A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints

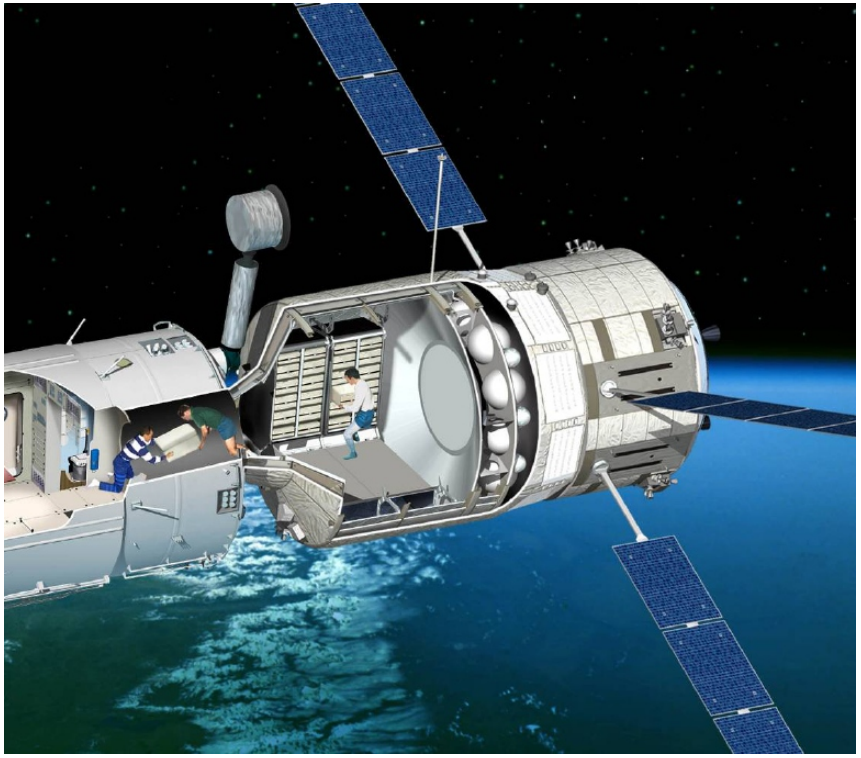
Julien FORGET

ONERA - Toulouse

Thesis supervised by: F. BONIOL(ONERA), D. LESENS (Astrium) and C. PAGETTI (ONERA)

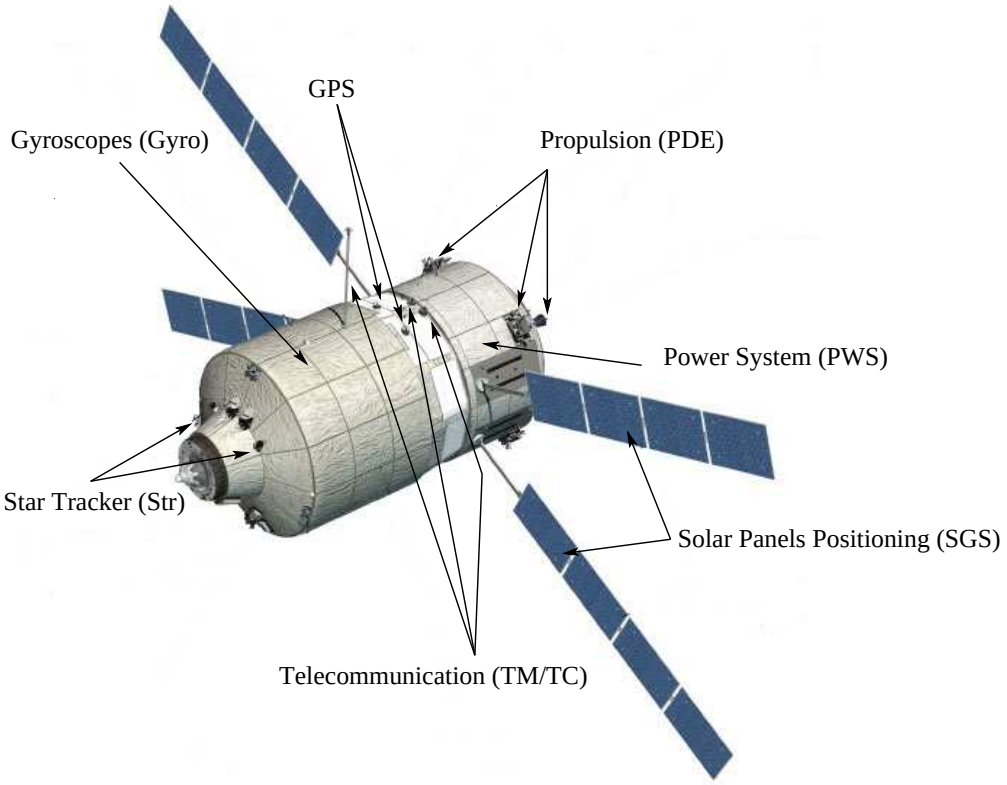
November 19, 2009

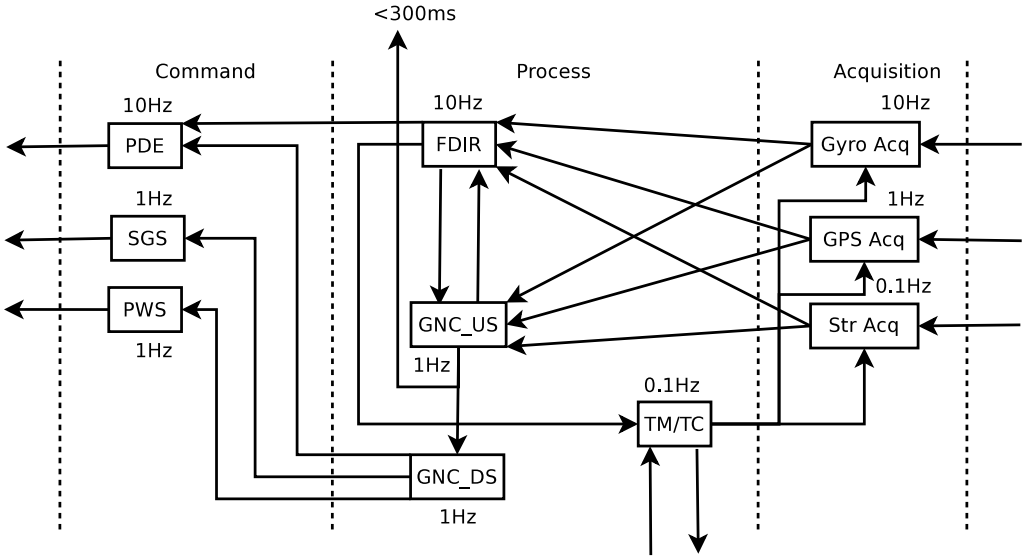
Embedded Control Systems: example



The **Automated Transfer Vehicle**, designed by **EADS Astrium Space Transportation** for ESA, for resupplying the International Space Station.

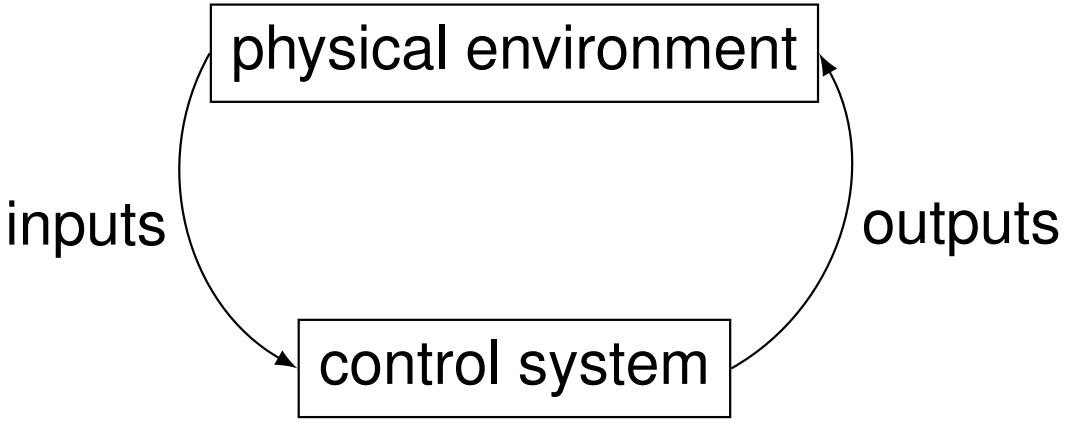
The Flight Application Software





Flight control system of the ATV.

Embedded Control Systems: definition



- **Control loop(s)**: Acquire inputs - Compute - Produce Outputs
  $\Rightarrow$ repeat indefinitely;
- **Role**: control a device in its physical environment.

Important characteristics

- **Highly regular**: no dynamic process creation, bounded loops, ...
- **Hard real-time constraints**: periods, deadlines;
- **Multi-rate**: different pieces of equipment = different control rates;
- **Operations of different rates communicate**;
- **Mission critical** systems.

Objective

Define a language and the associated compiler for Embedded Control Systems:

- To specify the **functional architecture**;
- To specify the **temporal architecture**;
- Critical systems $\Rightarrow$ requires **deterministic**, **predictable** programs.

Main characteristics of the language

- Architecture Design Language (ADL), **integration language**;
- Synchronous semantics, thus **formal**;
- High-level real-time primitives: **periods**, **deadlines**;
- **Rate transition** operators;
- Compiled into a set of **concurrent real-time tasks** $\Rightarrow$ efficient preemptive scheduling;
- Generates multi-threaded C code that **preserves the semantics** of the original program;
- Executes on a standard real-time platform.

Outline

Fundamental basis: the synchronous model

- Behaviour described as a succession of highly regular reactions called **instants**;
- Expressions and variables represent infinite sequences of values, called **flows**;
- **Synchronous hypothesis**: computations performed during an instant complete before the beginning of the next instant;
- If this condition is fulfilled, the programmer can simply **ignore the duration of an instant**;
- $\Rightarrow$ behaviour described on a **logical time scale**;
- **Clocks** (Boolean conditions) enable the definition of several logical time scales.

Benveniste, A. and Berry, G. (2001).
The synchronous approach to reactive and real-time systems.
In *Readings in hardware/software co-design*. Kluwer Academic Publishers.

Simple LUSTRE/SCADE program

Example

```
node everyN(n: int) returns(reached: bool)
var count: int;
let
  count=0 fby (count+1);
  reached=(count mod n=0);
tel
```

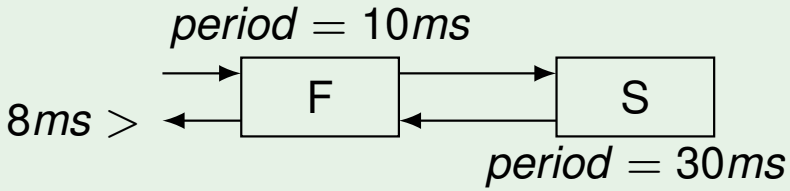
Behaviour:

`i`	3	3	3	3	3	...
`count=0 fby (count+1)`	0	1	2	3	4	...
`count mod n`	0	1	2	0	1	...
`reached`	true	false	false	true	false	...

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991).
The synchronous data-flow programming language LUSTRE.
Proc. IEEE, 79(9).

Multi-rate in LUSTRE/SCADE

Example



Program (base period=10ms)

```
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
```

Multi-rate in LUSTRE/SCADE

Behaviour:

vf	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	...
vf **when** clock3	vf_0			vf_3			vf_6	...
vs	vs_0			vs_1			vs_2	...
0 **fby** vs	0			vs_0			vs_1	...
current (0 **fby** vs)	0	0	0	vs_0	vs_0	vs_0	vs_1	...

Program (base period=10ms)

```
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
```

Limitations: comparing different rates

Program (base rate=10ms)

```
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
```

- **For the programmer**: not immediate to see that `vf when clock3` is 3 times slower than `vf`;
- **For the compiler**: clocks = Boolean expressions $\Rightarrow$ it cannot analyze clocks to see that "a clock is 3 times slower than the other".

Limitations: rate transitions

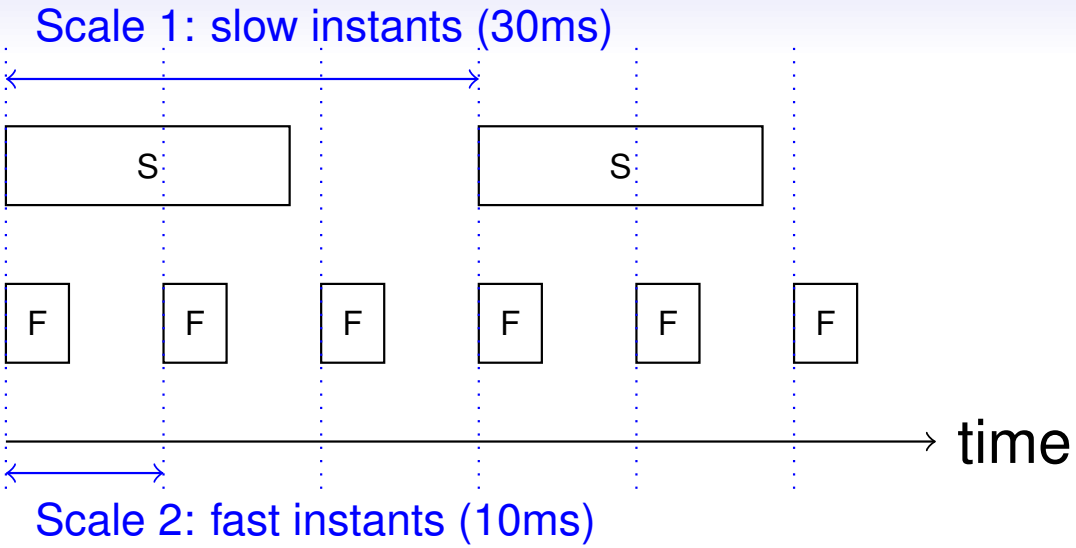
How to program transitions between flows of different rates ?

Example

```
node transition (i: int) returns (o: int)
var clock3, clock6: bool; i3, i6: int;
let
  clock3=everyN(3); i3=i when clock3;
  clock6=everyN(6); i6=i when clock6;
  v3=i3+(current(i6) when clock3);
  o=current(i3);
tel
```

NB: operands of the arithmetic/logic operations must have the same clock.

Extension: Multi-Rate Synchronous



Requirements:

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

$\Rightarrow$ Introduce the real-time scale, as a reference between different logical time scales.

A specific class of real-time clocks: Strictly Periodic Clocks

- We need to clearly separate two complementary notions:
  - The **real-time rate** of a flow $\Rightarrow$ strictly periodic clocks;
  - The **activation condition** of a flow on a given rate $\Rightarrow$ Boolean clocks.
- Strictly periodic clocks can statically be **compared** and **tested for equivalence**;
- Specific transformations are introduced to define **rate transition operators**.

Strictly periodic clocks:

$\Rightarrow$ **Enable efficient real-time scheduling;**

$\Rightarrow$ **Complement and do not replace Boolean clocks.**

A specific class of real-time clocks (2)

- Strictly periodic clocks can be considered as **a sub-class of Boolean clocks**;
- However, this restriction enables to compile real-time aspects more efficiently.

Alras, M., Caspi, P., Girault, A., and Raymond, P. (2009).
Model-based design of embedded control systems by means of a synchronous intermediate model.
In *International Conference on Embedded Software and Systems (ICESS'09)*, Hangzhou, China.

Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., and Pouzet, M. (2006).
N-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems.
In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, USA.

Smarandache, I. and Le Guernic, P. (1997).
A canonical form for affine relations in signal.
Technical Report RR-3097, INRIA.

Strictly Periodic Clocks: definitions

- Each value of a flow is **tagged by a date**: $x = (v_i, t_i)_{i \in \mathbb{N}}$;
- The sequence of tags is the clock of the flow;
- Value v_i must be produced during time interval $[t_i, t_{i+1}[$;
- A clock $ck = (t_i)_{i \in \mathbb{N}}$ is **strictly periodic** if and only if the interval between two successive tags is constant:

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

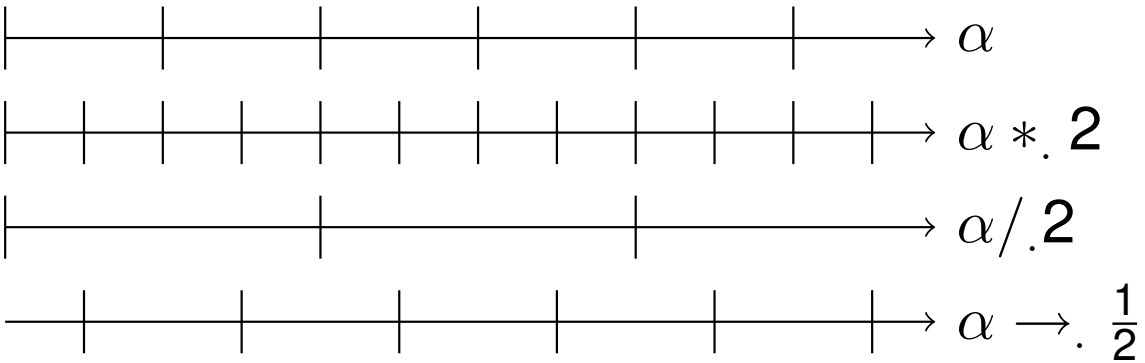
- $\pi(ck) = n$ is the **period** of h. $\varphi(ck) = t_0$ is the **phase** of h.
- Eg: $(120, 1/2)$ is the strictly periodic clock of period 120 and phase 60.

Periodic Clock Transformations

Different logical time scales (strictly periodic clocks) can be compared thanks to their real-time characteristics.

$\Rightarrow$ Rate transformations:

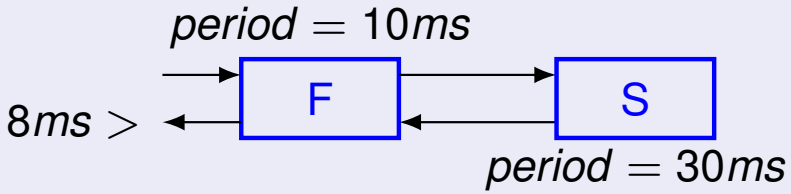
- **Division** : $\pi(\alpha /. k) = k * \pi(\alpha),\ \varphi(\alpha /. k) = \varphi(\alpha)\ (k \in \mathbb{N}^{+*})$
- **Multiplication** : $\pi(\alpha *. k) = \pi(\alpha)/k,\ \varphi(\alpha *. k) = \varphi(\alpha)\ (k \in \mathbb{N}^{+*})$
- **Phase offset** : $\pi(\alpha \rightarrow. q) = \pi(\alpha),\ \varphi(\alpha \rightarrow. q) = \varphi(\alpha) + q * \pi(\alpha)$ $(q \in \mathbb{Q})$



Outline

Operations

Multi-rate system



Operations: Imported nodes

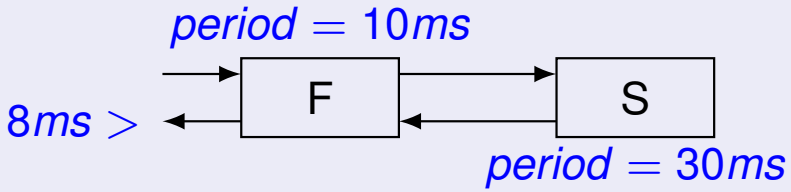
- The operations of the system are declared as **imported nodes**;
- Imported nodes are implemented by **external functions** (for instance in C, or LUSTRE);
- The programmer declares the worst case execution time (**wcet**) of the node.

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



Bertin, V., Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., and Yovine, S. (2001).
TAXYS=Esterel+Kronos. A tool for verifying real-time properties of embedded systems.
In *40th IEEE Conference on Decision and Control*, volume 3.

Curic, A. (2005).
Implementing Lustre programs on distributed platforms with real-time constraints.
PhD thesis, Université Joseph Fourier, Grenoble.

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.

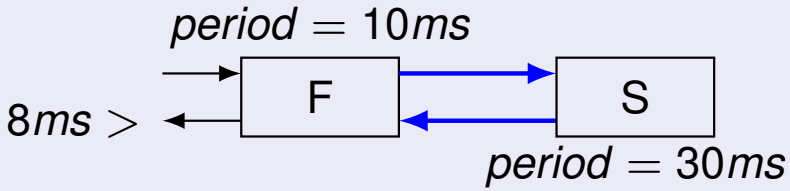
Example

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

The rate of an input/output can be left unspecified, it will be inferred by the compiler.

Multi-rate communications

Multi-rate system



Simulink: User's Guide.
The Mathworks.

Faucou, S., Déplanche, A.-M., and Trinquet, Y. (2004).
An ADL centric approach for the formal design of real-time systems.
In *Architecture Description Language Workshop at IFIP World Computer Congress (WADL'04)*.

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 based on periodic clock transformations:

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

Multi-rate communications: rate transition operators

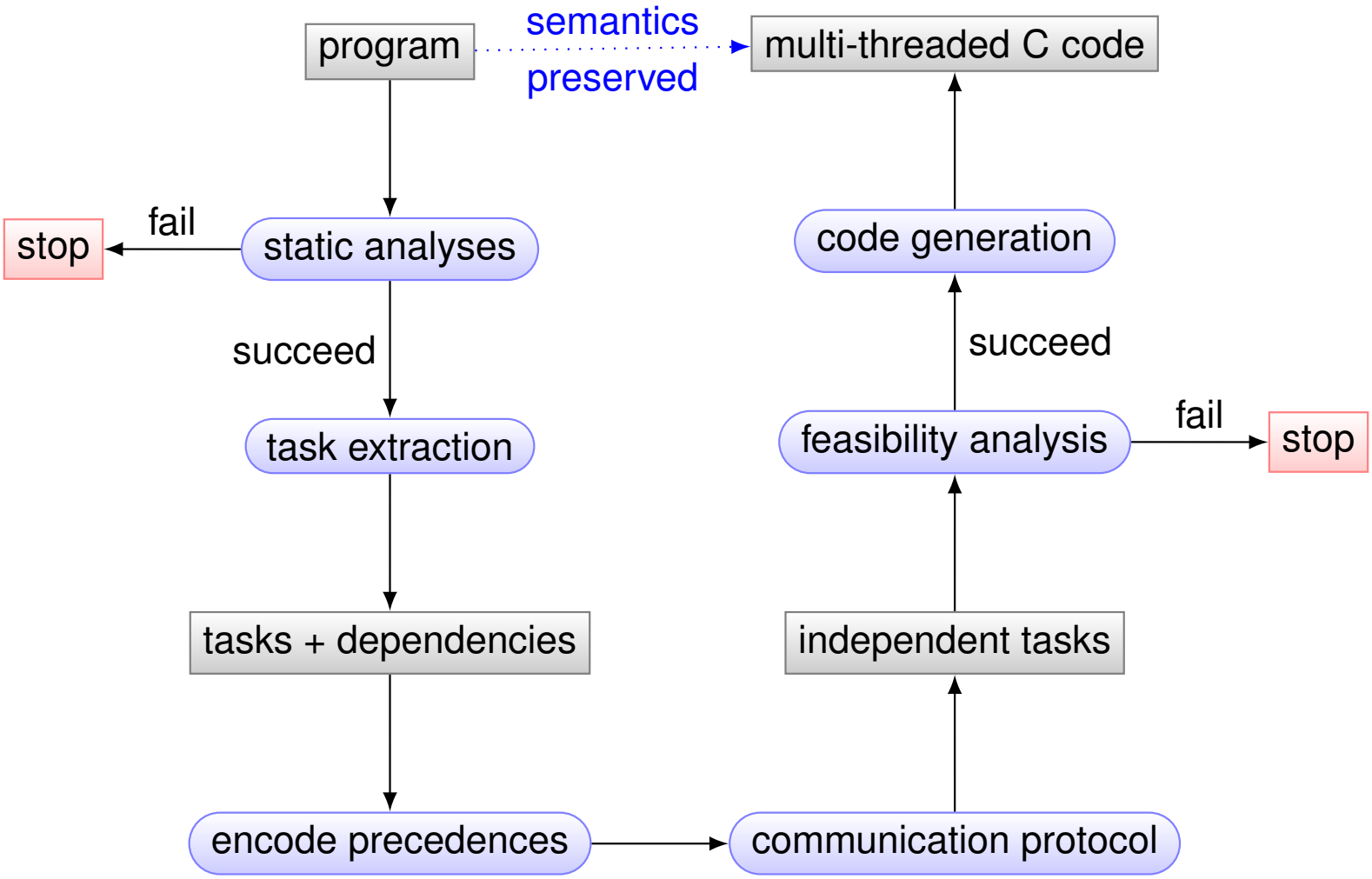
Example

```
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	...

Outline

Compilation overview



Outline

Ensure program correctness

Analyses:

- **Typing**: program only combines flows of the same type $\Rightarrow$ no run-time type error;
- **Causality analysis**: no cyclic data-dependencies $\Rightarrow$ an execution order satisfying all data-dependencies exists;
- **Clock calculus**: program only combines flows of the same clock $\Rightarrow$ no access to ill-defined values (values are only accessed when they should be).

Generate code only if static analyses succeed, ie the semantics of the program is well-defined.

Clock calculus

- Clock calculus = type system;
  - A clock = a type;
  - Flows can be combined only if they have the same ”clock type”;
- Clocks can be **polymorphic** (quantified types);
- Computes the clock of every flow (variable, expression) of the program.

Colaço, J.-L. and Pouzet, M. (2003).
Clocks as first class abstract types.
In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, USA.

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))
```

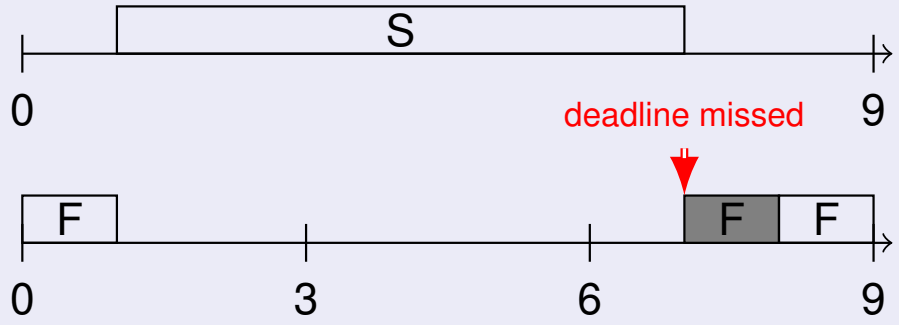
Outline

Task-based vs single-loop compilation

Program

```
imported node F(i: int) returns (o: int) wcet 1;
imported node S(i: int) returns (o: int) wcet 6;
node multi(i: rate(3, 0)) returns (o)
var v;
let v=F(i); o=S(v/^3); tel
```

Single-loop compilation

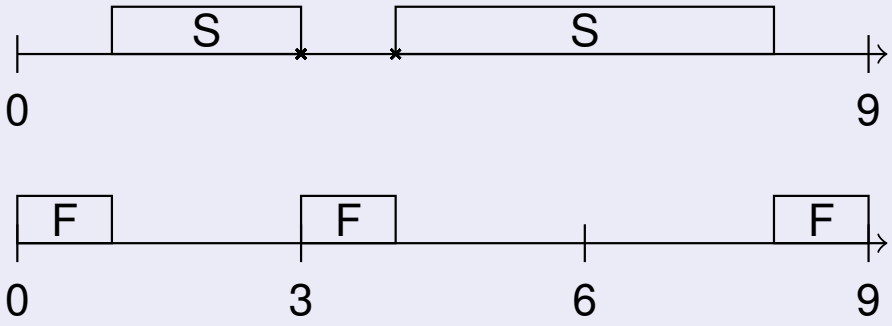


Task-based vs single-loop compilation

Program

```
imported node F(i: int) returns (o: int) wcet 1;
imported node S(i: int) returns (o: int) wcet 6;
node multi(i: rate(3, 0)) returns (o)
var v;
let v=F(i); o=S(v/^3); tel
```

Task-based compilation



Task graph extraction

- Imported node ⇒ **task**;
- Main input ⇒ **sensor** (task);
- Main output ⇒ **actuator** (task);
- Data dependency ⇒ **precedence constraint** between tasks;
- Predefined operator (rate transition, etc) ⇒ precedence annotation, ie **extended precedences**.

Aubry, P., Le Guernic, P., and Machard, S. (1996).
Synchronous distribution of Signal programs.
In *29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*.

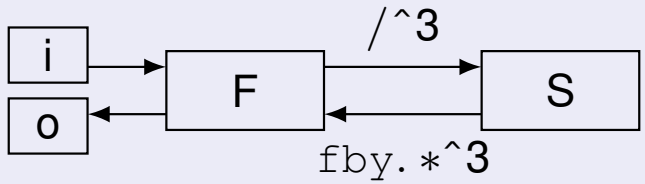
Girault, A., Nicollin, X., and Pouzet, M. (2006).
Automatic rate desynchronization of embedded reactive programs.
ACM Trans. Embedd. Comput. Syst., 5(3).

Task graph extraction: example

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 characteristics

For each task τ_i of clock ck_i:

- **Period**: $T_i = \pi(ck_i)$;
- **Execution time**: C_i as defined for the corresponding imported node.
- **Initial release date**: $r_i = \varphi(ck_i)$.
- **Relative deadline**: $d_i = T_i$ by default, otherwise explicit deadline constraint (eg `o: due 8`).

Scheduling and executing dependent tasks

Two solutions:

1. Rely on synchronization mechanisms:
   - Dependence $\Rightarrow$ semaphore;
   - Several problems:
     - **priority inversion**: solvable with priority affectation protocols;
     - **scheduling anomalies** (system becomes unschedulable due to tasks completing faster than their wcet);
     - Requires to **certify** semaphores implementation.
2. Translate dependent tasks into **independent tasks**.

   $\Rightarrow$ Better-suited for critical systems.

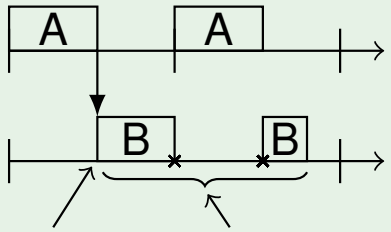
From dependent to independent tasks

Conditions to respect the synchronous semantics:

1. **Data can only be consumed after being produced** (precedence) $\Rightarrow$ precedence encoding, by adjusting real-time attributes;
2. **Data must not be overwritten before being consumed** $\Rightarrow$ communication protocol to keep data available until the deadline of the consumer.

Example

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



(1): $B[0]$ after $A[0]$ (2) keep $A[0]$ available

Precedence encoding, simple precedences

Simple precedences (precedences between tasks of the same period):

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

Chetto, H., Silly, M., and Bouchentouf, T. (1990).
Dynamic scheduling of real-time tasks under precedence constraints.
Real-Time Systems, 2.

Liu, C. L. and Layland, J. W. (1973).
Scheduling algorithms for multiprogramming in a hard-real-time environment.
Journal of the ACM, 20(1).

Adaptation: encoding extended precedences

Extended precedences = repetitive patterns of simple precedences $\Rightarrow$ **encode only one pattern**.

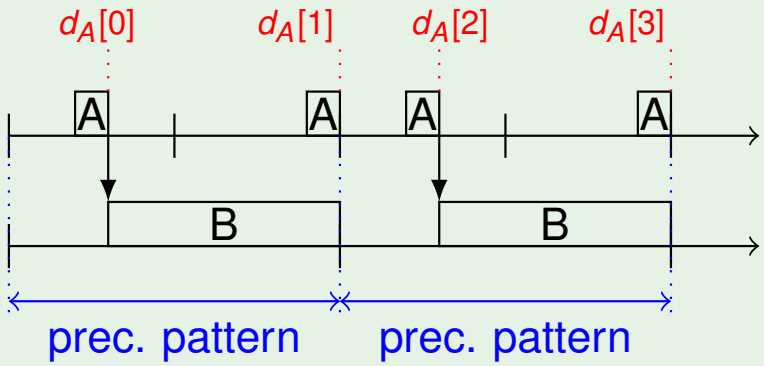
1. **Release dates**: synchronous context $\Rightarrow$ encoding respected by default;
2. $\neq$ relative deadlines for $\neq$ instances of the same task $\Rightarrow$ **deadline words**: $(3.5)^\omega$ = the sequence of task instance deadlines 3.5.3.5.3.5. . . .

Example

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

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

$d_A = (3.5)^\omega$



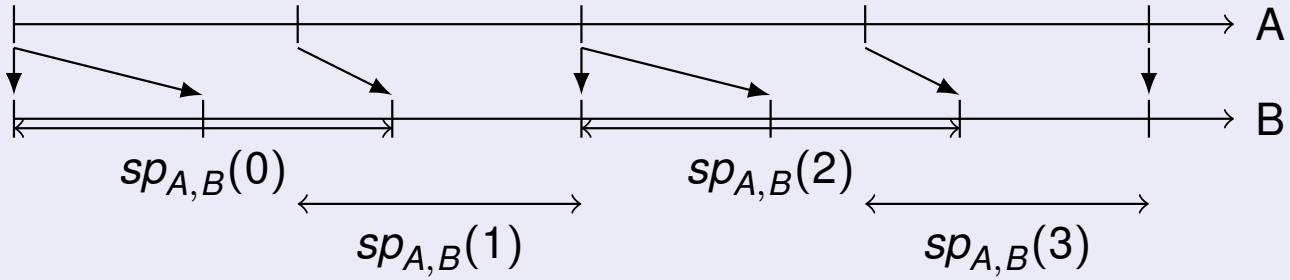
Communication protocol

Ex: `B(A(x)*^3/^2)`, ie $A \xrightarrow{*\hat{}3./\hat{}2} B$:

Semantics

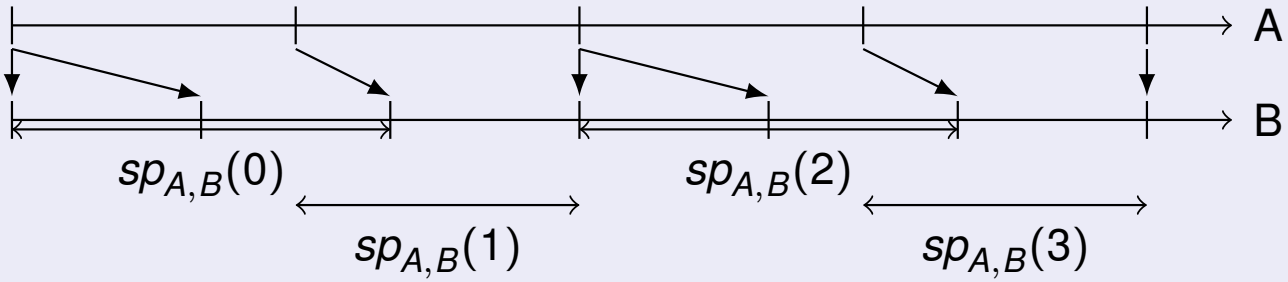
date	0	10	20	30	40	50	60	70	80	...
`A(x)`	a_0			a_1			a_2			...
`A(x)*^3`	a_0	a_0	a_0	a_1	a_1	a_1	a_2	a_2	a_3	...
`A(x)*^3/^2`	a_0		a_0		a_1		a_2		a_3	...

Lifespans



Communication protocol (2)

Lifespans



- Buffer of size 2;
- Write in the buffer cyclicaly;
- Read from the buffer cyclicaly;
- Do not advance at the same pace for reading and writing.

Sofronis, C., Tripakis, S., and Caspi, P. (2006).
A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling.
In *Sixth International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea.

Outline

Prototype

- Task set translated into a C **file**, using **POSIX.13 extensions for real-time** ;
- A task $\Rightarrow$ a **thread**;
- Threads **scheduled concurrently using the EDF policy**, extended to handle deadline words;
- Scheduler prototyped in MARTE OS ;
- **Prototype** of the compiler developed in OCAML, about 3000 lines of code.

POSIX.13 (1998).
IEEE Std. 1003.13-1998. POSIX Realtime Application Support (AEP).
The Institute of Electrical and Electronics Engineers.

Rivas, M. A. and Harbour, M. G. (2002).
POSIX-Compatible Application-Defined Scheduling in MaRTE OS.
In *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, Washington, USA.

Case study

Experiment:

- Prototype used to program the real-time architecture of the ”real” ATV Flight Application Software (2/3 of the periodic services programmed);
- 180 imported nodes, 70 inputs, 9 outputs.

Results:

- Language seems expressive enough;
- Compilation time on a very modest machine is less than 1s, most of it spent to write the output C file.

Outline

Summary (language)

- Real-time architecture description language;
- Synchronous, formal semantics;
- Small set of new primitives;
- Static analyses ensure that the semantics of a program is well-defined.

Summary (compilation)

- Compilation into a set of real-time tasks;
- Tasks of different rates can communicate;
- Precedence encoding allows to schedule tasks as if they were independent;
- Communication protocol preserves the semantics of the program;
- No synchronization mechanisms $\Rightarrow$ no risk of deadlock or priority inversion;
- Complete compilation scheme proved correct formally.

Perspectives

- Use a **static priority scheduling policy** (and reuse the rest): currently under review;
- **Clustering nodes** in the same task to reduce the number of tasks;
- Supporting **mode automata**: makes the clock calculus more complex;
- Ongoing PhD. thesis (M. Cordovilla): compilation for **multi-core architectures** (starting from the task graph).