




```
vf = current(vs, (1%2));
vs = vf when (1%2) + 1;
```



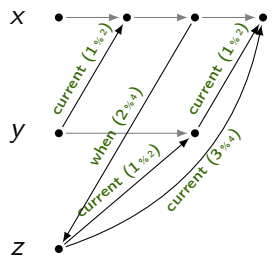

```
vf = current(vs, (0%2));
vs = vf when (1%2) + 1;
```



Causality: 2

Causality relations between `x` :: α, `y` :: $\frac{\alpha}{2}$, and `z` :: $\frac{\alpha}{4}$.

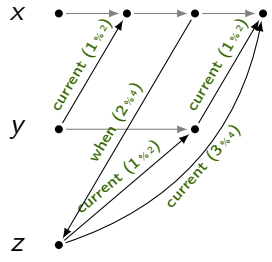
```
x = current(y, (1 % 2))
     + current(z, (3 % 4)) + 2;

y = current(z, (1 % 2)) + 20;

z = x when (2 % 4) + 200;
```



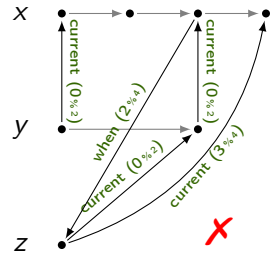
Causality: 2

Causality relations between `x :: α`, `y :: α/2`, and `z :: α/4`.

```
x = current(y, (1 % 2))
     + current(z, (3 % 4)) + 2;

y = current(z, (1 % 2)) + 20;

z = x when (2 % 4) + 200;
```

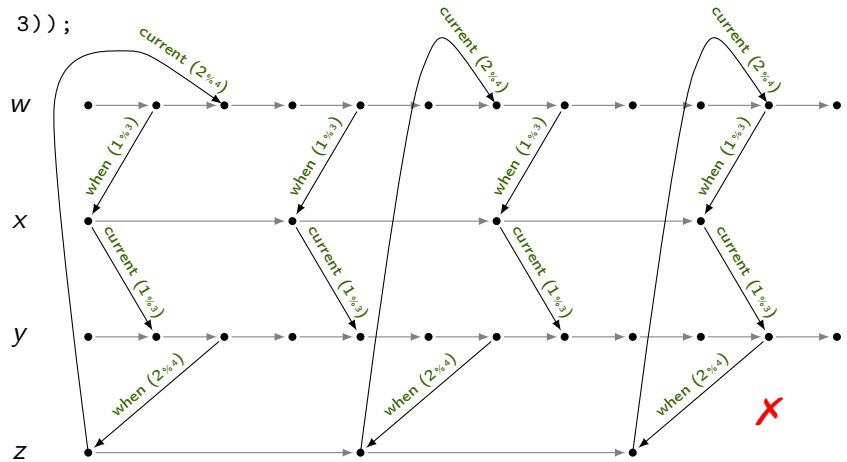



```
x = current(y, (0 % 2))
     + current(z, (3 % 4)) + 2;

y = current(z, (1 % 2)) + 20;

z = x when (2 % 4) + 200;
```



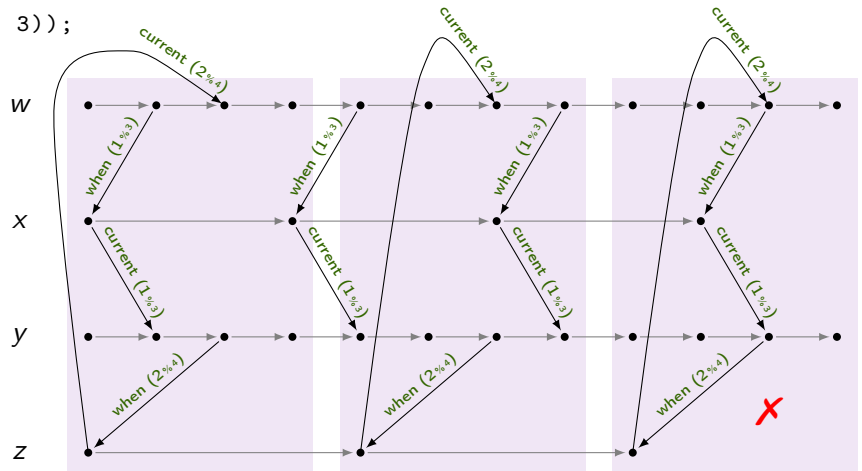
Causality: non-harmonic rates with ‘shifting’

```
w = current(z, (2 % 4));
x = w when (1 % 3);
y = current(x, (1 % 3));
z = y when (2 % 4);
```



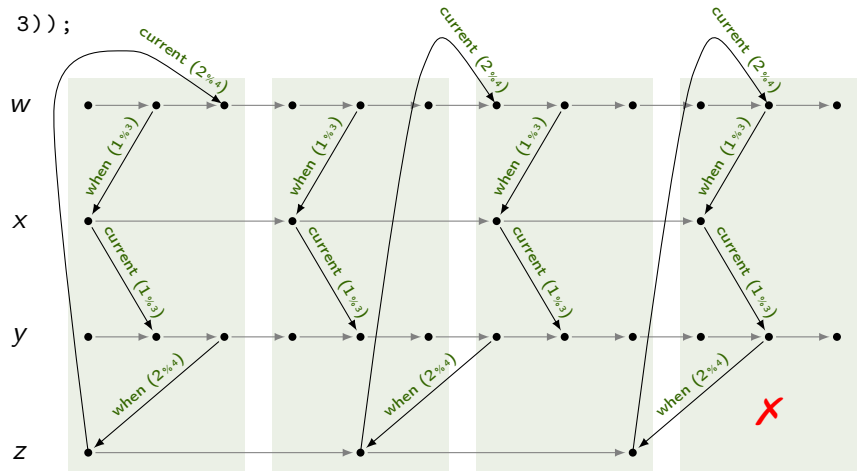
Causality: non-harmonic rates with ‘shifting’

```
w = current(z, (2 % 4));
x = w when (1 % 3);
y = current(x, (1 % 3));
z = y when (2 % 4);
```



Causality: non-harmonic rates with ‘shifting’

```
w = current(z, (2 % 4));
x = w when (1 % 3);
y = current(x, (1 % 3));
z = y when (2 % 4);
```



Related Work: Lucy-n

- Model and language [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems] [Mandel, Plateau, and Pouzet (2010): Lucy-n: a n-Synchronous extension of Lustre]
- Flexible scheduling patterns ($0010(010)$) and buffering
- Notion of jitter with clock envelopes [Cohen, Mandel, Plateau, and Pouzet (2008): Abstraction of Clocks in Synchronous Data-flow Systems]
- Sophisticated type-based analysis for causality and buffer sizes
- Less focus on code generation

Our work

- Less flexible scheduling
- Buffering is implicit and limited to size ≤ 1
- Less clock typing, more causality
- Generate imperative code

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: $[phase, period]$
  $(0^k 10^{n-k-1}$ 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) `whens` and `merges`

Our work

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

Conclusion

- Programming language for composing tasks
  - » Particularity: tasks must terminate in one cycle
  - » Semantics, static analysis (clock types), compilation
- Use an ILP solver for scheduling
  - » Load balancing
  - » End-to-end latency
- Prototype compiler in OCaml
  with ILP scheduling and basic code generation
- Tested on Airbus example with 5000 nodes
  (compiles in approx. 45 minutes).

M2 internship P. Robert

- Clock inference
- Inlining
- Normalization
- Introduction of 'buffers' by inserting assignments.

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.
- Bourke, T., V. Bregeon, and M. Pouzet (July 2023). "Scheduling and Compiling Rate-Synchronous Programs with End-to-End Latency Constraints". In: *35th Euromicro Conf. on Real-Time Systems (ECRTS 2023)*. Ed. by A. V. Papadopoulos. Vol. 262. Leibniz Int. Proc. in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 1:1–1:22.
- 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.
- Cohen, A., L. Mandel, F. Plateau, and M. Pouzet (Dec. 2008). "Abstraction of Clocks in Synchronous Data-flow Systems". In: *Proc. 6th Asian Symp. Programming Languages and Systems (APLAS 2008)*. Ed. by G. Ramalingam. Vol. 5356. LNCS. Bangalore, India: Springer, pp. 237–254.

References II

- Colaço, J.-L., B. Pagano, and M. Pouzet (Sept. 2017). "Scade 6: A Formal Language for Embedded Critical Software Development". In: *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*. Nice, France: IEEE Computer Society, pp. 4–15.
- 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.
- Forget, J., F. Boniol, D. Lesens, and C. Pagetti (Dec. 2008). "A Multi-Periodic Synchronous Data-Flow Language". In: *Proc. 11th IEEE High Assurance Systems Engineering Symposium (HASE 2008)*. Nanjing, China: IEEE, pp. 251–260.
- — (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.

References III

- Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud (Sept. 1991). "The synchronous dataflow programming language LUSTRE". In: *Proc. IEEE* 79.9, pp. 1305–1320.
- 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". preprint.
- 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.
- Pagetti, C., D. Saussié, R. Gratia, E. Noulard, and P. Siron (Apr. 2014). "The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution". In: *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014)*. IEEE. Berlin, Germany, pp. 309–318.

References IV

- 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.
- Wyss, R., F. Boniol, J. Forget, and C. Pagetti (Dec. 2012). "A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming". In: *Proc. 10th Asian Symp. Programming Languages and Systems (APLAS 2012)*. Ed. by R. Jhala and A. Igarashi. Vol. 7705. LNCS. Kyoto, Japan: Springer, pp. 223–238.