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

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

⇒ feasibility analysis and efficient preemptive scheduling.
Outline

1 Introduction
2 Language
   - Synchronous real-time
   - Language primitives
3 Compilation
   - Static analyses
   - Multi-task compilation
4 Scheduling
   - Dealing with precedence constraints
   - Multicore
5 Conclusion

Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
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:
  \[
  \exists n \in \mathbb{N}^+, \forall i \in \mathbb{N}, \quad t_{i+1} - t_i = n
  \]
- \( \pi(ck) = n \) 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;
• \( \alpha \times k \): multiply frequency;
• \( \alpha \rightarrow q \): offset activations.

\[
\begin{align*}
\frac{\alpha}{2} & \rightarrow \alpha \\
\frac{\alpha}{2} & \rightarrow \alpha \times 2 \\
\frac{\alpha}{2} & \rightarrow \alpha \rightarrow \frac{1}{2}
\end{align*}
\]
Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
Multi-rate system

\[ 8ms > \frac{10}{3} \] ms

**Operations**

Periods:
- \( F \): period = 10 ms
- \( S \): period = 30 ms

---

Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
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 $F(i, j : \text{int})$ returns $(o, p : \text{int})$ wcet 2;
imported node $S(i : \text{int})$ returns $(o : \text{int})$ wcet 10;
Real-time constraints

Multi-rate system

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

\[ \text{period} = 30\,\text{ms} \]
Real-time constraints are specified in the signature of a node; 
Periodicity constraints on inputs/outputs; 
Deadline constraints on inputs/outputs.

**Example**

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

Input/output can be unspecified, the compiler will infer it.
Multi-rate system

$\text{F}$

$\text{S}$

$8ms > \text{period} = 10ms$

$\text{period} = 30ms$
Multi-rate communications: rate transition operators

Example

```plaintext
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: \( x / ^3 \) (has \( clock(x) / .3 \));
- Over-sampling: \( x * ^3 \) (has \( clock(x) * .3 \)).
Multi-rate communications: rate transition operators

**Example**

```plaintext
node sampling(i: rate (10, 0)) returns (o)
    var vf, vs;
let
    (o, vf)=F(i, (0 fby vs) * ^3);
    vs=S(vf / ^3);
```

<table>
<thead>
<tr>
<th>date</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>vf</td>
<td>vf₀</td>
<td>vf₁</td>
<td>vf₂</td>
<td>vf₃</td>
<td>vf₄</td>
<td>vf₅</td>
<td>vf₆</td>
<td>vf₇</td>
<td>vf₈</td>
</tr>
<tr>
<td>vf/³</td>
<td>vf₀</td>
<td></td>
<td></td>
<td>vf₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vs</td>
<td>vs₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 fby vs</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0 fby vs) * ³</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>vs₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
Formal semantics (just a little)

\[ \text{fby} \ (v, (v', t)).s \ = \ (v, t). \text{fby} \ (v', s) \]

\[ \text{when} \ #((v, t).s, (true, t).cs) = (v, t). \text{when} \ #((s, cs)) \]

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

\[ \text{merge} \ #((true, t).s, (v, t).s_1, s_2)) = (v, t).\text{merge} \ #((s, s_1, s_2)) \]

\[ \text{merge} \ #((false, t).s, s_1, (v, t).s_2)) = (v, t).\text{merge} \ #((s, s_1, s_2)) \]
Formal semantics (just a little more)

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

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

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

Language
- Synchronous real-time
- Language primitives

Compilation
- Static analyses
- Multi-task compilation

Scheduling
- Dealing with precedence constraints
- Multicore

Conclusion
Static analyses

- Typing;
- Causality analysis;
- Clock calculus;

**Feasibility analysis**: check that all the deadlines will be met.
Clock calculus: example

Example

\begin{verbatim}
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
\end{verbatim}

Result inferred by the clock calculus

\begin{verbatim}
under_sample: 'a->'a/\cdot2
poly: ((10,0) * (5,0)) -> ((20,0) * (10,0))
\end{verbatim}
Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
Objective: translate the program into the “standard” task model to benefit from real-time scheduling theory.

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

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

**Program**

```plaintext
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**

![Task graph diagram]
Real-time attributes

For each task $\tau_i$ of clock $ck_i$:

- Period: $T_i = \pi(ck_i)$;
- Execution time: declared in the program;
- Initial release date: $O_i = \phi(ck_i)$.

Relative deadline: explicit constraint (eg $\circ: \text{due} \ 8$), 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

\[ A \xrightarrow{^2} B: \]

(1): \( B_0 \) after \( A_0 \)

(2) keep \( A_0 \) available
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.
Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
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 $\tau_i \rightarrow \tau_j$:
   
   $D_i^* = \min(D_i, \min_{\tau_j \in \text{succs}(\tau_i)} (D_j^* - C_j))$
   
   $R_j^* = \max(R_j, \max_{\tau_i \in \text{preds}(\tau_j)} (R_i^*))$

3. Resulting problem $\Leftrightarrow$ Original problem;
4. **Optimal** policy (finds a solution if there exists one).

---

Extended precedence constraints

For a pair of related tasks:
- Only a subset of the task instances are constrained;
- Constraints follow a repetitive pattern
  \[ \Rightarrow \text{Encode the pattern of task instance constraints.} \]

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**

\[ A \xrightarrow{2} B \]

\[ T_A = 5, \ T_B = 10, \ C_B = 7, \ C_A = 1 \]

\[ d_A = (3.5)^\omega \]
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

1. Introduction
2. Language
   - Synchronous real-time
   - Language primitives
3. Compilation
   - Static analyses
   - Multi-task compilation
4. Scheduling
   - Dealing with precedence constraints
   - Multicore
5. Conclusion

Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
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 (≃ OS for multi-core execution).

http://sites.onera.fr/schedmcore/
Programming Critical Embedded Systems with Multiple Real-Time Constraints with the PRELUDE language
C code generation (OS independent);

Multi-threaded execution (OS dependent):
- SCHEDM_CORE;
- MARTE OS (Cantabria) broken.

Compiler developed in OCAML \( \sim 3000 \) lines (sources available).

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

(See the website for bibliographic references).
Cluster nodes inside tasks to reduce the number of tasks;

Support **mode automata** (part of the thesis of Remy WYSS):

- Makes the clock calculus more complex;
- Conditional scheduling required.

A new kind of **temporal verification** ?