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:
  - Acquire inputs on sensors;
  - Compute;
  - 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.

---

### Classic model

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 $\tau_i$.

---

### 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.

---

### 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 60 ms
  - Deadline miss $\Rightarrow$ loss of control.
- **Failure detection** (internal): check inconsistencies every 200 ms
  - Deadline miss $\Rightarrow$ crash with motors on.
- ...
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;
  $$\Rightarrow$$ 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

$$\tau_B(T_B = 9, C_B = 5)$$ and $$\tau_A(T_A = 3, C_A = 1)$$:
- Without preemption:
  - Deadline miss
- With preemption:

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
- With preemption:
Scheduling policy example: Rate-Monotonic

- 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.

$\Rightarrow$ What does **optimal** mean?

Rate-Monotonic analysis

**Sufficient** schedulability test:

$$\sum_{i=0}^{m} \frac{C_i}{T_i} \leq m\left(2^{1/m} - 1\right)$$

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

$\Rightarrow$ What does **sufficient** mean?

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

Okay...

But, we were told to ignore real-time!

(cf...)

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?

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

Outline

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

Programming in the large: Aeronautics system design

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

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

Logical time and real-time in the Synchronous approach

Real-time Multi-rate system design Synchronous real-time PRELUDE Conclusion

Conception / synthesis of laws (Matlab / Simulink)

For each computing unit: Software architecture

"Flight control" example
- Set of communicating "blocks"
  5,000 blocks
  20,000 data-dependencies

For each computing unit: Software architecture

Functionnal block = Lustre/SCADE-Suite

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

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

Synchronous approach (reminder)

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.

Logical time and real-time in the Synchronous approach

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?
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

Behaviour:

<table>
<thead>
<tr>
<th>$vf$ when clock3</th>
<th>$v_0$</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$v_4$</th>
<th>$v_5$</th>
<th>$v_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs</td>
<td>$v_{00}$</td>
<td>$v_{01}$</td>
<td>$v_{02}$</td>
<td>$v_{03}$</td>
<td>$v_{04}$</td>
<td>$v_{05}$</td>
<td></td>
</tr>
<tr>
<td>$0 \text{ fby } vs$</td>
<td>0</td>
<td>$v_{10}$</td>
<td>$v_{11}$</td>
<td>$v_{12}$</td>
<td>$v_{13}$</td>
<td>$v_{14}$</td>
<td></td>
</tr>
<tr>
<td>current ($0 \text{ fby } vs$)</td>
<td>0</td>
<td>0</td>
<td>$v_{10}$</td>
<td>$v_{11}$</td>
<td>$v_{12}$</td>
<td>$v_{13}$</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
```

Example

```
8ms > F
  period = 10ms

period = 30ms
S
```

What's missing ?

• For the programmer: not immediate to see that $vf$ when clock3 is 3 times slower than $vf$;
• 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

Requirements:
- Define several logical time scales;
- Compare different logical time scales;
- Transition from one scale to another.

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

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.

N-Synchronous

- **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

\[
\text{let node resync } x = o \text{ where } \\
\quad \text{rec } x_1 = x \text{ when (10)} \\
\quad \text{and } x_2 = x \text{ when (01)} \\
\quad \text{and } o = (\text{buffer } x_1) + x_2
\]

Operators

- \(x \text{ when (01)}\): drop value, keep value, drop value, keep value, ...
- \(\text{buffer}(x_1)\): buffer values to enable clock “resynchronization”.

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;
  - \(\alpha * k\): multiply frequency;
  - \(\alpha \rightarrow q\): offset activations.

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

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.

**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;
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.

More general than periodic systems, thus:
- Buffer dimensioning is harder;
- Temporal analyses is harder;
- The user must specify the distribution criteria.
Logical time and real-time in the Synchronous approach

Prelude

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.

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

Lustre with Futures (2)

- Good multi-thread support;
- No real-time constraints attached to threads.
**Logical time and real-time in the Synchronous approach**

**Real-time constraints**

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

**Example**

```
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.

---

**Operations**

**Multi-rate system**

- Periodicity constraints on inputs/outputs:
  - \[ \text{period} = 10 \text{ms} \]
  - \[ 8 \text{ms} > \]
  - \[ \text{period} = 30 \text{ms} \]

**Conclusion**

- 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**

```
imported node F(i, j: int) returns (o, p: int) wcet 2;
imported node S(i: int) returns (o: int) wcet 10;
```
Multi-rate communications

Multi-rate system

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

\[ \text{period} = 30 \text{ms} \]

8ms > F \rightarrow S

Multi-rate communications: rate transition operators

Example

\[ \text{node sampling}(i: \text{rate}(10, 0)) \text{ returns } (o) \]
\[ \text{var } vf, \ vs; \]
\[ \text{let } \]
\[ (o, \vf) = F(i, (0 \text{ fby } vs) \ast \hat{3}); \]
\[ vs = S(vf / \hat{3}); \]
\[ \text{tel} \]

Rate transition operators:

- Sub-sampling: \( x / \hat{3} \ (ck(x) / 3); \)
- Over-sampling: \( x \ast \hat{3} \ (ck(x) \ast 3). \)

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 if:
  \[ \exists n \in \mathbb{N}^*, \forall i \in \mathbb{N}, t_{i+1} - t_i = n \]
- $n$ 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;
- \[ \Rightarrow \text{Static analyses required to check that program semantics is well-defined before further compilation.} \]

Formal semantics: classic operators

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

when $^#((v, t).s, (true, t).cs) = (v, t)$. when $^#(s, cs)$

when $^#((v, t).s, (false, t).cs) = \text{when } ^#(s, cs)$

Formal semantics: rate transitions

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

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

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

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

```c
node under_sampling(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_sampling(i);
        p=under_sampling(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

```c
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
Real-time characteristics

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

Multi-rate data-dependencies

For each task dependency:
- Data can only be consumed after being produced \( \Rightarrow \) precedence constraints for the scheduler;
- Data must not be overwritten before being consumed \( \Rightarrow \) communication protocol.

Example

\[ A \xleftarrow{\frac{3}{2}} B: \]

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.

Ex: \( B (A(x) \xrightarrow{3/2}) \), ie \( A \xleftarrow{\frac{3}{2}} B: \)

Semantics

\[
\begin{array}{c|cccccccccc}
\text{date} & 0 & 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & \ldots \\
A(x) & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & \ldots \\
A(x) \xrightarrow{\frac{3}{2}} & a_0 & a_0 & a_0 & a_1 & a_1 & a_2 & a_2 & a_3 & a_3 & \ldots \\
\end{array}
\]

Lifespans

- Requires a single central memory.
Communication protocol (2)

**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

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’Eté Temps Réel (ETR’2011), Brest, 2011*

---

**References**

Prelude is a joint work with Frédéric Boniol, David Lesens and Claire Pagetti.


