Logical time and real-time in the Synchronous approach

Julien Forget
LIFL - Université Lille 1
Overview

At the end of this session you should understand:

- Why introducing explicit real-time constraints in a synchronous language is useful;
- How we can deal with both logical-time and real-time;
- The implications of the introduction of real-time in the language structure and compilation.
Outline

1. Real-time
2. Multi-rate system design
3. Synchronous real-time
   - Arithmetic clocks
   - Multi-threaded execution
4. Prelude
   - The language
   - Compilation
5. Conclusion
Reactive system (reminder)

- React to inputs:
  1. Acquire inputs on sensors;
  2. Compute;
  3. Produce values on actuators.

- Actions impact the environment, thus subsequent inputs;

- Response time must be **bounded**, due to environment evolving autonomously.
Real-time system

Definition
Real-time systems must guarantee response within strict time constraints, often referred to as “deadlines”. (Wikipedia)

- Similar to reactive systems;
- Several, predefined time bounds.
Example: UAV control

Real-time constraints:

- GPS (input): 1 frame every 250 ms.
  - Deadline miss \(\Rightarrow\) frame lost (current position), wrong trajectory.
- Attitude regulation (output): consolidate actuator orders every 60ms
  - Deadline miss \(\Rightarrow\) loss of control.
- Failure detection (internal): check inconsistencies every 200ms
  - Deadline miss \(\Rightarrow\) crash with motors on.
Program=a set of tasks (threads) $\tau_i$:

- $T_i$: period;
- $D_i$: relative deadline ($D_i \leq T_i$);
- $C_i$: worst-case execution time (WCET);
- $O_i$: initial release date;
- $\tau_{i,p}$: $p^{th}$ job of $\tau_i$. 

Logical time and real-time in the Synchronous approach
Deadlines and periods

- **Deadline**: respond before some specified time;
- **Period**: processes are recurring at regular time intervals;
- The period is often an implicit deadline (non-reentrant tasks);
- Choice of the periods/deadlines:
  - Lower-bound: physical constraints of the sensors/actuators;
  - Lower-bound: computation time;
  - Upper-bound: too slow can lead to an unsteady system.
Execution times

- Evaluating the execution time of some process is HARD
  - Depends on the content of the memory;
  - Depends on the content of the pipeline;
  - Depends on the values processed;
  - Other processes may interfere;
  - OS may interfere...

- Validating temporal behaviour with variable execution times is complex;

⇒ Execution times are (largely) over-evaluated by a **Worst-Case Execution Time (WCET)**.
Real-time multi-tasking

Some classic problems:

- **Scheduling policy**: define an algorithm that finds an execution order (a schedule), that respects all deadlines;

- **Schedulability analysis**: ensure before execution that deadlines can and will be met (for a given policy);

- Data-dependencies $\Rightarrow$ scheduling policy for dependent tasks + synchronization primitives (e.g. semaphores, buffers, . . .);

- Shared resources $\Rightarrow$ problems similar to communication synchronizations.
Scheduling: multi-processor example

\[ \tau_B(T_B = 9, C_B = 5) \text{ and } \tau_A(T_A = 3, C_A = 1): \]

\[ \begin{array}{cccc}
0 & 3 & 6 & 9 \\
A & A & A & \\
0 & 3 & 6 & 9
\end{array} \]
Scheduling: mono-processor example

\( \tau_B(T_B = 9, C_B = 5) \) and \( \tau_A(T_A = 3, C_A = 1) \):

- Without preemption:

  
  Deadline miss

  \[
  \begin{array}{c|c|c|c}
  \text{A} & \text{B} & \text{A} \\
  0 & 3 & 6 & 9
  \end{array}
  \]

- With preemption:

  \[
  \begin{array}{c|c|c|c}
  \text{A} & \text{B} & \text{A} & \text{B} & \text{A} \\
  0 & 3 & 6 & 9
  \end{array}
  \]
Fixed-task priorities: a fixed priority is assigned to each task;
- Task with smaller relative deadline (period) gets a higher priority;
- Works only when \( D_i = T_i \);
- This policy is **optimal** among the fixed-task priority policies.
Fixed-task priorities: a fixed priority is assigned to each task;

- Task with smaller relative deadline (=period) gets a higher priority;

- Works only when $D_i = T_i$;

- This policy is **optimal** among the fixed-task priority policies.

⇒ What does **optimal** mean?
Rate-Monotonic analysis

**Sufficient** schedulability test:

\[
\sum_{i=0}^{m} \frac{C_i}{T_i} \leq m(2^{1/m} - 1)
\]

\(\simeq 0.8\) for \(m = 2\) and tends towards 0.7 for big \(m\).
**Rate-Monotonic analysis**

**Sufficient** schedulability test:

\[
\sum_{i=0}^{m} \frac{C_i}{T_i} \leq m(2^{1/m} - 1)
\]

\(\simeq 0.8\) for \(m = 2\) and tends towards 0.7 for big \(m\).

\(\Rightarrow\) What does **sufficient** mean?
Rate-Monotonic analysis

**Sufficient** schedulability test:

\[ \sum_{i=0}^{m} \frac{C_i}{T_i} \leq m(2^{1/m} - 1) \]

\( \simeq 0.8 \) for \( m = 2 \) and tends towards 0.7 for big \( m \).

⇒ What does **sufficient** mean?

**NB**: More general cases \( D_i \leq T_i \), multi-core, ...) are in many cases NP.
Okay...
Okay...

But, we were told to ignore real-time!

(cf...
Yet, knowing real-time constraints is useful

Based on real-time constraints we can:
Yet, knowing real-time constraints is useful

Based on real-time constraints we can:

- Schedule better:
  - Optimize processor utilization (do not execute tasks more frequently than required);
  - Ensure temporal correction by assigning priorities based on deadlines.
Yet, knowing real-time constraints is useful

Based on real-time constraints we can:

- Schedule better:
  - Optimize processor utilization (do not execute tasks more frequently than required);
  - Ensure temporal correction by assigning priorities based on deadlines.
- Statically analyze the real-time behaviour: check before execution that the system will not become overloaded/late;
Yet, knowing real-time constraints is useful

Based on real-time constraints we can:

- Schedule better:
  - Optimize processor utilization (do not execute tasks more frequently than required);
  - Ensure temporal correction by assigning priorities based on deadlines.

- Statically analyze the real-time behaviour: check before execution that the system will not become overloaded/late;

- As a side effect, this also enables a better dimensioning of the hardware platform.
So...

Did we break it?
So...

Did we break it?

No, but we need more to cover the development cycle.
Outline

1. Real-time
2. Multi-rate system design
3. Synchronous real-time
   - Arithmetic clocks
   - Multi-threaded execution
4. PRELUDE
   - The language
   - Compilation
5. Conclusion
Programming in the large: Aeronautics system design

Logical time and real-time in the Synchronous approach
Aeronautics system design

Identify “Aircraft functions” along with:
- Criticity levels
- Timing requirements
Aircraft functions

Example:

- Thruster control;
- Flight plan control;
- Aircraft control on ground;
  - Transition air/ground;
  - **Deceleration**;
  - Direction control on ground;
  - …

- …
Aeronautics system design

Logical time and real-time in the Synchronous approach

Identify “systems” that implement “functions”.

For each system:
- criticity level
- timing requirements
Aircraft systems

Example: **Ground deceleration** is made up of:

- The “thrust reversal” function of the **motor control** system;
- The “spoiler control” function of the **flight command** system;
- The **wheel brake** system.
Aeronautics system design

**Hardware architecture** of each system:
- Physical devices
- Computing units
- ...

Logical time and real-time in the Synchronous approach
Aeronautics system design

Logical time and real-time in the Synchronous approach

Conception / synthesis of laws
(Matlab / Simulink)
Aeronautics system design

For each computing unit:
Software architecture

Logical time and real-time in the Synchronous approach
Aeronautics system design

For each computing unit: 
*Software architecture*

« Flight control » example
- Set of communicating “blocks”
  ⇒ 5.000 blocks
  ⇒ 20.000 data-dependencies
Aeronautics system design

Logical time and real-time in the Synchronous approach
Aeronautics system design

Logical time and real-time in the Synchronous approach
Synchronous languages in the design

Logical time and real-time in the Synchronous approach
Synchronous languages in the design

- On the “system” level:
Synchronous languages in the design

- On the “system” level:
  - Functional level (SCADE, LUSTRE);
Synchronous languages in the design

On the “system” level:

- Functional level (SCADE, LUSTRE);
- Software architecture level?
Synchronous languages in the design

On the “system” level:

- Functional level (SCADE, LUSTRE);
- Software architecture level?

Timing requirements:
Synchronous languages in the design

- On the “system” level:
  - Functional level (SCADE, LUSTRE);
  - Software architecture level?

- Timing requirements:
  - Attached to blocks (software architecture);
  - Abstracted on functional level: blocks are mono-periodic.
Synchronous languages in the design

- On the “system” level:
  - Functional level (SCADE, LUSTRE);
  - Software architecture level?

- Timing requirements:
  - Attached to blocks (software architecture);
  - Abstracted on functional level: blocks are mono-periodic.

⇒ Can we introduce the synchronous paradigm at the software architecture level and deal with timing requirements there?
Real-time is replaced by a simplified, abstract, logical time.

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

- Zero-time?
  - In the semantics, the execution of one instant takes no time, everything happens simultaneously;
  - When implemented, the execution of one instant does take time;
  - The point is, when writing a synchronous program, we do not care about real-time.
A question of semantics

Zero-time?

- In the semantics, the execution of one instant takes no time, everything happens simultaneously;
- When implemented, the execution of one instant does take time;
- The point is, when writing a synchronous program, we do not care about real-time.

Synchronous hypothesis validation:

- In aeronautics design (and in many other cases), the periodicity of a block (LUSTRE program) sets the bound for the duration of an instant;
- At the end of the implementation process, the synchronous hypothesis must be validated, i.e. “do we have $C_i \leq T_i$?” (WCET analysis)
Multi-rate in Lustre/SCADe

Example

$\text{period} = 10\text{ms}$

$8\text{ms} \rightarrow F \rightarrow S \rightarrow 30\text{ms}$

Program (base period=10ms)

```plaintext
node multi_rate(i: int) returns (o: int)
var vf: int; clock3: bool; vs: int when clock3;
let
  (o, vf)=F(i, current(0 fby vs));
  clock3=everyN(3);
  vs=S(vf when clock3);
ret
```
Multi-rate in Lustre/SCADE

Behaviour:

<table>
<thead>
<tr>
<th>$vf$</th>
<th>$vf_0$</th>
<th>$vf_1$</th>
<th>$vf_2$</th>
<th>$vf_3$</th>
<th>$vf_4$</th>
<th>$vf_5$</th>
<th>$vf_6$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vf$ when $clock3$</td>
<td>$vf_0$</td>
<td></td>
<td>$vf_3$</td>
<td></td>
<td>$vf_6$</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>$vs$</td>
<td>$vs_0$</td>
<td>$vs_1$</td>
<td>$vs_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 fby vs</td>
<td>0</td>
<td>$vs_0$</td>
<td></td>
<td>$vs_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current (0 fby vs)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$vs_0$</td>
<td>$vs_0$</td>
<td>$vs_0$</td>
<td>$vs_1$</td>
<td></td>
</tr>
</tbody>
</table>

Program (base period=10ms)

```plaintext
node multi_rate(i: int) returns (o: int)
var vf: int; clock3: bool; vs: int when clock3;
let
  (o, vf)=F(i, current(0 fby vs));
clock3=everyN(3);
vs=S(vf when clock3);
tel
```
What’s missing?

- **For the programmer**: not immediate to see that \( v_f \) when \( \text{clock3} \) is 3 times slower than \( v_f \);

- **For the static analyses**: clocks = Boolean expressions \( \Rightarrow \) compiler does not see that "some clock is 3 times slower than another";

- **For the code generation**: computations must all complete during one base period (10ms).
Objective: multi-Rate Synchronous

Scale 1: slow instants (30ms)

Scale 2: fast instants (10ms)

Requirements:

- Define several logical time scales;
- Compare different logical time scales;
- Transition from one scale to another.
Bridging the gap

Main ideas:

- **Arithmetic clocks**: clocks defined, compared and transformed, using numbers and/or operations on numbers;

- **Multi-threaded execution**: not all operations must be executed within the same base period.
Outline

1. Real-time
2. Multi-rate system design
3. Synchronous real-time
   - Arithmetic clocks
   - Multi-threaded execution
4. PRELUDE
   - The language
   - Compilation
5. Conclusion
Motivation: implementing real-time streaming applications (e.g. video systems);

- Multi-rate systems;
- Combine flows that are “nearly synchronous”, i.e. the same production rate on a period of time, but not at the same instants.

Compiled into classic synchronous code + buffering mechanisms.
N-Synchronous (2)

Example

\[
\text{let node resync } x = o \text{ where}
\]
\[
\begin{align*}
\text{rec } & x1 = x \text{ when } (10) \\
\text{and } & x2 = x \text{ when } (01) \\
\text{and } & o = (\text{buffer } x1) + x2
\end{align*}
\]

Operators

- \(x \text{ when } (01)\): drop value, keep value, drop value, keep value, ...
- \(\text{buffer}(x1)\): buffer values to enable clock “resynchronization”.

Logical time and real-time in the Synchronous approach
N-Synchronous (2)

Example

```plaintext
let node resync x = o where
  rec x1 = x when (10)
  and x2 = x when (01)
  and o = (buffer x1) + x2
```

<table>
<thead>
<tr>
<th>flow</th>
<th>5 7 3 6 2 8 ...</th>
<th>clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>x1</td>
<td>5 3 2 ...</td>
<td>(10)</td>
</tr>
<tr>
<td>buffer(x1)</td>
<td>5 3 2 ...</td>
<td>(01)</td>
</tr>
<tr>
<td>x2</td>
<td>7 6 8 ...</td>
<td>(01)</td>
</tr>
<tr>
<td>o</td>
<td>12 9 10 ...</td>
<td>(01)</td>
</tr>
</tbody>
</table>
N-Synchronous (3)

- Rate relations are more explicit;
- Better static analyses;
- More general (too general?) than purely multi-periodic systems (e.g. clock (10110));
- Semantics still requires computations to fit within an instant.
CCSL

(Presented previously by AG).

- Very expressive: periodic, sampled, alternation, etc;
- Targeted mainly for simulation/verification;
- Too general for efficient compilation (?)
Strictly Periodic Clocks

- Definition: Clock \((n, p)\) is a clock of period \(n\) and phase \(p\);
- Example: \((120, 1/2)\) activates at dates 60, 180, 300, 420, . . .
- Rate transformations:
  - \(\alpha / k\): divide frequency;
  - \(\alpha \times k\): multiply frequency;
  - \(\alpha \rightarrow q\): offset activations.
Strictly Periodic Clocks(2)

- Strictly periodic clocks are dedicated to multi-periodic real-time systems;
- Strictly periodic clocks are a sub-class of Boolean clocks and of N-Synchronous clocks;
- This restriction enables to compile real-time aspects more efficiently.
Outline

1. Real-time
2. Multi-rate system design
3. Synchronous real-time
   - Arithmetic clocks
   - Multi-threaded execution
4. Prelude
   - The language
   - Compilation
5. Conclusion

Logical time and real-time in the Synchronous approach
Relaxed Synchronous hypothesis

Classic Synchronous hypothesis
All computations complete before the end of the instant.

Relaxed Synchronous hypothesis
Computations complete before their next activation.

- Relaxed: mere reformulation of classic;
- Classic: particular case of relaxed;
- Relaxed: supports several logical time scales;
- Relaxed: fits with periodicity constraints "a task instance must complete before the next task release".
Automated code distribution into threads

(Presented previously by AG-not the same).

**Approach 1**: Automatically split the code into several threads:
- In Signal: split code based on clocks;
- In Lustre: split code based on inputs/outputs;
- Add buffers to communicate between threads.
Automated code distribution into threads (2)

More general than periodic systems, thus:

- Buffer dimensioning is harder;
- Temporal analyses is harder;
- The user must specify the distribution criteria.
Lustre with Futures

**Approach 2**: Explicit thread encapsulation.

**Example**

```plaintext
code
node slow_fast() = (y:float)
var big : bool; yf, v : float; ys : future float;
let
    big = everyN(3);
    ys = (async 0.0) fby (async slow(y when big));
    yf = fast(v whenot big);
    y = merge big (!ys) (yf);
    v = 0.0 fby y;
tel
```

- **async** encapsulates a node inside a thread;

- The value of an asynchronous flow is fetched using operator `!`.

**NB** The values and clocks of `!x` and `x` are exactly the same.
Lustre with Futures

**Approach 2**: Explicit thread encapsulation.

**Example**

```plaintext
node slow_fast() = (y:float)
var big : bool; yf, v : float; ys : future float;
let
    big = everyN(3);
    ys = (async 0.0) fby (async slow(y when big));
    yf = fast(v whenot big);
    y = merge big (!ys) (yf);
    v = 0.0 fby y;
```

<table>
<thead>
<tr>
<th>big</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>!ys</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yf</td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4.14</td>
<td>...</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
<td>3.14</td>
<td>4.14</td>
</tr>
<tr>
<td>v</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
<td>3.14</td>
</tr>
</tbody>
</table>
```

Logical time and real-time in the Synchronous approach
Lustre with Futures (2)

- Good multi-thread support;
- No real-time constraints attached to threads.
Prelude

**Approach 3**: Thread assembly language.

- Each node invocation is encapsulated inside a thread;
- Targeted for the software architecture level;
- Real-time characteristics are associated to each node/thread.
Outline

1. Real-time
2. Multi-rate system design
3. Synchronous real-time
   - Arithmetic clocks
   - Multi-threaded execution
4. PRELUDE
   - The language
   - Compilation
5. Conclusion
Prelude: a real-time synchronous language

- **Initial question:** how to program systems with multiple real-time constraints in a synchronous style?

- **Context:**
  - Defined and developed at ONERA (first during speaker thesis);
  - Motivated by collaborations with Airbus and Astrium (satellites).

- **Main principles:**
  - Strictly periodic clocks;
  - Relaxed synchronous hypothesis;
  - Fully multi-threaded;
  - At the software architecture level.
Multi-rate system

\[ \text{period} = 10\text{ms} \]

\[ 8\text{ms} > \]

\[ \text{period} = 30\text{ms} \]
Operations: imported nodes

- Operations of the system are imported nodes;
- External functions (e.g. C, or LUSTRE);
- Declare the worst case execution time (wcet) of the node.

**Example**

```plaintext
imported node F(i, j: int) returns (o, p: int) wcet 2;
imported node S(i: int) returns (o: int) wcet 10;
```
Real-time constraints

Multi-rate system

\[ \text{period} = 10\text{ms} \]

\[ 8\text{ms} > \]

\[ \text{period} = 30\text{ms} \]
Real-time constraints: clocks and deadlines

- Real-time constraints are specified in the signature of a node;
- Periodicity constraints on inputs/outputs;
- Deadline constraints on inputs/outputs.

Example

```plaintext
node sampling(i: rate (10,0)) returns (o: rate (10,0) due 8)
let
  ...
tel
```

Input/output rate can be unspecified, the compiler will infer it.
Multi-rate communications

Multi-rate system

\[ \text{period} = 10\text{ms} \]

\[ 8\text{ms} \]

\[ \text{period} = 30\text{ms} \]
Multi-rate communications: rate transition operators

Example

node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
  let
  (o, vf)=F(i, 0 fby vs)∗3);
  vs=S(vf/ˆ3);
  tel

Rate transition operators:

- Sub-sampling: \( x/^{\wedge}3 (ck(x)/.3) \);
- Over-sampling: \( x*^{\wedge}3 (ck(x)*.3) \).
## Multi-rate communications: rate transition operators

### Example

```javascript
node sampling(i: rate (10, 0)) returns (o)
    var vf, vs;
    let
        (o, vf)=F(i, (0 fby vs) * ^3);
        vs=S(vf/^3);
    tel
```

| date  | 0  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | ...
|-------|----|----|----|----|----|----|----|----|----|-----|
| vf    | vf0| vf1| vf2| vf3| vf4| vf5| vf6| vf7| vf8| ...
| vf/^3 | vf0|    |    |    |    |    |    |    |    | ...
| vs    | vs0|    |    |    |    |    |    |    |    | ...
| 0 fby vs | 0 |    |    |    |    |    |    |    |    | ...
| (0 fby vs) * ^3 | 0 | 0 |    |    |    |    |    |    |    | ...

Logical time and real-time in the Synchronous approach
And...
And...

That’s all folks!
Formal semantics: Strictly Periodic Clocks

- Flow values are tagged by a date: \( f = (v_i, t_i)_{i \in \mathbb{N}} \);
- Clock = sequence of tags of the flow;
- Value \( v_i \) must be produced during time interval \([t_i, t_{i+1}]\);
- A clock is **strictly periodic** iff:
  \[
  \exists n \in \mathbb{N}^*, \forall i \in \mathbb{N}, \quad t_{i+1} - t_i = n
  \]
- \( n \) is the period of \( h \), \( t_0 \) is the phase of \( h \).
- Eg: \((120, 1/2)\) is the clock of period 120 and phase 60.
Formal semantics: operators

Example

\[ +\#((v, t).s, (v', t).s') = (v + v', t).+\#(s, s') \]

- \((v, t).s\): denotes value \(v\) produced at time \(t\) and followed by sequence \(s\);
- \(op\#(f, f') = (v_1, t_1).(v_2, t_2)\ldots\) denotes the flow produced when applying \(op\) to flows \(f\) and \(f'\).

Warning:

- The semantics is **ill-defined for asynchronous flows**;
- \(\Rightarrow\) Static analyses required to check that program semantics is well-defined before further compilation.
Formal semantics: classic operators

\[ \text{fb}_y \# (v, (v', t).s) = (v, t). \text{fb}_y \# (v', s) \]

when \( \# ((v, t).s, (true, t).cs) = (v, t). \text{when} \# (s, cs) \)

when \( \# ((v, t).s, (false, t).cs) = \text{when} \# (s, cs) \)
Formal semantics: rate transitions

\[ \ast \hat{\#}( (v, t).s, k) = \prod_{i=0}^{k-1} (v, t'_i).\ast \hat{\#}(s, k) \]

(with \( t'_0 = t \) and \( t'_{i+1} - t'_i = \pi(s)/k \))

\[ /\hat{\#}( (v, t).s, k) = \begin{cases} (v, t)./\hat{\#}(s, k) & \text{if } k \ast \pi(s) \mid t \\ /\hat{\#}(s, k) & \text{otherwise} \end{cases} \]
Compilation overview

- **Program**: A program undergoes static analyses which extract real-time tasks.
- **Static Analyses**: Determine if the semantics are preserved.
- **Task Extraction**: Extracts real-time tasks.
- **Schedulability Analysis**: Checks if the tasks can be scheduled.
- **Code Generation**: Generates multi-threaded C code.
- **Multi-threaded C Code**: Represents the final output.

Flow Diagram:
- **Stop**: Represents a failure point.
- **Succeed**: Represents a successful step.

Logical time and real-time in the Synchronous approach.
Static analyses

- Typing: no run-time type error;
- Causality analysis: no cyclic data-dependencies;
- Clock calculus: values are only accessed when they should be.
Clock calculus: example

Example

node under_sample(i) returns (o)
  let o=i/^2; tel

node poly(i: int rate (10, 0); j: int rate (5, 0))
  returns (o, p: int)
  let
    o=under_sample(i);
    p=under_sample(j);
  tel

Result inferred by the clock calculus

under_sample: 'a->'a/.2
poly: ((10,0) * (5,0)) -> ((20,0) * (10,0))
Task graph extraction

Program

node sampling(i: rate (10, 0)) returns (o)
    var vf, vs;
    let
        (o, vf)=F(i, (0 fby vs)"^3);
        vs=S(vf/"^3);
    tel

Task graph

Logical time and real-time in the Synchronous approach
For each task:

- Repetition period: $T_i = \pi(ck_i)$;
- Relative deadline: $D_i = T_i$ by default or explicit constraint (e.g., o: due 8);
- Worst case execution time: $C_i$, declared for each imported node;
- Initial release date: $O_i = \varphi(ck_i)$. 

Logical time and real-time in the Synchronous approach
Multi-rate data-dependencies

For each task dependency:

1. **Data can only be consumed after being produced** ⇒ precedence constraints for the scheduler;

2. **Data must not be overwritten before being consumed** ⇒ communication protocol.

**Example**

\[ A \, ^{\wedge} \rightarrow \, B:\]

\[ A \quad B \quad B \]

(1): \( B_0 \) after \( A_0 \)

(2) keep \( A_0 \) available
Communication protocol

- Tailor-made buffering mechanism;
- For each dependency, computes:
  - Size of the buffer;
  - Where each job writes/reads;
- **Independent of the scheduling policy**;
- Requires a single central memory.
Communication protocol

Ex: \( B(A(x) \ast ^{3/2}) \), ie \( A \rightarrow ^{3/2} B \):

**Semantics**

<table>
<thead>
<tr>
<th>date</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(x)</td>
<td>(a_0)</td>
<td>(a_1)</td>
<td>(a_2)</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(x) * ^3</td>
<td>(a_0)</td>
<td>(a_0)</td>
<td>(a_0)</td>
<td>(a_1)</td>
<td>(a_1)</td>
<td>(a_2)</td>
<td>(a_2)</td>
<td>(a_3)</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>A(x) * ^3/2</td>
<td>(a_0)</td>
<td>(a_0)</td>
<td>(a_1)</td>
<td>(a_2)</td>
<td>(a_3)</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lifespans**

Logical time and real-time in the Synchronous approach
Lifespans

- Buffer of size 2;
- Write in the buffer cyclically;
- Read from the buffer cyclically;
- Do not advance at the same pace for reading and writing.
Scheduling: problem parameters

- A set of recurring tasks with:
  - Periods, deadlines, wcets, release dates;
  - Multi-rate precedence constraints.

- Hardware architecture:
  - Mono-core;
  - Multi-core (with a single central shared memory).

- Scheduler class:
  - On-line/off-line;
  - Static/dynamic priorities;
Outline

1. Real-time
2. Multi-rate system design
3. Synchronous real-time
   - Arithmetic clocks
   - Multi-threaded execution
4. Prelude
   - The language
   - Compilation
5. Conclusion

Logical time and real-time in the Synchronous approach
Summary

What you should remember:

- When we deal with multi-periodic systems, we need explicit real-time constraints;

- Explicit RT constraints enable:
  - Static real-time analyses;
  - Optimized processor utilization and platform dimensioning.

- Real-time constraints can be introduced without breaking the synchronous paradigm;

- Mixing real time and logical time can be done by using real-time as a “dimension” for logical time.
Some inspirations for this course:

- **Frédéric Boniol (ONERA Toulouse)**, Modélisation et programmation des systèmes embarqués critiques : la voie synchrone, *course at Ecole Polytechnique de Montreal, 2013*

- **Emmanuel GROLLEAU (LIAS/ISAE-ENSMA)**, Ordonnancement et ordonnançabilité monoprocesseur, *Ecole d’Été Temps Réel (ETR’2011), Brest, 2011*
Prelude is a joint work with Frédéric Boniol, David Lesens and Claire Pagetti.

**Julien Forget.**
Prelude: programming critical real-time systems.

**Julien Forget.**
A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints.

**Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti.**
A real-time architecture design language for multi-rate embedded control systems.

**Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens.**
Multi-task implementation of multi-periodic synchronous programs.