Oversampling in a Dataflow Synchronous Language (Heptagon)

Léonard Gérard¹

¹PARKAS team ENS

Synchron'11

Oversampling in a Dataflow Synchronous Language (Heptagon)

(Heptagon)

Léonard Gérard¹

¹PARKAS team
ENS

Synchron'11

Oversampling in a Dataflow Synchronous Language (Heptagon)

Heptagon

A small Scade v6

- Automaton
- Arrays and iterators
- ► Modular reset
- ► Static parameters

Novelties

- ► Memory optimization for arrays
- ► Controller synthesis
- ► and WIP
 - asynchronous computations
 - oversampling
 - ▶ lucy-n generation
 - **...**

Soon to be released as open source...

Oversampling in a Dataflow Synchronous Language (Heptagon)

A small Scale of A small Scale

```
Classic oversampling example
   node f(x :int) returns (cpt, y :int)
   let
     y = x + 1
     cpt = (0 fby cpt) + 1
   tel
   node g(x :int; c :bool) returns (out :int)
   var t, cpt, y, last_y :int;
   let
     (cpt, y) = f(t);
     t = merge c x (last_y whenot c);
     last_y = 0 fby y;
     out = y whenot c;
   tel
   node main() returns (out :int; c :bool) var x :int;
   let
     x = 0 \text{ fby } (x+10);
     c = true fby false fby c;
     out = g(x,c);
                                                        3/19
```

```
var t, cpt, y, last_y :int;
-Classic oversampling example
                                                                                                                t = merre c x (last v whenot c):
                                                                                                                last v = 0 fbv v:
                                                                                                                out = y whenot c;
                                                                                                               node main() returns (out :int; c :bool) var x :int;
                                                                                                                c = true fby false fby c;
                                                                                                                out = g(x,c);
```

Oversampling in a Dataflow Synchronous Language (Heptagon)

Classic oversampling example

cpt = (0 fby cpt) + 1 tel node g(x :int: c :bool) returns (out :int)

node f(x :int) returns (cot. v :int)

Classic oversampling example

```
node g(x :int; c :bool) returns (out :int)
var t, cpt, y, last_y :int;
let
  (cpt, y) = f(t);
  t = merge c x (last_y whenot c);
  last_y = 0 fby y;
  out = y whenot c;
tel

val g:: (. on c, c : .) -> . on not c
```

С	true	false	true	false	true	
X	<i>x</i> ₀		<i>x</i> ₁		<i>x</i> ₂	
t	<i>x</i> ₀	$f(x_0)$	<i>x</i> ₁	$f(x_1)$	<i>x</i> ₂	
У	$f(x_0)$	$f^{2}(x_{0})$	$f(x_1)$	$f^{2}(x_{1})$	$f(x_2)$	
cpt	1	2	3	4	5	
out		$f^2(x_0)$		$f^{2}(x_{1})$		

Oversampling in a Dataflow Synchronous Language (Heptagon)

Classic oversampling example

Classic oversampling example

Classic oversampling example

Oversampling with clock given as argument.

Why hiding the oversampling clock?

- ▶ It is strange to define the clock outside of g.
- ▶ The node g communicate at each of its steps, even if no value for x and out is meaningful.
- ► From the outside, the clocks of x and out are needlessly complex.

We would like

val g:: . -> .

X	<i>x</i> ₀	<i>x</i> ₁	<i>X</i> ₂	
С	[true false]	[true false]	[true false]	
t	$\begin{bmatrix} x_0 & f(x_0) \end{bmatrix}$	$[x_1 f(x_1)]$	$\begin{bmatrix} x_2 & f(x_2) \end{bmatrix}$	
cpt	[1 2]	[3 4]	[5 6]	
У	$[f(x_0) f^2(x_0)]$	$[f(x_1) f^2(x_1)]$	$[f(x_2) f(f(x_2))]$	
out	$f^{2}(x_{0})$	$f^{2}(x_{1})$	$f^{2}(x_{2})$	

Oversampling in a Dataflow Synchronous Language (Heptagon)

Why hiding the oversampling clock?

Why hiding the oversampling clock?

The node g communicate at each of its steps,

even if no value for x and out is meaningful. From the outside, the clocks of x and out are needlessly

CI	omplex.						
We would like							
val g	g::> .						
×	X0	XI.	70				
c	true false	true false	true false				
-	$[x_0 \ f(x_0)]$	$[x_1 f(x_1)]$	[x ₂ f(x ₂)]				
cpt	[1 2]	[3 4]	[5 6]				
У	$[f(x_0) f^2(x_0)]$	$[f(x_1) \ f^2(x_1)]$	$[f(x_2) \ f(f(x_2))]$				
out	$f^{2}(x_{0})$	$f^{2}(x_{1})$	$f^{2}(x_{2})$				

Local Hiding of Oversampling in Heptagon

Any node which would be given the *usually illegal* signature

```
val n:: on c \rightarrow on c is transformed into a node with signature val n:: . \rightarrow .
```

with a simple transformation in the generated sequential code:

Oversampling in a Dataflow Synchronous Language (Heptagon)

Local Hiding of Oversampling in Heptagon

Any node which would be given the usually illegal signature val n:: . on c -> . on c is transformed into a node with signature val n:: . -> . . with a simple transformation in the generated sequential code:

Local Hiding of Oversampling in Heptagon

Local Hiding of Oversampling in Heptagon (bis)

```
val n:: (c : . on e on d, . on e on d on c)
         -> . on e on d on c
is transformed into a node with signature
val n:: (c:., on c) \rightarrow . on c
```

```
step_n(x) {
                             [vars n]
step_n(c,x) {
  [vars_n]
                             do {
  [code n]
                             [code n]
                            } while (!(d && e));
 return y;
                             return y;
```

PS: The common root of the clocks of the signature is the local oversampling. Here . on e on d.

Oversampling in a Dataflow Synchronous Language (Heptagon) is transformed into a node with signature Local Hiding of Oversampling in Heptagon (bis) [vars_n] [code_n] PS: The common root of the clocks of the signature is the local oversampling. Here , on e on d.

First attempt to use LHO, before LHO transformation

```
node g(x :int) returns (out :int)
var c :bool; t, cpt, y, last_y :int;
let
  c = true fby false fby c;
  (cpt, y) = f(t);
  t = merge c x (last_y whenot c);
  last_y = 0 fby y;
  out = y when c;
tel
val g:: . on c -> . on c
```

С	true	false	true	false	true	
X	<i>x</i> ₀		<i>x</i> ₁		<i>x</i> ₂	
t	<i>x</i> ₀	$f(x_0)$	<i>x</i> ₁	$f(x_1)$	<i>x</i> ₂	
У	$f(x_0)$	$f^{2}(x_{0})$	$f(x_1)$	$f^{2}(x_{1})$	$f(x_2)$	
cpt	1	2	3	4	5	
out	$f(x_0)$		$f(x_1)$		$f(x_2)$	

First attempt to use LHO, before LHO transformation Oversampling in a Dataflow Synchronous Language (Heptagon) t = merge c x (last v whenot c): last v = 0 fbv v: First attempt to use LHO, before LHO transformation

We are asked to give the same sampling to the input and the output. So naively we do so.

First attempt to use LHO, LHO done

```
node g(x :int) returns (out :int)
var c :bool; t, cpt, y, last_y :int;
let
   c = true fby false fby c;
   (cpt, y) = f(t);
   t = merge c x (last_y whenot c);
   last_y = 0 fby y;
   out = y when c;
tel
val g:: . -> .
```

С	[true] [false	true] [false	true] [
X	[x ₀] [x ₁] [x_2] [
t	$\begin{bmatrix} x_0 \end{bmatrix} \begin{bmatrix} f(x_0) \end{bmatrix}$	x_1] [$f(x_1)$	<i>x</i> ₂] [
У	$[f(x_0)][f^2(x_0)]$	$f(x_1)$] [$f^2(x_1)$	$f(x_2)$] [
cpt	[1] [2	3] [4	5] [
out	[f(x ₀)] [$f(x_1)$]	$f(x_2)$] [

Oversampling in a Dataflow Synchronous Language (Heptagon)

First attempt to use LHO, LHO done

First attempt to use LHO, LHO done

some g(x in land) returne (use init)

see c label; t, cgt, r, last, r init)

see c label; t, cgt, r, last, r init)

see c label; t, cgt, r, last, r init)

see c label;

see c l

- The square brackets are used to display the oversampling: from the outside of the node, the signature hide the inner steps of these brackets.
- Nothing new, to be able to do oversampling, we need to loose one instant. See the Lucid V3 manual page 24.

Correct use of LHO

```
node g(x :int) returns (out :int)
var c :bool; t, cpt, y, last_y :int;
let
  c = true fby false fby c;
  (cpt, y) = f(t);
 t = merge c x (last_y whenot c);
 last_y = 0 fby y;
  out = last_y when c;
tel
val g:: . -> .
```

С	[true] [false	true] [false	true] [
X	[x ₀] [x_1] [<i>x</i> ₂] [
t	$[x_0] [f(x_0)$	x_1] [$f(x_1)$	x ₂] [
У	$[f(x_0)][f^2(x_0)]$	$f(x_1)] [f^2(x_1)]$	$f(x_2)$] [
cpt	[1] [2	3] [4	5][
out	[0] [$f^{2}(x_{0})$] [$f^2(x_1)$] [

Oversampling in a Dataflow Synchronous Language (Heptagon)

-Correct use of LHO

Correct use of LHO

node g(x :int) returns (out :int) c = true fby false fby c; (cpt, y) = f(t); t = merge c x (last v whenot c): last v = 0 fbv v: out = last_y when c;

8						
C	true	false	true	false	true	100
х	(XO		xı		30	T-
7	10	[f(x ₀)	xı	f(x1)	30	100
y.	f(x ₀)	$f^{2}(x_{0})$	$f(x_1)$	$f^{2}(x_{1})$	$f(x_2)$	To.
p!	[1]	[2	3	4	5]	1
Ş,	[0]		f2(x0)		f2(x1)]	100

Correct use of LHO (bis)

```
node g(x :int) returns (out :int)
var c :bool; t, cpt, y, last_y :int;
let
   c = true fby false fby false fby c;
   (cpt, y) = f(t);
   t = merge c x (last_y whenot c);
   last_y = 0 fby y;
   out = last_y when c;
tel
val g:: . -> .
```

С	[true]	[false	false	true] [false	false	true]
X	[x ₀]	[x ₁] [x ₂]
t	[x ₀]	$[f(x_0)]$	$f^{2}(x_{0})$	x_1] [$f(x_1)$	$f^{2}(x_{1})$	$f^3(x_1)\dots$
У	$[f(x_0)]$	$[f^2(x_0)]$	$f^{3}(x_{0})$	$f(x_1)$] [$f^2(x_1)$	$f^{3}(x_{1})$	$f(x_2)$]
cpt	[1]	[2	3	4] [5	6	7]
out	[0]	[$f^3(x_0)$] [$f^3(x_1)$]

Oversampling in a Dataflow Synchronous Language (Heptagon)

Correct use of LHO (bis)

Correct use of LHO (bis)

It is now easy to do any number of oversampling steps.

But can't we do it without delay!?

```
node g(x :int) returns (out :int)
var t, cpt, y, last_y :int; c :bool;
let
   c = true fby false fby c;
   (cpt, y) = f(t);
   t = merge c x (last_y whenot c);
   last_y = 0 fby y;
   out = y whenot c;
tel
```

 $val g:: . on c \rightarrow . on not c$

С	[true	false]	[true	false]	[true	false]
X	[x ₀]	[x ₁]	[x ₂]
t	[x ₀	$f(x_0)$	[x ₁	$f(x_1)$	[x ₂	$f(x_2)$]
У	$[f(x_0)]$	$f^{2}(x_{0})$	$[f(x_1)]$	$f^{2}(x_{1})$	$[f(x_2)]$	$f(f(x_2))$]
cpt	[1	2]	L	4]	[5	6]
out	[$f^2(x_0)$]	[$f^{2}(x_{1})$	[$f(f(x_2))$]

Oversampling in a Dataflow Synchronous Language (Heptagon)

But can't we do it without delay!?

pode g(x :int) returns (out :int)
rar t, cpt, y, last_y :int; c :bool;
c = true fby false fby c;
(cpt, y) = f(t);
t = nerge c x (last_y whenot c);
last_y = 0 fby y;
out = y whenot c;
tel

But can't we do it without delay!?

ı	true false	[true false]	true false].	
i	(X)	[20]	X2 .	
ı	[x ₀ f(x ₀)]	[x ₁ f(x ₁)]	$\begin{bmatrix} x_2 & f(x_2) \end{bmatrix}$.	×
	$f(x_0) f^2(x_0)$	$f(x_1) f^2(x_1)$	$f(x_2)$ $f(f(x_2))$.	
	1 2	3 4	5 6.	
1	f2(x0)	f2(x1)	$f(f(x_2))$.	

Even if this seems to generate correct code with the LHO transformation, the compiler rejects this program... It is not able to recognize the interleaving of the clock.

No, we cannot generalize LHO

```
node g(x :int) returns (out :int)
var t, cpt, y, last_y :int; c :bool;
let
   c = true fby false fby false fby c;
   (cpt, y) = f(t);
   t = merge c x (last_y whenot c);
   last_y = 0 fby y;
   out = y whenot c;
tel
```

There are two outputs for one input...

С	[true	false	false]	[true	false	false] [
X	[x ₀]	[x ₁] [
t	[x ₀		$f^{2}(x_{0})$			$f^{2}(x_{1})$] [
У	$[f(x_0)]$	$f^2(x_0)$	$f^3(x_0)$]	$f(x_1)$	$f^{2}(x_{1})$	$f^3(x_1)$] [
cpt	[1	2	3]	[4	5	6][
out	[$f^{2}(x_{0})$	$f^{3}(x_{0})$	[$f^{2}(x_{1})$	$f^3(x_1)$] [

Oversampling in a Dataflow Synchronous Language (Heptagon)

No, we cannot generalize LHO

var t, cpt, y, last_y :int; c :bool;
let c = true fby false fby false fby c;
(cpt, y) = f(t);
t = merge c x (last_y whenet c);
last_y = 0 fby y;
out = y whenet c;
tel
There are two outputs for one input...

c true false false true

No. we cannot generalize LHO

The compiler rejects this program rightfully.

Enumerated clocks are equivalent, but insightful

out

```
type t = In | C | Out
node g(x :int) returns (out :int)
var t, cpt, y, last_y :int; c :t;
let
  c = In fby C fby C fby Out fby c;
  (cpt, y) = f(t);
  t = merge c (In \rightarrow x) (C \rightarrow last_y when C(c))
                              (Out -> last_y when Out(c));
  last_y = 0 fby y;
  out = y when Out(c);
tel
val g:: on In(c) \rightarrow . on Out(c)
                                 Out
                                         ln
        x_0
                                         X_1
                                                        . . .
                       \overline{f}^3(x_0)
       \int f(x_0)
               f^{2}(x_{0})
                               f^4(x_0)
                                                        . . .
                                                  6
   cpt
                                                        . . .
```

 $f^4(x_0)$

Enumerated clocks are equivalent, but insightful

| Enumerated clocks are equivalent, but insightful
| Enumerated clocks are equivalent, but insightful
| Enumerated clocks are equivalent, but insightful
| Enumerated clocks are equivalent, but insightful
| Enumerated clocks are equivalent, but insightful
| Enumerated clocks are equivalent, but insightful
| Enumerated clocks are eq

Enumerated clocks are equivalent, but insightful

node g(x :int) returns (out :int)

The compiler rejects this program...

Oversampling in a Dataflow Synchronous Language (Heptagon)

. . .

No, we still cannot generalize LHO

cpt

out

```
type t = In | C | Out
node g(x :int) returns (out :int)
var t, cpt, y, last_y :int; c :t;
let
  c = In fby C fby Out fby Out fby c;
  (cpt, y) = f(t);
  t = merge c (In \rightarrow x) (C \rightarrow last_y when C(c))
                            (Out -> last_y when Out(c));
  last_y = 0 fby y;
  out = y when Out(c);
tel
val g:: on In(c) \rightarrow . on Out(c)
                       Out
                               Out 1
                                     [ In
                                                     . . .
       f(x_0)
               f^{2}(x_{0})
                      f^{3}(x_{0})
                                                     . . .
```

 $f^{3}(x_{0})$

 $f^4(x_0)$

Oversampling in a Dataflow Synchronous Language (Heptagon)

No, we still cannot generalize LHO

No, we still cannot generalize LHO

No, we still cannot generalize LHO

No. we still cannot generalize LHO

The compiler reject this program rightfully.

. . .

. . .

6

No, we still cannot generalize LHO (ter)

 $[f(x_0)]$

cpt

out

```
type t = In | C | Out
node g(x :int) returns (out :int)
var t, cpt, y, last_y :int; c :t;
let
  c = In fby C fby C fby C;
  (cpt, y) = f(t);
  t = merge c (In \rightarrow x) (C \rightarrow last_y when C(c))
                            (Out -> last_y when Out(c));
  last_y = 0 fby y;
  out = y when Out(c);
tel
val g:: on In(c) \rightarrow . on Out(c)
                                      Ιn
        x_0
                                      X_1
                                                   . . .
              f^{2}(x_{0})
                     f^{3}(x_{0})
```

```
Oversampling in a Dataflow Synchronous Language (Heptagon)
                                                                                                   node g(x :int) returns (out :int)
                                                                                                   var t, cpt, y, last_y :int; c :t;
                                                                                                    t = merge c (In -> x) (C -> last_y when C(c))
           └─No, we still cannot generalize LHO (ter)
                                                                                                    out = v when Out(c):
```

No. we still cannot generalize LHO (ter)

The compiler reject this program rightfully.

. . .

. . .

. . .

6

Bursts must be well formed

Observed instant

An observed instant of a burst is an instant accessed, from someone which is observing the burst as one instant.

—In and Out are observed. C isn't.

Sufficient and necessary condition to apply LHO

During one burst, every observed instant must appear one and only one time.

Burst boundaries

- ▶ The boundaries of bursts are constrained by the causality.
- ▶ In the case of causal functions with outputs depending on all inputs, the end of the burst is aligned with the last output.

Oversampling in a Dataflow Synchronous Language (Heptagon)

Bursts must be well formed

erved instant

Observed instant
An observed instant of a burst

An observed instant of a burst is an instant accessed, from someone which is observing the burst as one instant.

—In and Out are observed, C isn't.

Sufficient and necessary condition to apply LHO
During one burst, every observed instant must appear

Burst boundaries

The boundaries of bursts are constrained by the causality.

In the case of causal functions with outputs depending on a inputs, the end of the burst is alimed with the last output.

Back to Heptagon:

Note that on $c \equiv \text{on true}(c)$ and on not $c \equiv \text{on false}(c)$

- ▶ . on C(b) -> . on C(b): accepted by LHO
 - ▶ Reactivity requires C(b) to be true an infinite amount of time.
 - ► Should/how can we ensure it?
 - ▶ Right now our prototype in Heptagon doesn't check it.
- . on C(b) -> . on C2(b): rejected by LHO
 - ► Only the perfect interleaving of the two constructors is possible.

Oversampling in a Dataflow Synchronous Language (Heptagon)

011-12-13

-Back to Heptagon:

Back to Heptagon:

Note that on $c \equiv on \ true(c)$ and on not $c \equiv on \ false(c)$

- Reactivity requires C(b) to be true an infinite amount of time.
 Should/how can we ensure it?
- Should/how can we ensure it?
 Right now our prototype in Heptagon doesn't check it.
 on C(b) -> . on C2(b): rejected by LHO
- Only the perfect interleaving of the two constructors is possible.

Proposal and questions

Iterator primitive:

```
► Static: b = iter [In; C; C; Out]
```

- ▶ Dynamic: b = iter list
- ► How much do we need dynamic iteration?
- ► What should be dynamic (in increasing difficulty)
 - ► the size?
 - ▶ the order?
 - ► the type?

U

se a restricted

Oversampling in a Dataflow Synchronous Language (Heptagon)

Proposal and questions

Proposal and questions

Proposal and questions

Proposal and questions

se a restricted