A Multi-Periodic Synchronous Data-Flow Language

Julien Forget[1] Frédéric Boniol[1] David Lesens[2] Claire Pagetti[1]
firstname.lastname@onera.fr

[1]ONERA - Toulouse, FRANCE

[2]EADS Astrium Space Transportation - Les Mureaux, FRANCE

November 19, 2008

Outline

Implementing Multi-Periodic Reactive Systems

An increasingly complex task:

- Implementing functional aspects.
- Implementing real-time aspects.
- Developing the hardware platform (outside our scope).
- Critical systems: strong determinism required (functional as well as temporal).
- At the same time, optimize latency, hardware cost, etc.

Contribution

We propose:

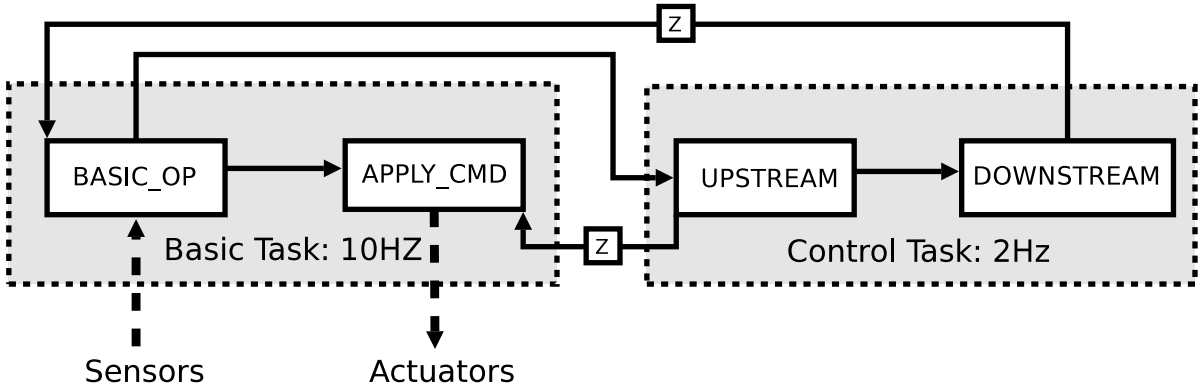
- A high-level, formal language
- With automated code generation (from design to implementation).
- Based on synchronous languages.

This provides:

- High confidence in the generated code.
- Easier design (higher level of abstraction).
- Faster development cycle.

A reactive system : the Automated Transfer Vehicle

- The ATV is the resupplying vehicle for the International Space Station.
- We present a version adapted from the Mission Safing Unit (MSU) of the vehicle developed by EADS Astrium Space Transportation.



Repeat the same behaviour indefinitely: Input-Compute-Output.

Designing the system

1. Design each functional process separately (BASIC_OP, APPLY_CMD, UPSTREAM, DOWNSTREAM).
2. Assemble the processes.

The assembly level:

- Specify the rate of each process.
- Handle inter-process communications: communications must be deterministic.

$\Rightarrow$ Our language focuses on the specification of this assembly level.

Outline

Principles

- Describe the computations performed at each iteration of the system, called instant.
- Each variable or expression is a flow (sequence of values).
- Flows are activated/deactivated using clocks (Boolean conditions). Clocks define the temporal behaviour of a process.
- Only synchronous flows can be combined, ie flows present at the same instants.
- Flows are defined by equations.
- Equations are structured hierarchically into nodes.
- The main node is activated by an external program that repeats the classic reactive loop (usually periodically):
  1. provide inputs from sensors to the main node;
  2. execute the node;
  3. transfer the outputs of the node to the actuators.

Operations on flows

Example

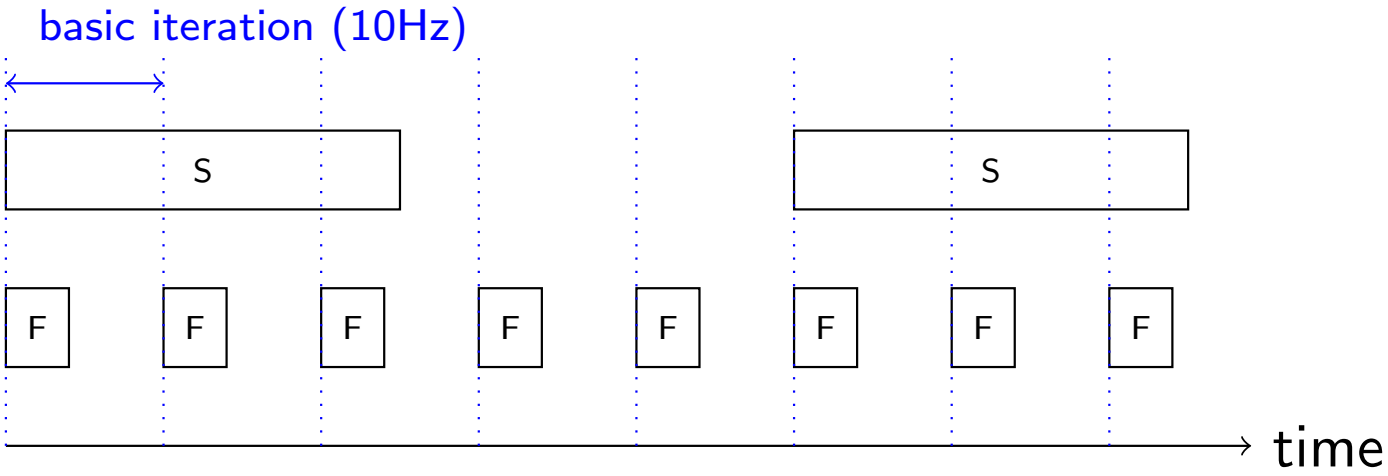
```
node N(i,j:int) returns (o:int)
let
  c=(i=j);
  y=i when c;
  z=0 fby y;
tel
```

Behaviour:

i	2	3	1	5	6	...
j	2	3	1	7	4	...
c	*True*	*True*	*True*	*False*	*False*	...
y	2	3	1			...
z	0	2	3			...

Basic iteration in Multi-Periodic Systems

Programming a reactive system = program an iteration of the process and repeat it indefinitely always at the same base rate.



Basic iteration: F + some part of S

Implementing the MSU

```
node msu(fromEnv: int) returns (toEnv:  int)
var clock0,clock1,clock2, clock3, clock4 : bool;
    count,  bop1,  bop2: int;
    us_0: int when clock0;
    us1, us2: int when clock1;
    ds0: int when clock2;
    ds:  int when clock3;
let
  count=countN(5);
  clock0=(count=0); clock1=(count=1); ...

  -- fast tasks
  bop1, bop2=basicOp(fromEnv,current(0 fby ds));
  toEnv=applyCmd(current(0 fby us1),bop1);

  -- slow tasks: split computations between successive instants
  us_0=upStream0(bop2 when clock0);
  us1, us2=upStream1(current(us_0) when clock1);
  ds0=downStream(current(us2) when clock2);
  ds=downStream(current(ds) when clock3);
tel
```

Problem: Manual Scheduling

- Slow operations have to be manually split into several nodes.
- Difficult to distribute the slow processes fairly between successive iterations, in terms of execution times.
- Splitting operations may be difficult due to the software architecture.

$\Rightarrow$ We define a language that enables automated scheduling.

Outline

Strictly Periodic Clocks

- Flow: $(v_i, t_i)_{i \in \mathbb{N}}$. v_i: a value in the set of values $\mathcal{V}$. t_i: a tag in $\mathbb{N}^+$. For all i, $t_i < t_{i+1}$.
- Clock of a flow: its projection on $\mathbb{N}^+$.
- v_i must be produced between t_i and t_{i+1}.

Definition

Clock $h = (t_i)_{i \in \mathbb{N}^+}$, $t_i \in \mathbb{N}^+$, is strictly periodic if and only if:

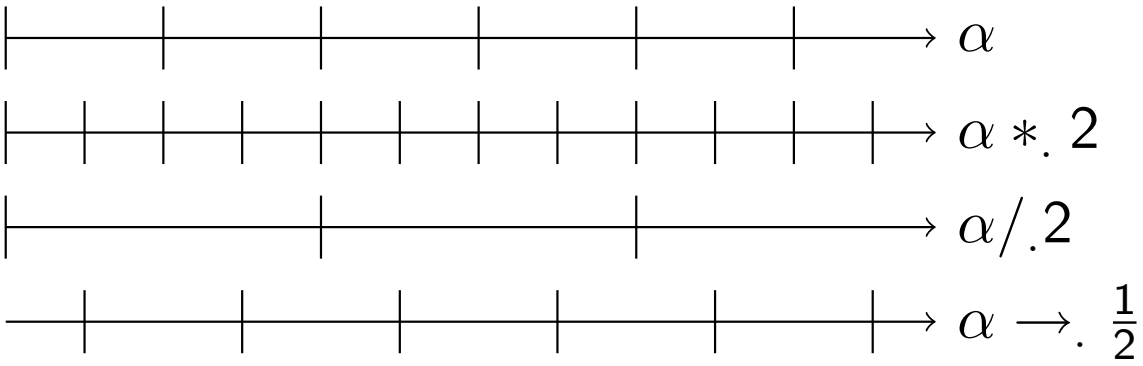
$$\exists n \in \mathbb{N}^{+*}, \forall i \in \mathbb{N}, \; t_{i+1} - t_i = n$$

- $\pi(h) = n$: the period of h. $\varphi(h) = t_0$: the phase of h.
- (n, p): the clock α such that $\pi(\alpha) = n$ and $\varphi(\alpha) = \pi(\alpha) * p$ $(p \in \mathbb{Q}^+)$.

Periodic clock transformations

Transformations that produce new strictly periodic clocks:

- 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: $\pi(\alpha \rightarrow. q) = \pi(\alpha),\ \varphi(\alpha \rightarrow. q) = \varphi(\alpha) + q * \pi(\alpha)$ $(q \in \mathbb{Q})$



Why a new class of clocks ?

1. To clearly separate two complementary notions:
   - A strictly periodic clock defines the real-time rate of a flow.
   - A Boolean clock specifies on this rate the activation condition of the flow.
2. Strictly periodic clocks and their transformations are statically evaluable.
   - This is mandatory to enable efficient scheduling.
   - Boolean clocks can emulate strictly periodic clocks but they are not statically evaluable.

Strictly periodic clocks do not replace Boolean clocks, they complement them.

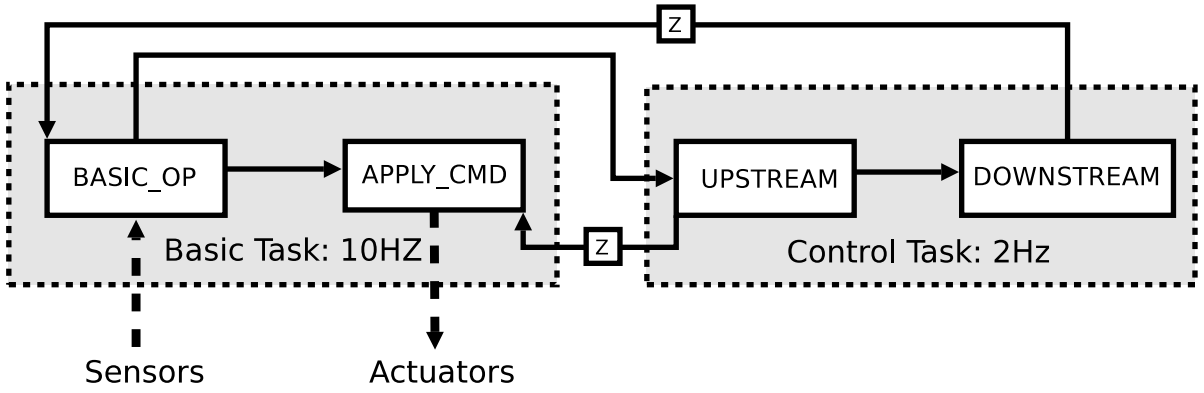
Operators based on strictly periodic clocks

If the flow x has clock α:

- x$*$^k has clock $\alpha *_. k$.
- x/^k has clock $\alpha /_. k$.
- x$\sim>$q has clock $\alpha \rightarrow_. q$.

tag	0	2	4	6	8	...
x	x_1		x_2		x_3	...
$x *\hat{}2$	x_1	x_1	x_2	x_2	x_3	...
$x/\hat{}2$	x_1				x_3	...
$x \sim> 1/2$		x_1		x_2		...

Reminder: the Mission Safing Unit



Repeat the same behaviour indefinitely: Input-Compute-Output.

Programming the MSU: Step 1

Define each "functional" node:

```
imported node basicOp(i,j) returns (o,p);
imported node A(i) returns (o); ...
wcet basicOp=40; wcet applyCmd=20; wcet A=30;
wcet B=10; wcet C=20; wcet D=40;wcet E=10; wcet F=30;

node upStream(i) returns (o1,o2)
let
  o1=A(B(i)); o2=C(i);
tel
node downStream(i) returns (o)
let
  o=D(E(F(i)));
tel
```

Programming the MSU: Step 2

Assemble the functional nodes:

```
-- assembling nodes
node msu(fromEnv) returns (toEnv)
var bop1,bop2,us1,us2,ds;
let
  bop1,bop2=basicOp(fromEnv,(0 fby ds)*^5);
  toEnv=applyCmd((0 fby us1)*^5,bop1);
  us1, us2=upstream(bop2/^5);
  ds=downStream(us2);
tel

-- optional level: clock instanciation + activation condition
node main(c,fromEnv: rate (100,0))
  returns (toEnv: rate (100,0) when c)
let
  toEnv,toOtherMSU=(msu(fromEnv,otherMSU)) when c;
tel
```

Outline

Ensuring program correction

Static analysis:

- Typing: the program only combines values of the same type.
- Causality analysis: no loop in the data-dependencies.
- Initialisation analysis: included in the clock calculus in our case.
- Clock calculus: the program does not access to undefined values.
- Scheduling: the program respects its real-time constraints.

Only then: generate the code corresponding to the program.

The Clock Calculus: checking program synchronism

- An expression is well-synchronized if it does not access to undefined values.
- The role of the clock calculus is to verify that a program only uses well-synchronized expressions.
- Well-synchronized programs cannot go wrong: if the program is well-synchronized then its semantics are well-defined.

The clock calculus on strictly periodic clocks can be implemented as a type system with simple sub-typing constraints.

Scheduling: from a synchronous program to a set of real-time tasks.

- Transform the program into a set of tasks.
- Compute the real-time characteristics of each task.
- Schedule the resulting set of tasks.

Obtaining tasks:

- Tasks=imported nodes.
- Precedences=data dependencies.

Let ck_i be the clock of task τ_i. $pparent(ck_i)$ denotes the closest strictly periodic clock parent of ck_i (in case ck_i is Boolean).

- $T_i = \pi(pparent(ck_i))$
- $r_i = \varphi(pparent(ck_i))$
- C_i is known from the node wcet declaration.
- $d_i = T_i$.

Scheduling multi-periodic dependent tasks

Problem: Few scheduling algorithms support multi-periodic tasks related by precedence constraints.

Solution (ongoing work):

- Automatically encode precedences in the real-time attributes of the tasks (Chetto90).
- Use an EDF scheduler.
- The use of preemptions avoids to manually split the slow task into several sub-tasks.

Outline

Conclusion

The language:

- Provides a high level of abstraction.
- Enables flexible description of multi-rate communicating systems.
- Provides automatic code generation, the correction of which is proved formally.

Main benefits:

- Avoids manual scheduling (vs classic synchronous languages).
- Prevents non-deterministic communications (vs asynchronous languages).

Future work: define the scheduling of a program.