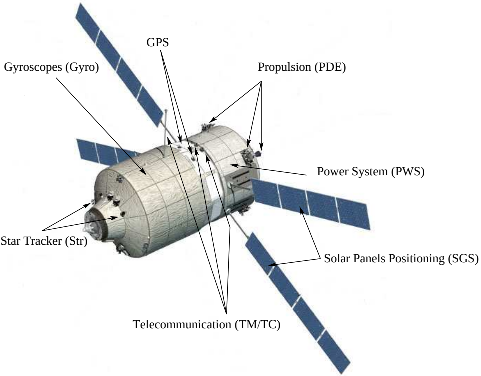
Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language

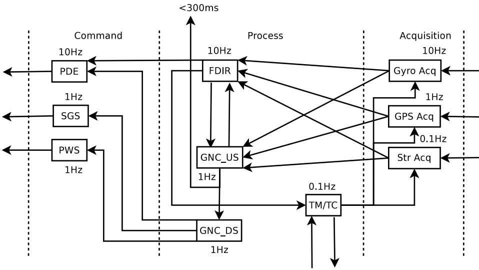
Julien FORGET, Frédéric BONIOL, Claire PAGETTI

LIFL - Lille / ONERA - Toulouse

Synchron 2011

Context: the Flight Application Software





Important characteristics

- **Multi-periodic**: different pieces of equipment = different control rates;
- Operations of different rates communicate;
- Mission critical systems (functional and temporal determinism required).

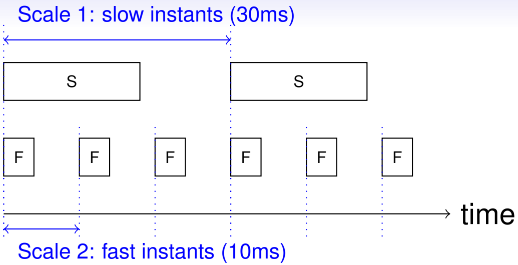
Main characteristics of PRELUDE

- Integration language (Architecture Design Language);
- Synchronous semantics;
- High-level **real-time primitives**: periods, deadlines;
- Compiled into a set of concurrent real-time tasks (semantics preserved);

$\Rightarrow$ **feasibility analysis** and **efficient preemptive scheduling**.

Outline

Multi-periodic synchronous



Requirements:

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

⇒ Introduce the real-time scale, as a reference between different logical time scales.

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 $[t_i, t_{i+1}[$;
- Clock $ck = (t_i)_{i \in \mathbb{N}}$ is **strictly periodic** iff:

$$\exists n \in \mathbb{N}^{+*},\ \forall i \in \mathbb{N},\ t_{i+1} - t_i = n$$

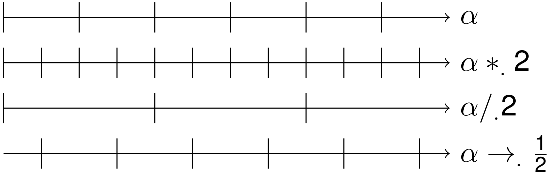
- $\pi(ck) = n$ is the **period** of h. $\varphi(ck) = t_0$ is the **phase** of h.
- Eg: $(120, 1/2)$ is the strictly periodic clock of period 120 and phase 60.

Advantage: easy to extract real-time characteristics.

Periodic Clock Transformations

Rate transformations:

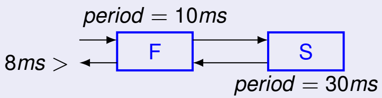
- $\alpha /. k$: divide frequency;
- $\alpha *. k$: multiply frequency;
- $\alpha \rightarrow. q$: offset activations.



Outline

Operations

Multi-rate system



Operations: Imported nodes

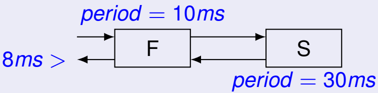
- Operations of the system are imported nodes;
- External functions (e.g. C, or LUSTRE);
- Declare the worst case execution time (wcet) of the node.

Example

```
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



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.

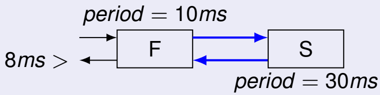
Example

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

Input/output can be unspecified, the compiler will infer it.

Multi-rate communications

Multi-rate system



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/^3);
tel
```

Rate transition operators:

- Sub-sampling: $x/\hat{}3$ (has $clock(x)/_{.}3$);
- Over-sampling: $x*\hat{}3$ (has $clock(x)*_{.}3$).

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/^3);
tel
```

date	0	10	20	30	40	50	60	70	80	...
`vf`	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	vf_7	vf_8	...
`vf/^3`	vf_0			vf_3			vf_6			...
`vs`	vs_0			vs_1			vs_2			...
`0 fby vs`	0			vs_0			vs_1			...
`(0 fby vs)*^3`	0	0	0	vs_0	vs_0	vs_0	vs_1	vs_1	vs_1	...

Formal semantics (just a little)

$$\texttt{fby}^{\#}(v,(v',t).s) = (v,t).\ \texttt{fby}^{\#}(v',s)$$

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

$$\texttt{merge}^{\#}((true,t).s,(v,t).s_1,s_2)) = (v,t).\texttt{merge}^{\#}(s,s_1,s_2))$$
$$\texttt{merge}^{\#}((false,t).s,s_1,(v,t).s_2)) = (v,t).\texttt{merge}^{\#}(s,s_1,s_2)$$

Formal semantics (just a little more)

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

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

$$\hat{/}^{\#}((v,t).s,k) = \begin{cases} (v,t).\hat{/}^{\#}(s,k) & \text{if } k * \pi(s) | t \\ \hat{/}^{\#}(s,k) & \text{otherwise} \end{cases}$$

Outline

Static analyses

- Typing;
- Causality analysis;
- Clock calculus;
- **Feasibility analysis**: check that all the **deadlines will be met**.

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

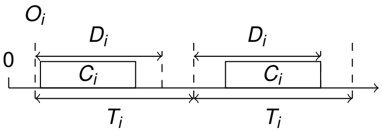
```
under_sample: 'a->'a/.2
poly: ((10,0) * (5,0))->((20,0) * (10,0))
```

Outline

Task model

Objective: translate the program into the “standard” task model to benefit from real-time scheduling theory.

A set of tasks $\tau_i(T_i, D_i, C_i, O_i)$ + **precedence constraints**.



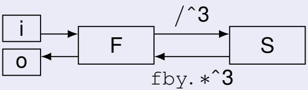
- T_i: period;
- $D_i \leq T_i$: relative deadline;
- C_i: worst case execution time;
- O_i: initial release date.

Tasks and data-dependencies

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



Real-time attributes

For each task τ_i of clock ck_i:

- Period: $T_i = \pi(ck_i)$;
- Execution time: declared in the program;
- Initial release date: $O_i = \varphi(ck_i)$.
- Relative deadline: explicit constraint (eg `o: due 8`), or $D_i = T_i$ by default.

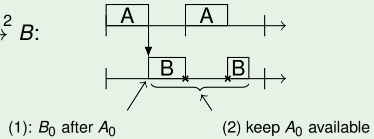
Respecting multi-rate data-dependencies

For each data-dependency:

1. Data can only be consumed after being produced ⇒ **precedence constraints+ad-hoc scheduling policy**;
2. Data must not be overwritten before being consumed ⇒ **communication protocol**.

Example

$A \overset{/\hat{}2}{\rightarrow} B$:



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.

Outline

Reminder: simple precedences

Constraints between tasks of the same period (CHETTO90):

1. Use the Earliest-Deadline-First policy;
2. Adjust D_i and R_i for all precedence $\tau_i \rightarrow \tau_j$:
   - $D_i^* = min(D_i, \min_{\tau_j \in \textbf{succs}(\tau_i)}(D_j^* - C_j))$
   - $R_j^* = max(R_j, \max_{\tau_i \in \textbf{preds}(\tau_j)}(R_i^*))$
3. Resulting problem $\Leftrightarrow$ Original problem;
4. **Optimal** policy (finds a solution if there exists one).

Houssine Chetto, Marilyne Silly, and T. Bouchentouf.
Dynamic scheduling of real-time tasks under precedence constraints.
Real-Time Systems, 2, 1990.

Extended precedence constraints

For a pair of related tasks:

- Only a subset of the task instances are constrained;
- Constraints follow a repetitive pattern
  $\Rightarrow$ **Encode the pattern of task instance constraints**.

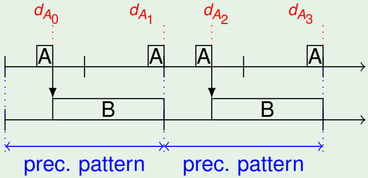
1. **Release dates**: synchronous $\Rightarrow$ encoding respected by default;
2. **Deadline words**: $(3.5)^\omega$ = the sequence of task instance deadlines 3.5.3.5.3.5. . . .

Example

$A \overset{/\hat{}2}{\rightarrow} B$

$T_A = 5,\ T_B = 10,\ C_B = 7,$
$C_A = 1$

$d_A = (3.5)^\omega$



Feasibility analysis

Definition: check that **deadlines** and **precedence constraints** will be respected at execution.

1. Encode extended precedence constraints $\Rightarrow$ a deadline word for each task;
2. Unfold task set on feasibility interval, compute task instance deadlines based on deadline words;
3. Reuse existing feasibility tests.

This works basically the same way for static and dynamic priority scheduling policies.

Implementing synchronizations

Three possible implementations:

1. Embed deadline words in the program;
   - No need for semaphores / High space complexity.
2. Use counting semaphores:
   - Low complexity / Industrials hate semaphores !
3. Compute adjusted deadlines on-the-fly:
   - Low space-complexity / High time-complexity.

With the 3 implementations, we are sure that we will have:

- No priority inversion;
- No scheduling anomaly.

Outline

Overview

Solution proposed at ONERA-Toulouse:

- Encode dependent task system as an automata;
- Check that no “unschedulable” state is reachable;

either Use a model-checker to compute an off-line schedule (optimal);

or Schedule on-line with existing policies + counting semaphores (sub-optimal).

Implemented in SchedMCore ($\simeq$ OS for multi-core execution).

`http://sites.onera.fr/schedmcore/`

Outline

Download !

- C code generation (OS independent);
- Multi-threaded execution (OS dependent):
  - SCHEDMCORE;
  - MARTE OS (Cantabria) **broken**.
- Compiler developed in OCAML $\simeq$ 3000 lines (sources available).

http://www.lifl.fr/~forget/

(See the website for bibliographic references).

Perspectives

- **Cluster nodes** inside tasks to reduce the number of tasks;
- Support **mode automata** (part of the thesis of Rèmy WYSS):
  - Makes the clock calculus more complex;
  - Conditional scheduling required.
- A new kind of **temporal verification** ?