A Multi-Periodic Synchronous Data-Flow Language

Julien Forget (speaker)$^1$ 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 13, 2008
1 Context

2 Synchronous Languages

3 A Multi-Periodic Synchronous Language

4 Implementation
Outline

1. Context
2. Synchronous Languages
3. A Multi-Periodic Synchronous Language
4. Implementation
Implementing Multi-Periodic Reactive Systems

An increasingly complex task:

- Implementing functional aspects.
- Implementing real-time aspects.
- Developing the hardware platform (not covered here).
- Critical systems: strong determinism required (functional as well as temporal).
- At the same time, optimize latency, hardware cost, etc.
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 functional aspects of each 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.

⇒ Our language focuses on the specification of this assembly level.
Outline

1. Context
2. Synchronous Languages
3. A Multi-Periodic Synchronous Language
4. Implementation
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, i.e., 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

| Operation       | x1  | x2  | x3  | x4  | x5  | ...
|-----------------|-----|-----|-----|-----|-----|-----
| Addition        | +y1 | +y2 | +y3 | +y4 | +y5 | ...
| Multiplication  | y1  | y2  | y3  | y4  | y5  | ...
| z=x when h      | x1  |     |     | x4  |     | x5  | ...
| t=y when not h  |     | y2  | y3  |     |     |     | ...
| current(z)      | x1  | x1  | x1  | x4  | x5  | ...
Example: counter with reset

Code

```plaintext
node counter(reset: bool) returns (count: int)
let
  count = if reset then 0
  else (0 fby (count + 1));
tel
```

Behaviour

<table>
<thead>
<tr>
<th>reset</th>
<th>False</th>
<th>False</th>
<th>False</th>
<th>True</th>
<th>False</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>
Instants 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 (10Hz)

\[ F + \text{some part of } S \]
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.

⇒ This manual scheduling should be automated.
Outline

1. Context
2. Synchronous Languages
3. A Multi-Periodic Synchronous Language
4. Implementation
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 \(\alpha\) such that \(\pi(\alpha) = n\) and \(\varphi(\alpha) = \pi(\alpha) \ast p\) (\(p \in \mathbb{Q}^+\)).
Periodic clock transformations

Transformations that produce new strictly periodic clocks:

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

\[
\begin{align*}
\begin{array}{c}
| |
| |
| |
| |
| |
| |
| |
\end{array} & \rightarrow \alpha \\
\begin{array}{c}
| |
| |
| |
| |
| |
| |
| |
\end{array} & \rightarrow \alpha \cdot . 2 \\
\begin{array}{c}
| |
| |
| |
| |
| |
\end{array} & \rightarrow \alpha / . 2 \\
\begin{array}{c}
| |
| |
| |
| |
| |
| |
\end{array} & \rightarrow \alpha \rightarrow . \frac{1}{2}
\end{align*}
\]
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 $\alpha$:

- $x^\wedge k$ has clock $\alpha \cdot k$.
- $x/\wedge k$ has clock $\alpha / . k$.
- $x \sim > q$ has clock $\alpha \rightarrow . q$.

<table>
<thead>
<tr>
<th>tag</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x^\wedge 2$</td>
<td>$x_1$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>...</td>
</tr>
<tr>
<td>$x/\wedge 2$</td>
<td>$x_1$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$x \sim &gt; 1/2$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reminder: the Mission Safing Unit

Repeat the same behaviour indefinitely: Input-Compute-Output.
Programming the MSU: Step 1

Define each “functional” node:

```plaintext
imported node basicOp(i, j) returns (o, p);
imported node A(i) returns (o); ...
wct basicOp=40; wct applyCmd=20; wct A=30;
wct B=10; wct C=20; wct D=40; wct E=10; wct 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 instantiation + 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
“Queuing” communications

The communication scheme presented for the MSU relies on sampling the data produced by fast operations. We can use a different scheme based on queuing with Lustre arrays.

```plaintext
node store_n(i, n, init) returns (A: int^n)
let
  A[0] = i;
  A[1..(n-1)] = (init ^^ (n-1)) fby (A[0..(n-2)]);
tel

node split(i, n) returns (o)
var ifast;
let
  ifast = i * ^^ n;
  o = ifast [countN n];
tel

node join(i, n, init) returns (o)
let
  ofast = store_n(i, n, init);
  o = ofast / ^^ n;
tel
```
The MSU, sampling vs queuing

Queuing:

```plaintext
node msu_queuing(fromEnv) returns (toEnv)
var  bop1, bop2, us1, us2, ds;
let
  bop1, bop2=basicOp(fromEnv, split((0 fby ds),5,0));
  toEnv=applyCmd(split((0 fby us1),5,0), bop1);
  us1, us2=upstream(join(bop2,5));
  ds=downStream(us2);
```

tel

Sampling:

```plaintext
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
Outline

1. Context
2. Synchronous Languages
3. A Multi-Periodic Synchronous Language
4. Implementation
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 $\tau_i$. $pparent(ck_i)$ denotes the closest strictly periodic clock parent of $ck_i$ (in case $ck_i$ is Boolean).

- $T_i = \pi(pparent(ck_i))$
- $r_i = \phi(pparent(ck_i))$
- $C_i$ 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).
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.