Synchronous semantics of multi-mode multi-periodic systems

Frédéric Fort and Julien Forget
firstname.lastname@univ-lille.fr

November 22-26, 2021

Centre de Recherche en Informatique, Signal et Automatique de Lille

Problem statement

Goal : Program real-time multi-mode systems

1. Extending the semantics of a synchronous language
2. Static analysis (clock calculus) to guarantee soundness
3. Allow different mode change protocols

Multi-mode real-time systems

- Change func. requirements during execution
- Mode = task set
- Mode change protocol
  - Switch from task set $\mathcal{T}$ to task set $\mathcal{T}'$
- How to transition between $\mathcal{T}$ and $\mathcal{T}'$?
- Metrics observed by the scheduling community :
  - Promptness
  - Schedulability

Real, and Crespo. *Mode change protocols for real-time systems : A survey and a new proposal.* 2004.

Multi-mode real-time systems

- Change func. requirements during execution
- Mode = task set
- Mode change protocol
  - Switch from task set $\mathcal{T}$ to task set $\mathcal{T}'$
- How to transition between $\mathcal{T}$ and $\mathcal{T}'$?
- Metrics observed by the scheduling community :
  - Promptness
  - Schedulability
- Semantics ?

Real, and Crespo. *Mode change protocols for real-time systems : A survey and a new proposal.* 2004.

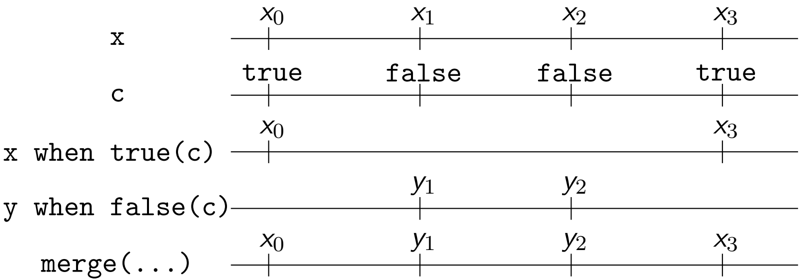
Synchronous state machines

- Formally defined language for multi-mode systems
- Based on Lustre and Lucid Synchrone
- Transpiles state machines into `when` and `merge`

Colaço, Pagano, and Pouzet. *A conservative extension of synchronous data-flow with state machines.* 2005.

Synchronous state machines

Reminder : `when` and `merge`



Synchronous state machines

Transpilation process

```
automaton
| S1 ->
  unless c then S2;
  o = i;
| S2 ->
  unless c then S1;
  o = j;
end
```

```
ps = S1 fby s;
s = merge(ps,
  S1->if c when S1(ps)
    then S2 else S1,
  S2->if c when S2(ps)
    then S1 else S2);
o = merge(s,
  S1->i when S1(s),
  S2->j when S2(s));
```

Synchronous state machines
Limitations

Problem

1. No explicit time constraints
2. All flows within the automaton must share the same clock

$\Rightarrow$

Solution

Synchronous state machines
Limitations

Problem

1. No explicit time constraints
2. All flows within the automaton must share the same clock

$\Rightarrow$

Solution

1. Prelude

Synchronous state machines
Limitations

Problem

1. No explicit time constraints
2. All flows within the automaton must share the same clock

$\Rightarrow$

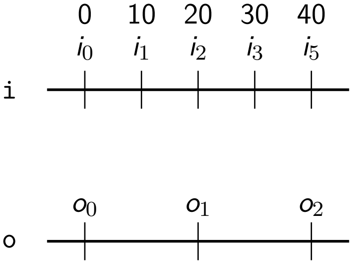
Solution

1. PRELUDE
2. Extension of PRELUDE

Prelude

- Lustre-like synchronous dataflow language
- Explicit real-time constraints

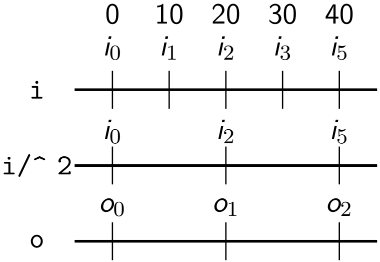
```
node main(i: int rate (10,0))
returns (o: int rate (20,0))
let
  o = f(i ???);
tel
```



Prelude

- Lustre-like synchronous dataflow language
- Explicit real-time constraints
- Built-in *rate-transition operators*

```
node main(i: int rate (10,0))
returns (o: int rate (20,0))
let
  o = f(i/^2);
tel
```



Prelude

- Lustre-like synchronous dataflow language
- Explicit real-time constraints
- Built-in *rate-transition operators*
- `when` and `merge` inherit the same definition as Lustre

State Machines in Prelude

Naive approach

```
i: rate(5,0) j: rate(7,0)
c: rate(11,0)
automaton
| S1 ->
  unless c then S2;
  o = f1(i);
  p = g1(j);
| S2 ->
  unless c then S1;
  o = f2(i);
  p = g2(j);
end
```

```
ps = S1 fby s;
s = merge(ps, ...);
o = merge(s,
  S1->f1(i when S1(s)),
  S2->f2(i when S2(s)));
p = merge(s,
  S1->g1(j when S1(s)),
  S2->g2(j when S2(s)));
```

State Machines in Prelude

Naive approach

```
i: rate(5,0) j: rate(7,0)
c: rate(11,0)
automaton
| S1 ->
  unless c then S2;
  o = f1(i);
  p = g1(j);
| S2 ->
  unless c then S1;
  o = f2(i);
  p = g2(j);
end
```

```
ps = S1 fby s;
s = merge(ps, ...);
o = merge(s,
  S1->f1(i when S1(s)),
  S2->f2(i when S2(s)));
p = merge(s,
  S1->g1(j when S1(s)),
  S2->g2(j when S2(s)));
```

s must be synchronous with both i and j

State Machines in Prelude

Further issues with rate-transition operators

`i: rate (3,0)` $\Rightarrow (3,0)$

`i *^3` $\Rightarrow (1,0)$

`i /^2*^3` $\Rightarrow (2,0)$

`c: rate (3,0)` $\Rightarrow (3,0)$

`i /^2` $\Rightarrow (6,0)$

`i *^3/^2` $\Rightarrow (2,0)$

`i when true(c)` $\Rightarrow (3,0)$ **on** $\mathit{true}(c)$

`i /^2 *^3` and `i *^3 /^2` don't produce the same values, but are synchronous (produce the values at the same instants)

State Machines in PRELUDE

Further issues with rate-transition operators

`i: rate (3,0)` $\Rightarrow (3,0)$

`i *^3` $\Rightarrow (1,0)$

`i /^2*^3` $\Rightarrow (2,0)$

`c: rate (3,0)` $\Rightarrow (3,0)$

`i /^2` $\Rightarrow (6,0)$

`i *^3/^2` $\Rightarrow (2,0)$

`i when true(c)` $\Rightarrow (3,0)$ **on** *true*(c)

`i when true(c) *^3 /^2` $\Rightarrow (2,0)$ **on** *true*(c)

`i when true(c) /^2 *^3` $\Rightarrow (2,0)$ **on** *true*(c)

But the expressions `i when true(c) *^3 /^2` and `i when true(c) /^2 *^3` aren't synchronous !

State Machines in Prelude

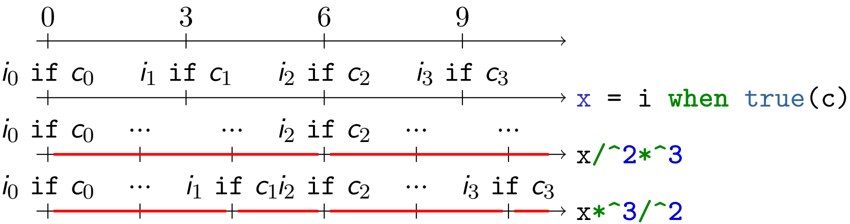
Further issues with rate-transition operators

`i: rate (3,0)` $\Rightarrow (3,0)$ `c: rate (3,0)` $\Rightarrow (3,0)$

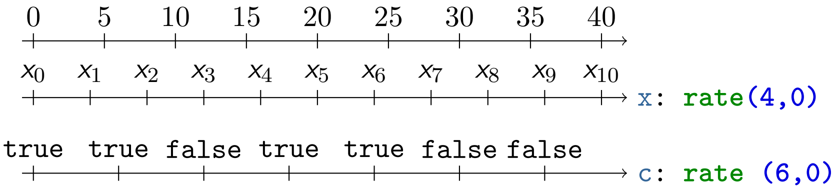
`i when true(c)` $\Rightarrow (3,0)$ **on** $\mathit{true}(c)$

`i when true(c) *^3 /^2` $\Rightarrow (2,0)$ **on** $\mathit{true}(c)$

`i when true(c) /^2 *^3` $\Rightarrow (2,0)$ **on** $\mathit{true}(c)$

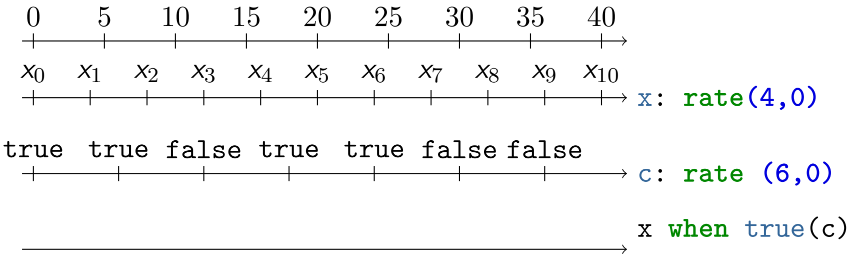


Clock views
Overview



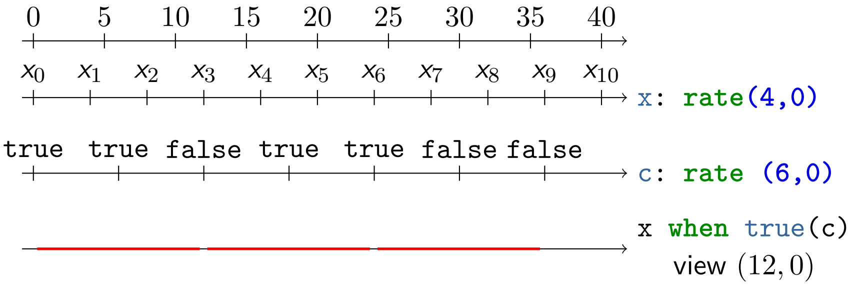
Clock views

Overview



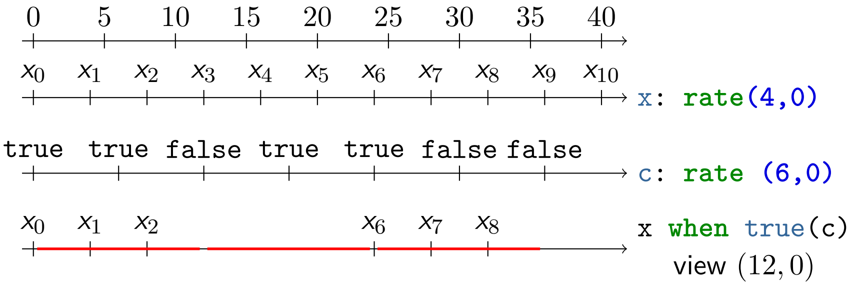
Clock views

Overview



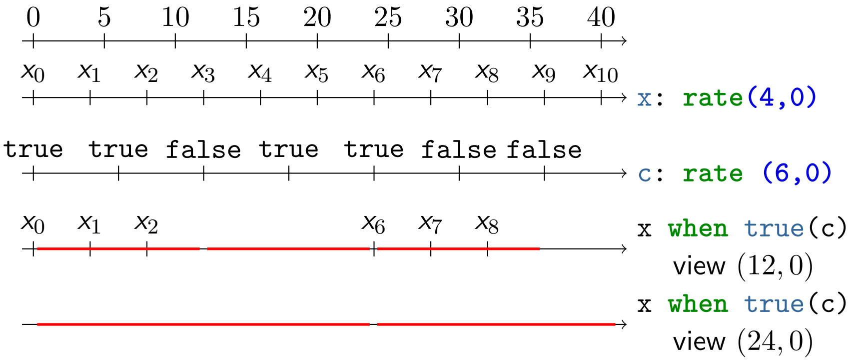
Clock views

Overview

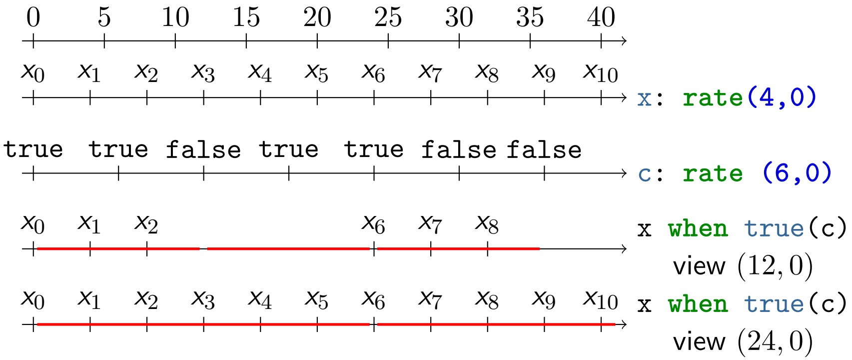


Clock views

Overview



Clock views
Overview



Formal clock semantics
Notation

- Tagged-signal model
  - Clocks define a set of tags (instants)
  - Dataflows define a set of tag-value pairs
- $t \in ck \Leftrightarrow$ ck is present at instant t
- $\pi(ck)$: period $\varphi(ck)$: offset
- Strictly periodic clock $(n, p) = \{p + i * n \mid i \in \mathbb{N}\}$

Formal clock semantics

Clock views

- Conditionally sub-sampled clock $ck \ \mathbf{on} \ C(c, w)$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Formal clock semantics

Clock views

- Conditionally sub-sampled clock ck **on** $C(c, w)$

  $w = (n, p)$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Formal clock semantics
Clock views

- Conditionally sub-sampled clock ck **on** $C(c, w)$

  $w = (n, p)$

  $\{t \mid t \in ck,$

  $\}$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Formal clock semantics
Clock views

- Conditionally sub-sampled clock ck **on** $C(c, w)$

  $w = (n, p)$

  $\{t \mid t \in ck, \exists(C, t'') \in c,$

  $\}$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Formal clock semantics
Clock views

- Conditionally sub-sampled clock ck **on** $C(c, w)$

  $w = (n, p)$

  $\{t \mid t \in ck, \exists(C, t'') \in c, \exists t' \in w,$

  $\}$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Formal clock semantics
Clock views

- Conditionally sub-sampled clock ck **on** $C(c, w)$

  $w = (n, p)$

  $$\{t \mid t \in ck, \exists (C, t'') \in c, \exists t' \in w, \quad t' \leq t < t' + n, \quad \}$$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Formal clock semantics

Clock views

- Conditionally sub-sampled clock ck **on** $C(c, w)$

  $w = (n, p)$

  $$\{t \mid t \in ck, \exists (C, t'') \in c, \exists t' \in w,$$
  $$t' \leq t < t' + n, t'' = t' + \varphi(c) - p\}$$

  - ck : Sub-sampled clock
  - C : Condition value
  - c : Condition dataflow
  - w : View

Clock calculus

- Need to verify the clock consistency
- Compute clock views

⇒ Clock calculus : Dedicated type system

Refinement typing

- Extends an existing type system
- *Refine* types with predicates (in a decidable logic)
- SMT solvers are used to verify those predicates

$\{\nu{:}b \mid r\}$

The *base type* b (e.g. `int`, `int list`) refined by the boolean predicate r (e.g. $\nu \geq 0 \land \nu < x$) such that r is true for all values inhabiting $\{\nu{:}b \mid r\}$. The variable ν represents the value of the typed expression.

Refinement typing

- Extends an existing type system
- *Refine* types with predicates (in a decidable logic)
- SMT solvers are used to verify those predicates

4: `int`

`mod`: `int` $\rightarrow$ `int` $\rightarrow$ `int`

`static_assert`: `bool` $\rightarrow$ `unit`

Refinement typing

- Extends an existing type system
- *Refine* types with predicates (in a decidable logic)
- SMT solvers are used to verify those predicates

$$4\text{: } \{\nu : \texttt{int} | \nu = 4\}$$

$$\texttt{mod}\text{: } \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$$

$$\texttt{static\_assert}\text{: } \texttt{bool} \rightarrow \texttt{unit}$$

Refinement typing

- Extends an existing type system
- *Refine* types with predicates (in a decidable logic)
- SMT solvers are used to verify those predicates

$$4\text{: } \{\nu : \texttt{int} | \nu = 4\}$$

$$\texttt{mod}\text{: } \texttt{x:int} \rightarrow \texttt{y:}\{\nu : \texttt{int} \mid \nu > 0\} \rightarrow \{\nu : \texttt{int} \mid \nu = \texttt{x\%y}\}$$

$$\texttt{static\_assert}\text{: } \texttt{bool} \rightarrow \texttt{unit}$$

Refinement typing

- Extends an existing type system
- *Refine* types with predicates (in a decidable logic)
- SMT solvers are used to verify those predicates

$$4\text{: } \{\nu : \texttt{int} | \nu = 4\}$$

$$\texttt{mod}\text{: } \texttt{x}\text{:}\texttt{int} \rightarrow \texttt{y}\text{:}\{\nu : \texttt{int} \mid \nu > 0\} \rightarrow \{\nu : \texttt{int} \mid \nu = \texttt{x}\%\texttt{y}\}$$

$$\texttt{static\_assert}\text{: } \texttt{b}\text{:}\{\nu : \texttt{bool} \mid \nu\} \rightarrow \texttt{unit}$$

Refinement clock calculus

- Clocks $\neq$ clock types
- Clock type : Reconstruction of the clock by the type system
- Base clocks ck_b
  - pck : A strictly periodic clock ((n, p) with n and p unknown)
  - ck_b **on** $C(c, w)$: Application of **on** $C(c, w)$ to ck_b
- Refinements relate to the period and offset of the clock

`i: rate (10,5)` $\Rightarrow \{\nu{:}\texttt{pck} \mid \pi(\nu) = 10 \wedge \varphi(\nu) = 5\}$

Refinement clock calculus

- Clocks $\neq$ clock types
- Clock type : Reconstruction of the clock by the type system
- Base clocks ck_b
  - pck : A strictly periodic clock ((n, p) with n and p unknown)
  - ck_b **on** $C(c, w)$: Application of **on** $C(c, w)$ to ck_b
- Refinements relate to the period and offset of the clock

`i: rate (10,5)` $\Rightarrow \{\nu{:}\mathtt{pck} \mid \pi(\nu) = 10 \wedge \varphi(\nu) = 5\}$

`*^2` $\Rightarrow x{:}\{\nu{:}\mathtt{pck} \mid 2 \ \mathbf{div}\ \pi(\nu)\} \rightarrow$

$\{\nu{:}\mathtt{pck} \mid \pi(\nu) = \pi(x)/2 \wedge \varphi(\nu) = \varphi(x)\}$

Refinement clock calculus

- Clocks $\neq$ clock types
- Clock type : Reconstruction of the clock by the type system
- Base clocks ck_b
  - pck : A strictly periodic clock ((n, p) with n and p unknown)
  - ck_b **on** $C(c, w)$: Application of **on** $C(c, w)$ to ck_b
- Refinements relate to the period and offset of the clock

`i: rate (10,5)` $\Rightarrow \{\nu{:}\mathtt{pck} \mid \pi(\nu) = 10 \wedge \varphi(\nu) = 5\}$

`*^2` $\Rightarrow x{:}\{\nu{:}\mathtt{pck} \mid 2 \ \mathbf{div} \ \pi(\nu)\} \rightarrow$

$$\{\nu{:}\mathtt{pck} \mid \pi(\nu) = \pi(x)/2 \wedge \varphi(\nu) = \varphi(x)\}$$

For brevity : $\{\nu{:}ck \mid \pi(\nu) = r_n \wedge \varphi(\nu) = r_o\} = \{\nu{:}ck \mid \langle r_n, r_o\rangle\}$

Refinement clock calculus

- Clocks $\neq$ clock types
- Clock type : Reconstruction of the clock by the type system
- Base clocks ck_b
  - pck : A strictly periodic clock ((n, p) with n and p unknown)
  - ck_b **on** $C(c, w)$: Application of **on** $C(c, w)$ to ck_b
- Refinements relate to the period and offset of the clock

`i: rate (10,5)` $\Rightarrow \{\nu{:}\texttt{pck} \mid \langle 10, 5\rangle\}$

`*^2` $\Rightarrow x{:}\{\nu{:}\texttt{pck} \mid 2\ \mathbf{div}\ \pi(\nu)\} \rightarrow \{\nu{:}\texttt{pck} \mid \langle x/2, x\rangle\}$

For brevity : $\{\nu{:}ck \mid \pi(\nu) = r_n \wedge \varphi(\nu) = r_o\} = \{\nu{:}ck \mid \langle r_n, r_o\rangle\}$

Refinement clock calculus

View computation

$\texttt{i when true(c)} \Rightarrow \{\nu{:}\texttt{pck}\ \mathbf{on}\ \mathit{true}(c, \{\nu{:}\texttt{pck} \mid \langle 20, 5\rangle\}) \mid \langle 10, 5\rangle\}$

- Users don't annotate views
- Collect constraints on the view

⇒ Refinement typer (SMT solver) finds solution with lowest period

Automata semantics
Classification

```
i: rate(10,0) j: rate(20,0)
c: rate(15,0)
o,p = h(k,l);
automaton
| S1 ->
  unless c then S2;
  k = f1(i);
  l = g1(j);
| S2 ->
  unless c then S1;
  k = f2(i);
  l = g2(j);
end
```

Transitioning from mode S1 to S2 ...

Dataflow compliant

Breaks dataflow (not supported)

Automata semantics

Classification

```
i: rate(10,0) j: rate(20,0)
c: rate(15,0)
o,p = h(k,l);
automaton
| S1 ->
  unless c then S2;
  k = f1(i);
  l = g1(j);
| S2 ->
  unless c then S1;
  k = f2(i);
  l = g2(j);
end
```

Transitioning from mode S1 to S2 ...

Unchanged task (`h(k,l)`)

- Periodic : Unaffected
- Aperiodic : Execution suppressed

Dataflow compliant

Breaks dataflow (not supported)

Automata semantics
Classification

```
i: rate(10,0) j: rate(20,0)
c: rate(15,0)
o,p = h(k,l);
automaton
| S1 ->
  unless c then S2;
  k = f1(i);
  l = g1(j);
| S2 ->
  unless c then S1;
  k = f2(i);
  l = g2(j);
end
```

Transitioning from mode S1 to S2 ...

Old-mode task (`f1(i)`, `g1(j)`)

- Late retirement : Executes until specific point
- Early retirement : Abort immediately

Dataflow compliant

Breaks dataflow (not supported)

Automata semantics
Classification

```
i: rate(10,0) j: rate(20,0)
c: rate(15,0)
o,p = h(k,l);
automaton
| S1 ->
  unless c then S2;
  k = f1(i);
  l = g1(j);
| S2 ->
  unless c then S1;
  k = f2(i);
  l = g2(j);
end
```

Transitioning from mode S1 to S2 ...

New-mode task (`f2(i)`, `g2(j)`)

- Non-overlapping : Distinct before-after
- Overlapping : Potential co-overlapp between modes

Dataflow compliant

Breaks dataflow (not supported)

Automata semantics

Flexibility

```
i: rate(10,0) j: rate(20,0)
c: rate(15,0)
o,p = h(k,l);
automaton
| S1 ->
  unless c then S2;
  k = f1(i);
  l = g1(j);
| S2 ->
  unless c then S1;
  k = f2(i);
  l = g2(j);
end
```

- `k` observes the automaton state with view $(30, 0)$
- `l` observes the automaton state with view $(60, 0)$
- Implements an *overlapping* mode change protocol, i.e. during a transition, `f2(i)` and `g1(j)` might co-exist
- Switching to a *non-overlapping* requires to give all dataflows the same view

Automata semantics
Flexibility

```
i: rate(10,0) j: rate(20,0)
c: rate(15,0)
o,p = h(k,l);
c_slow = c/^2;
automaton
| S1 ->
  unless c_slow then S2;
  k = f1(i);
  l = g1(j);
| S2 ->
  unless c_slow then S1;
  k = f2(i);
  l = g2(j);
end
```

- `k` observes the automaton state with view $(30, 0)$
- `l` observes the automaton state with view $(60, 0)$
- Implements an *overlapping* mode change protocol, i.e. during a transition, `f2(i)` and `g1(j)` might co-exist
- Switching to a *non-overlapping* requires to give all dataflows the same view

Conclusion

- Extended a multi-periodic sync. language to support mode-change automata
- Flexible enough for different mode change protocols
- Clock calculus guarantees that programs remain sound

Future work

- Lift requirement to annotate node inputs
- View computation $\sim$ program synthesis ?
  - "Completely" remove rate-transition operators

Thank you for your attention

Questions ? Postdoc offers ?