

MECHANIZED SEMANTICS AND VERIFIED COMPILATION FOR A DATAFLOW SYNCHRONOUS LANGUAGE WITH RESET

Timothy Bourke^{1,2} Lélío Brun^{1,2} Marc Pouzet^{3,2,1}

POPL'20 — January 24, 2020

¹Inria Paris

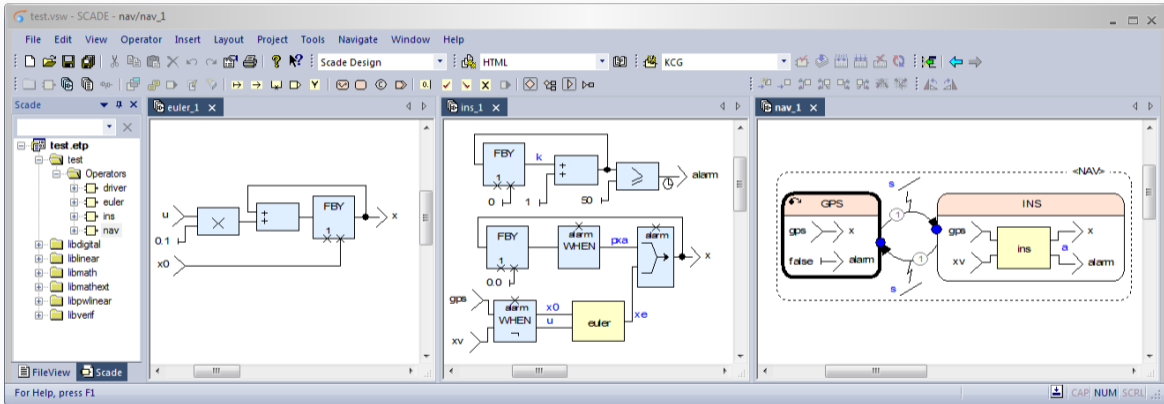
²École normale supérieure – PSL University

³Sorbonne University

`velus.inria.fr`
`github.com/INRIA/velus`

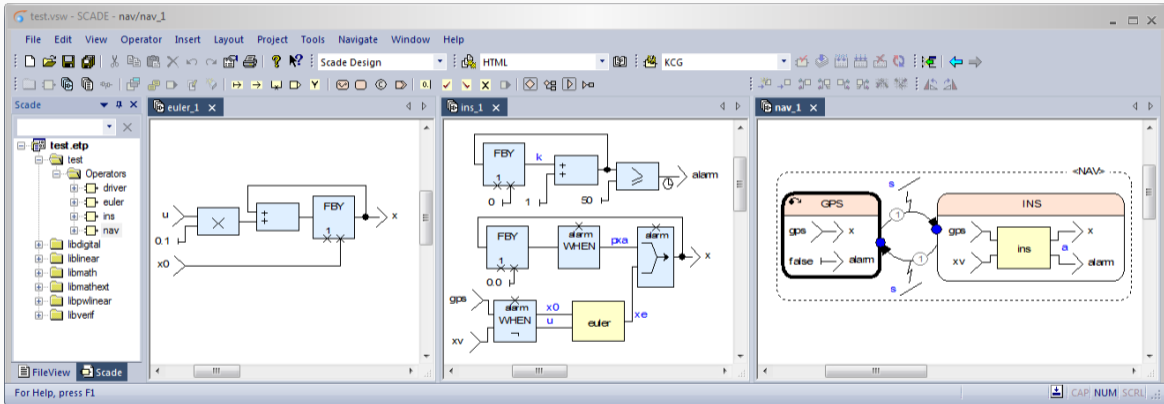
MOTIVATION: MODEL BASED DESIGN IN SCADE SUITE

www.ansys.com/products/embedded-software/ansys-scade-suite



MOTIVATION: MODEL BASED DESIGN IN SCADE SUITE

www.ansys.com/products/embedded-software/ansys-scade-suite

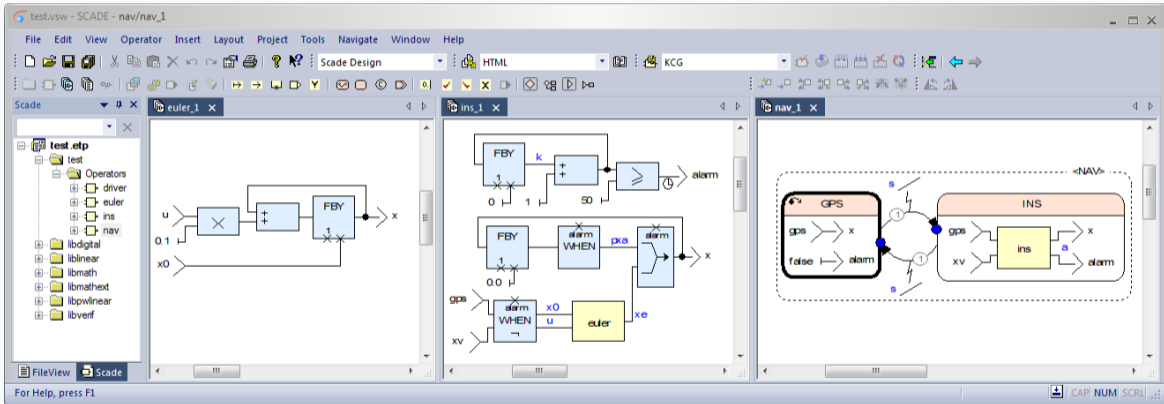


block / node = system

line = signal

MOTIVATION: MODEL BASED DESIGN IN SCADE SUITE

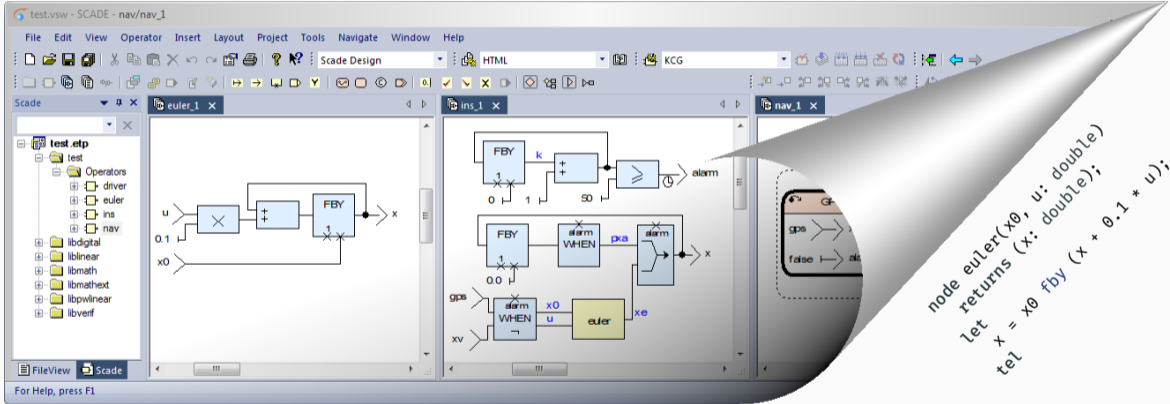
www.ansys.com/products/embedded-software/ansys-scade-suite



block / node = system = stream function
line = signal = stream of values

MOTIVATION: MODEL BASED DESIGN IN SCADE SUITE

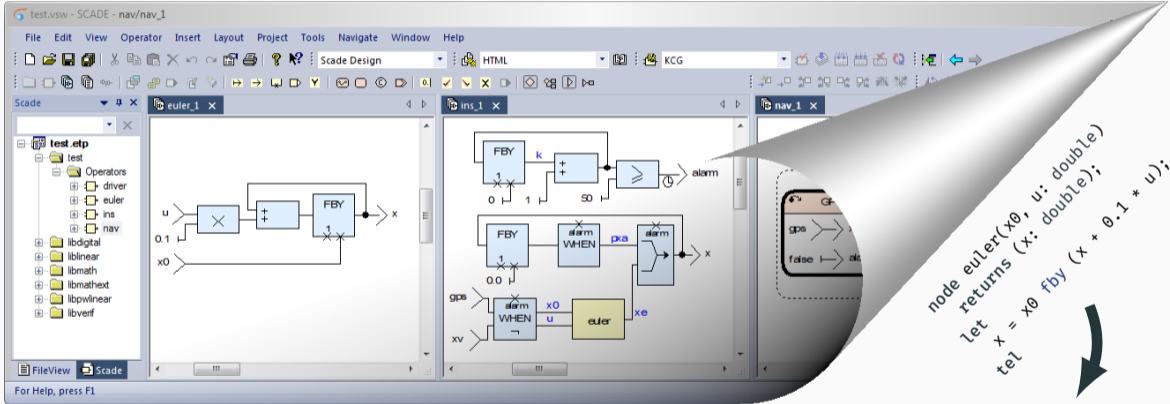
www.ansys.com/products/embedded-software/ansys-scade-suite



block / node = system = stream function
line = signal = stream of values

MOTIVATION: MODEL BASED DESIGN IN SCADE SUITE

www.ansys.com/products/embedded-software/ansys-scade-suite



The screenshot shows the SCADE Suite software interface. The main window displays a block diagram with several components: a multiplier block, an adder block, and a feedback block labeled 'FBY'. The diagram is connected to a navigation pane on the left, which shows a tree structure of the project files, including 'test.etp', 'test', 'Operators', 'driver', 'euler', 'ins', 'nav', 'libdigital', 'liblinear', 'libmath', 'libmathext', 'libpwnlinear', and 'libverif'. The interface also includes a menu bar (File, Edit, View, Operator, Insert, Layout, Project, Tools, Navigate, Window, Help) and a toolbar with various icons for file operations and simulation.

block / node = system = stream function
line = signal = stream of values

```
node euler(x0, u: double)  
returns (x: double);  
let  
  x = x0 fby (x + 0.1 * u);  
tel
```

sequential program
(C, Ada, assembly)

Model-Based Design Languages

SCADE, Lustre, Simulink



Interactive Theorem Provers

Coq

Challenges

1. Mechanize the semantics
2. Prove the compilation algorithms correct

Model-Based Design Languages

SCADE, Lustre, Simulink



Interactive Theorem Provers

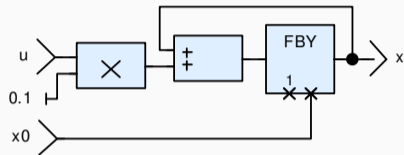
Coq

Challenges

1. Mechanize the semantics
2. Prove the compilation algorithms correct

Focus: modular reset

EXAMPLE



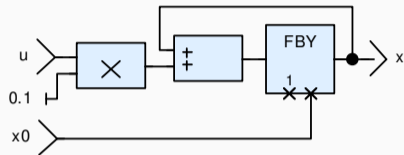
```

node euler(x0, u: double)
  returns (x: double);
let
  x = x0 fby (x + 0.1 * u);
tel

```

x_0	0.00	1.55	3.62	5.46	...
u	15.00	20.00	17.00	12.00	...
$x + 0.1 \times u$	1.50	3.50	5.20	6.70	...
x	0.00	1.50	3.50	5.20	...

EXAMPLE



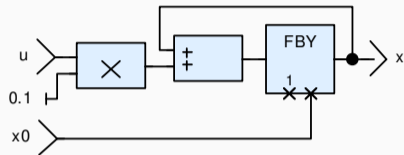
```

node euler(x0, u: double)
  returns (x: double);
let
  x = x0 fby (x + 0.1 * u);
tel

```

x_0	0.00	1.55	3.62	5.46	...
u	15.00	20.00	17.00	12.00	...
$x + 0.1 \times u$	1.50	3.50	5.20	6.70	...
x	0.00	1.50	3.50	5.20	...

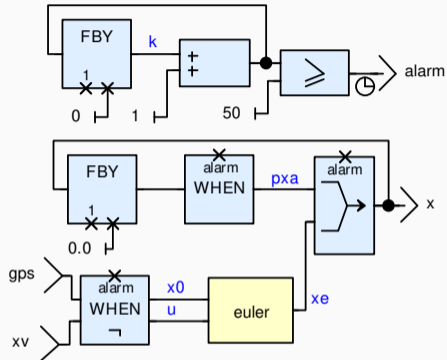
EXAMPLE



```
node euler(x0, u: double)
  returns (x: double);
let
  x = x0 fby (x + 0.1 * u);
tel
```

x_0	0.00	1.55	3.62	5.46	...
u	15.00	20.00	17.00	12.00	...
$x + 0.1 \times u$	1.50	3.50	5.20	6.70	...
x	0.00	1.50	3.50	5.20	...

EXAMPLE



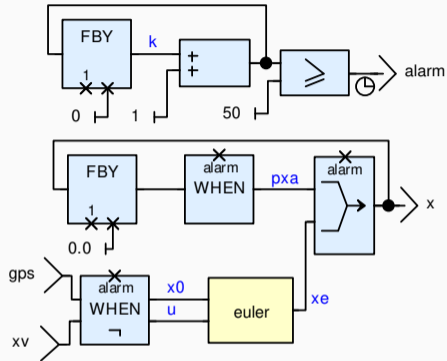
```

node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
  let
    k = 0 fby (k + 1);
    alarm = (k ≥ 50);
    xe = euler((gps, xv) when not alarm);
    pxa = (0. fby x) when alarm;
    x = merge alarm pxa xe;
  tel

```

<i>gps</i>	0.00	1.55	3.62	5.46	...	86.52	88.40	90.91	...
<i>xv</i>	15.00	20.00	17.00	12.00	...	18.00	23.00	20.00	...
<i>k</i>	0	1	2	3	...	49	50	51	...
<i>alarm</i>	F	F	F	F	...	F	T	T	...
<i>xe</i>	0.00	1.50	3.50	5.20	...	77.35			...
<i>pxa</i>					...		77.35	77.35	...
<i>x</i>	0.00	1.50	3.50	5.20	...	77.35	77.35	77.35	...

EXAMPLE



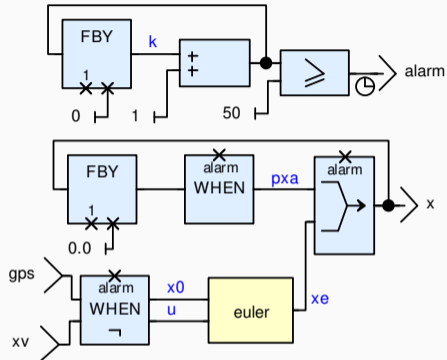
```

node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
let
  k = 0 fby (k + 1);
  alarm = (k ≥ 50);
  xe = euler((gps, xv) when not alarm);
  pxa = (0. fby x) when alarm;
  x = merge alarm pxa xe;
tel

```

gps	0.00	1.55	3.62	5.46	...	86.52	88.40	90.91	...
xv	15.00	20.00	17.00	12.00	...	18.00	23.00	20.00	...
k	0	1	2	3	...	49	50	51	...
alarm	F	F	F	F	...	F	T	T	...
xe	0.00	1.50	3.50	5.20	...	77.35			...
pxa					...		77.35	77.35	...
x	0.00	1.50	3.50	5.20	...	77.35	77.35	77.35	...

EXAMPLE



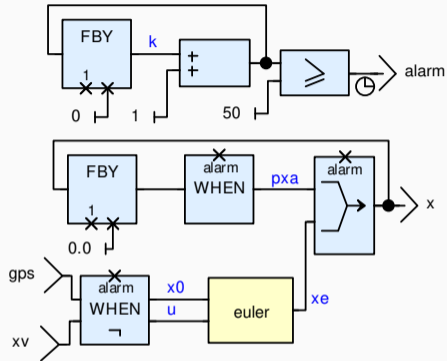
```

node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
  let
    k = 0 fby (k + 1);
    alarm = (k ≥ 50);
    xe = euler((gps, xv) when not alarm);
    pxa = (0. fby x) when alarm;
    x = merge alarm pxa xe;
  tel

```

<i>gps</i>	0.00	1.55	3.62	5.46	...	86.52	88.40	90.91	...
<i>xv</i>	15.00	20.00	17.00	12.00	...	18.00	23.00	20.00	...
<i>k</i>	0	1	2	3	...	49	50	51	...
<i>alarm</i>	F	F	F	F	...	F	T	T	...
<i>xe</i>	0.00	1.50	3.50	5.20	...	77.35			...
<i>pxa</i>					...		77.35	77.35	...
<i>x</i>	0.00	1.50	3.50	5.20	...	77.35	77.35	77.35	...

EXAMPLE



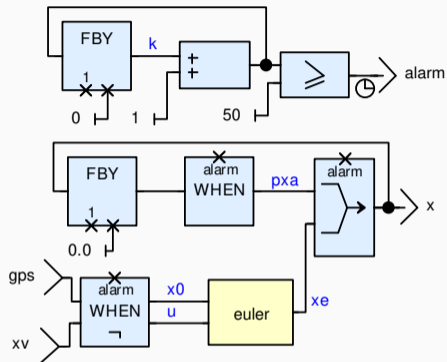
```

node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
  let
    k = 0 fby (k + 1);
    alarm = (k ≥ 50);
    xe = euler((gps, xv) when not alarm);
    pxa = (0. fby x) when alarm;
    x = merge alarm pxa xe;
  tel

```

<i>gps</i>	0.00	1.55	3.62	5.46	...	86.52	88.40	90.91	...
<i>xv</i>	15.00	20.00	17.00	12.00	...	18.00	23.00	20.00	...
<i>k</i>	0	1	2	3	...	49	50	51	...
<i>alarm</i>	F	F	F	F	...	F	T	T	...
<i>xe</i>	0.00	1.50	3.50	5.20	...	77.35			...
<i>pxa</i>					...		77.35	77.35	...
<i>x</i>	0.00	1.50	3.50	5.20	...	77.35	77.35	77.35	...

EXAMPLE



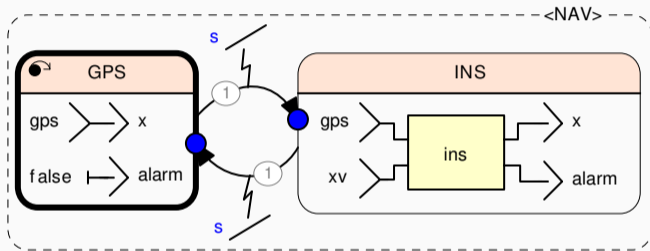
```

node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
  let
    x = merge alarm pxa xe;
    k = 0 fby (k + 1);
    pxa = (0. fby x) when alarm;
    xe = euler((gps, xv) when not alarm);
    alarm = (k ≥ 50);
  tel

```

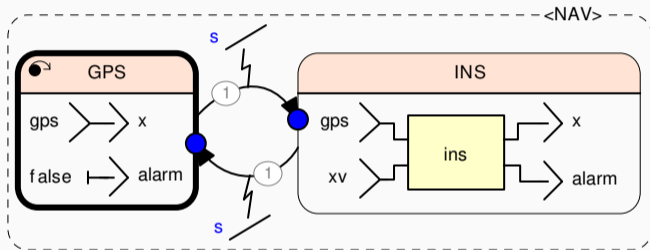
<i>gps</i>	0.00	1.55	3.62	5.46	...	86.52	88.40	90.91	...
<i>xv</i>	15.00	20.00	17.00	12.00	...	18.00	23.00	20.00	...
<i>k</i>	0	1	2	3	...	49	50	51	...
<i>alarm</i>	F	F	F	F	...	F	T	T	...
<i>xe</i>	0.00	1.50	3.50	5.20	...	77.35			...
<i>pxa</i>					...		77.35	77.35	...
<i>x</i>	0.00	1.50	3.50	5.20	...	77.35	77.35	77.35	...

EXAMPLE



Can be compiled into simple constructs

EXAMPLE



Can be compiled into simple constructs

We need a way to **reset the state of a node**

WITHOUT MODULAR RESET

```
node euler(x0, u: double, r: bool)
  returns (x: double);
let
  x = if r then x0 else x0 fby (x + 0.1 * u);
tel
```

```
node ins(gps, xv: double, r: bool)
  returns (x: double, alarm: bool)
  var k: int;
let
  x = merge alarm
      ((0. fby x) when alarm)
      (euler((gps, xv, r) whennot alarm));
  alarm = (k ≥ 50);
  k = if r then 0 else 0 fby (k + 1);
tel
...
(x, a) = ins(gps, xv, r);
```

WITHOUT MODULAR RESET

```

node euler(x0, u: double, r: bool)
  returns (x: double);
let
  x = if r then x0 else x0 fby (x + 0.1 * u);
tel

node ins(gps, xv: double, r: bool)
  returns (x: double, alarm: bool)
  var k: int;
let
  x = merge alarm
    ((0. fby x) when alarm)
    (euler((gps, xv, r) whenot alarm));
  alarm = (k ≥ 50);
  k = if r then 0 else 0 fby (k + 1);
tel
...
(x, a) = ins(gps, xv, r);

```

WITH MODULAR RESET

```

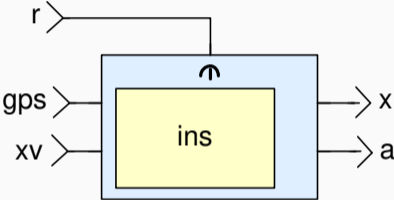
node euler(x0, u: double)
  returns (x: double);
let
  x = x0 fby (x + 0.1 * u);
tel

node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
let
  k = 0 fby (k + 1);
  alarm = (k ≥ 50);
  xe = euler((gps, xv) when not alarm);
  pxa = (0. fby x) when alarm;
  x = merge alarm pxa xe;
tel
...
(x, a) = (restart ins every r) (gps, xv);

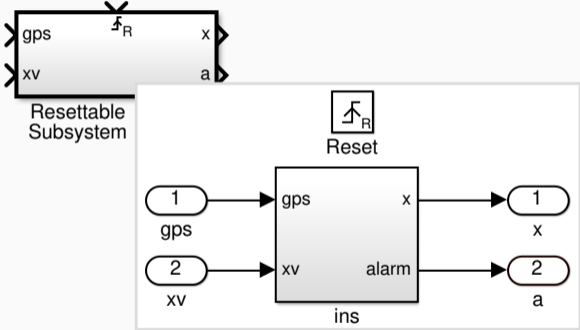
```

GRAPHICAL MODULAR RESET CONSTRUCT

SCADE

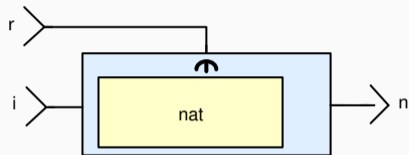
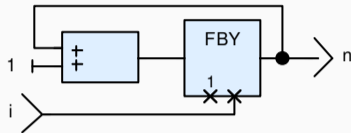


Simulink



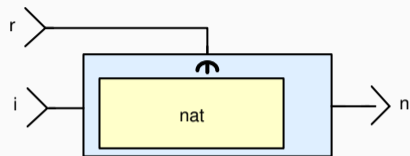
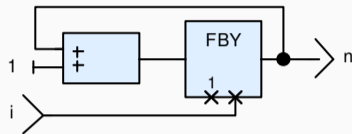
A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

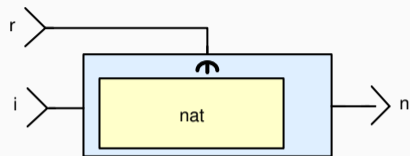
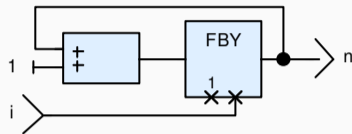
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



r	F
i	0
<hr/>	
$nat(i)$	0
$(\text{restart } nat \text{ every } r)(i)$	0

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

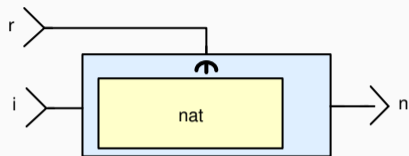
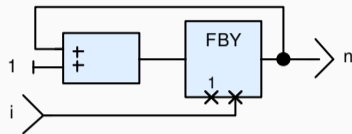
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



r	F	F
i	0	5
<hr/>		
$nat(i)$	0	1
$(\text{restart } nat \text{ every } r)(i)$	0	1

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

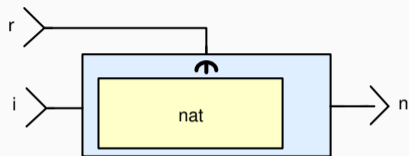
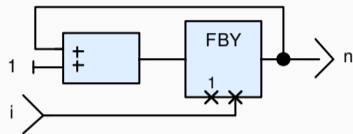
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



<i>r</i>	F	F	T
<i>i</i>	0	5	10
<hr/>			
<i>nat</i> (<i>i</i>)	0	1	2
(restart <i>nat</i> every <i>r</i>)(<i>i</i>)	0	1	10

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

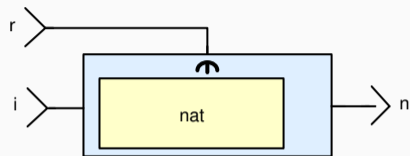
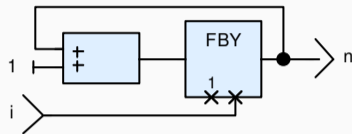
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



<i>r</i>	F	F	T	F
<i>i</i>	0	5	10	15
<hr/>				
<i>nat</i> (<i>i</i>)	0	1	2	3
(restart <i>nat</i> every <i>r</i>)(<i>i</i>)	0	1	10	11

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

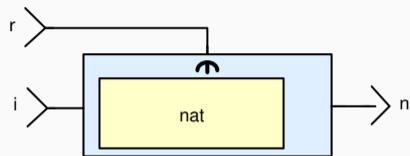
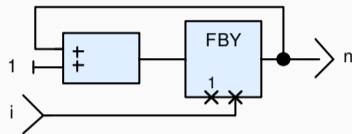
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



<i>r</i>	F	F	T	F	F
<i>i</i>	0	5	10	15	20
<hr/>					
<i>nat(i)</i>	0	1	2	3	4
<i>(restart nat every r)(i)</i>	0	1	10	11	12

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

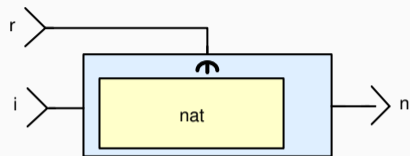
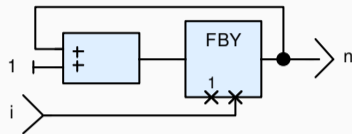
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



<i>r</i>	F	F	T	F	F	T
<i>i</i>	0	5	10	15	20	25
<hr/>						
<i>nat</i> (<i>i</i>)	0	1	2	3	4	5
(<i>restart nat every r</i>)(<i>i</i>)	0	1	10	11	12	25

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

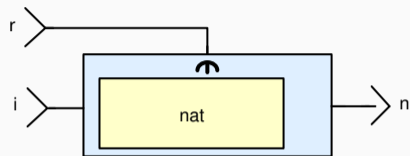
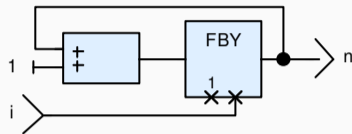
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



<i>r</i>	F	F	T	F	F	T	F
<i>i</i>	0	5	10	15	20	25	30
<i>nat(i)</i>	0	1	2	3	4	5	6
<i>(restart nat every r)(i)</i>	0	1	10	11	12	25	26

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```



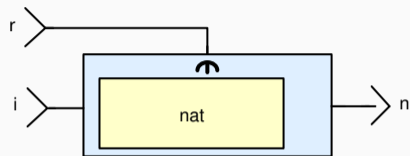
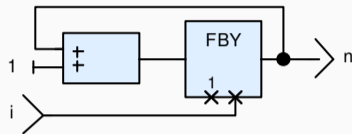
<i>r</i>	F	F	T	F	F	T	F	...
<i>i</i>	0	5	10	15	20	25	30	...
<hr/>								
<i>nat</i> (<i>i</i>)	0	1	2	3	4	5	6	...
(restart <i>nat</i> every <i>r</i>)(<i>i</i>)	0	1	10	11	12	25	26	...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

```

node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel

```



r	F	F	T	F	F	T	F	...
i	0	5	10	15	20	25	30	...
$nat(i)$	0	1	2	3	4	5	6	...
$(\text{restart } nat \text{ every } r)(i)$	0	1	10	11	12	25	26	...

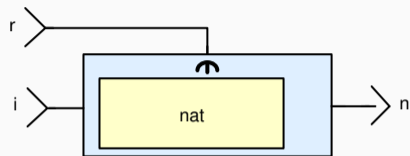
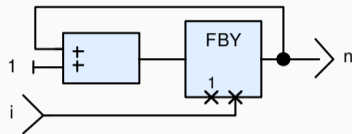
Could be implemented in a higher-order recursive language

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

```

node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel

```



<i>r</i>	F	F	T	F	F	T	F	...
<i>i</i>	0	5	10	15	20	25	30	...
<i>nat</i> (<i>i</i>)	0	1	2	3	4	5	6	...
(restart nat every r)(<i>i</i>)	0	1	10	11	12	25	26	...

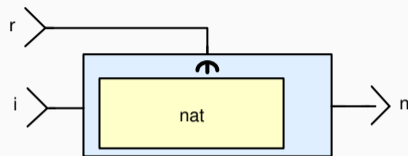
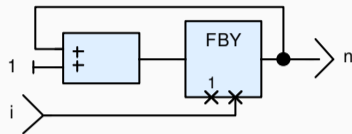
Could be implemented in a higher-order recursive language

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

```

node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel

```



<i>r</i>	F	F	T	F	F	T	F	...
<i>i</i>	0	5	10	15	20	25	30	...
<i>nat</i> (<i>i</i>)	0	1	2	3	4	5	6	...
(restart nat every r)(<i>i</i>)	0	1	10	11	12	25	26	...

Could be implemented in a higher-order recursive language

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r			F	F	T	F	F	T	F	...
<hr/>										
i			0	5	10	15	20	25	30	...

`(restart nat every r)(i)` 0 1 10 11 12 25 26 ...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
<hr/>								
count r	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...

(restart nat every r)(i) 0 1 10 11 12 25 26 ...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...

$(\text{restart nat every } r)(i)$ 0 1 10 11 12 25 26 ...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...
$\text{nat}(\text{mask}_r^0 i)$	0	1						...

$(\text{restart } \text{nat } \text{every } r)(i)$ 0 1 10 11 12 25 26 ...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...
$\text{nat}(\text{mask}_r^0 i)$	0	1						...
$\text{mask}_r^1 i$			10	15	20			...

$(\text{restart } \text{nat } \text{every } r)(i)$ 0 1 10 11 12 25 26 ...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...
$\text{nat}(\text{mask}_r^0 i)$	0	1						...
$\text{mask}_r^1 i$			10	15	20			...
$\text{nat}(\text{mask}_r^1 i)$			10	11	12			...
$(\text{restart } \text{nat } \text{every } r)(i)$	0	1	10	11	12	25	26	...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...
$\text{nat}(\text{mask}_r^0 i)$	0	1						...
$\text{mask}_r^1 i$			10	15	20			...
$\text{nat}(\text{mask}_r^1 i)$			10	11	12			...
$\text{mask}_r^2 i$						25	30	...
$(\text{restart nat every } r)(i)$	0	1	10	11	12	25	26	...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...
$\text{nat}(\text{mask}_r^0 i)$	0	1						...
$\text{mask}_r^1 i$			10	15	20			...
$\text{nat}(\text{mask}_r^1 i)$			10	11	12			...
$\text{mask}_r^2 i$						25	30	...
$\text{nat}(\text{mask}_r^2 i)$						25	26	...
$(\text{restart nat every } r)(i)$	0	1	10	11	12	25	26	...

A SIMPLER EXAMPLE: INTUITIVE SEMANTICS

r	F	F	T	F	F	T	F	...
$\text{count } r$	0	0	1	1	1	2	2	...
i	0	5	10	15	20	25	30	...
$\text{mask}_r^0 i$	0	5						...
$\text{nat}(\text{mask}_r^0 i)$	0	1						...
$\text{mask}_r^1 i$			10	15	20			...
$\text{nat}(\text{mask}_r^1 i)$			10	11	12			...
$\text{mask}_r^2 i$						25	30	...
$\text{nat}(\text{mask}_r^2 i)$						25	26	...
\vdots								
$(\text{restart nat every } r)(i)$	0	1	10	11	12	25	26	...

Node instantiation

$$H \vdash_{\text{eqn}} \mathbf{x} = f(\mathbf{e})$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es}{H \vdash_{\text{eqn}} x = f(e)}$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \vdash_{\text{node}} f(es) \Downarrow xs}{H \vdash_{\text{eqn}} x = f(e)}$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \vdash_{\text{node}} f(es) \Downarrow xs \quad H(x) = xs}{H \vdash_{\text{eqn}} x = f(e)}$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \vdash_{\text{node}} f(es) \Downarrow xs \quad H(x) = xs}{H \vdash_{\text{eqn}} x = f(e)}$$

Modular reset

$$H \vdash_{\text{eqn}} x = (\text{restart } f \text{ every } y)(e)$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \vdash_{\text{node}} f(es) \Downarrow xs \quad H(x) = xs}{H \vdash_{\text{eqn}} x = f(e)}$$

Modular reset

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad H(x) = xs}{H \vdash_{\text{eqn}} x = (\text{restart } f \text{ every } y)(e)}$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \vdash_{\text{node}} f(es) \Downarrow xs \quad H(x) = xs}{H \vdash_{\text{eqn}} x = f(e)}$$

Modular reset

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \begin{array}{l} H(y) = rs \quad r = \text{bools-of } rs \\ \forall k, \vdash_{\text{node}} f(\text{mask}_r^k es) \Downarrow \text{mask}_r^k xs \quad H(x) = xs \end{array}}{H \vdash_{\text{eqn}} x = (\text{restart } f \text{ every } y)(e)}$$

Node instantiation

$$\frac{H \vdash_{\text{exp}} e \Downarrow es \quad \vdash_{\text{node}} f(es) \Downarrow xs \quad H(x) = xs}{H \vdash_{\text{eqn}} x = f(e)}$$

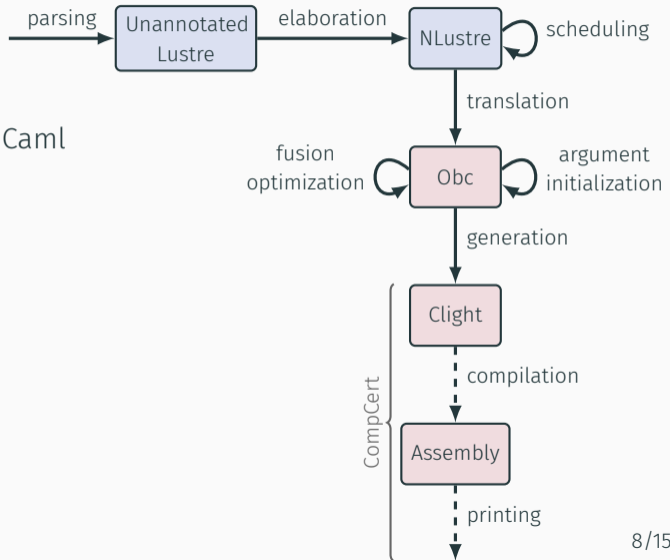
Modular reset

$$\frac{H(y) = rs \quad r = \text{bools-of } rs \quad H \vdash_{\text{exp}} e \Downarrow es \quad \forall k, \vdash_{\text{node}} f(\text{mask}_r^k es) \Downarrow \text{mask}_r^k xs \quad H(x) = xs}{H \vdash_{\text{eqn}} x = (\text{restart } f \text{ every } y)(e)}$$

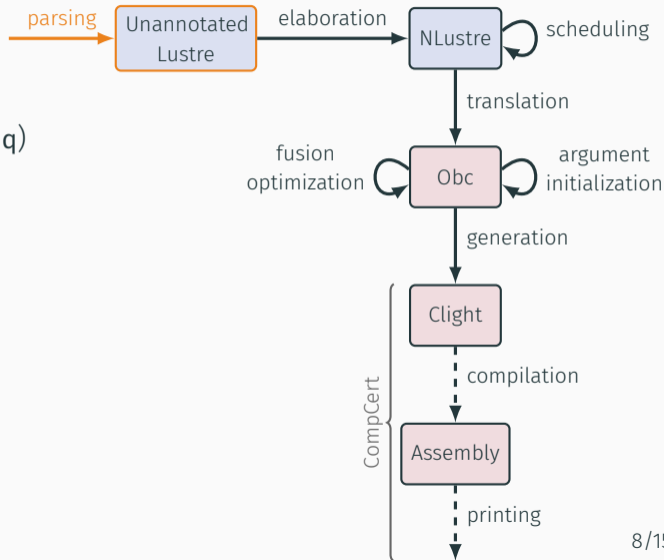
Universally quantified relation: unbounded number of constraints

VÉLUS: A VERIFIED LUSTRE COMPILER

Implemented in Coq and (some) OCaml

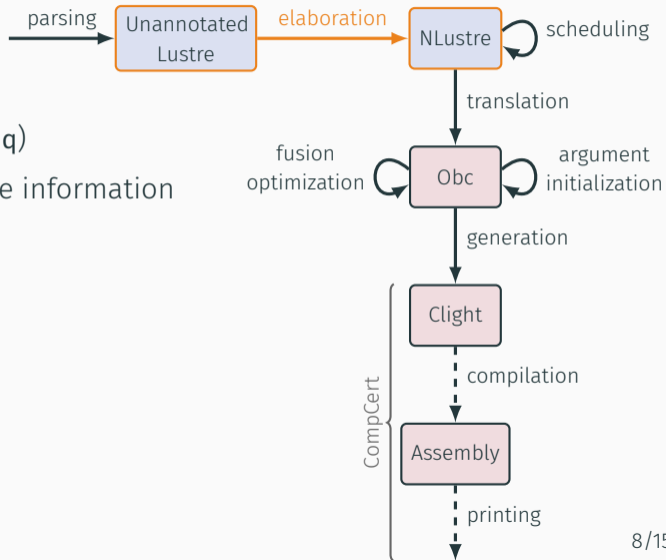


- validated `parsing` (`menhir --coq`)



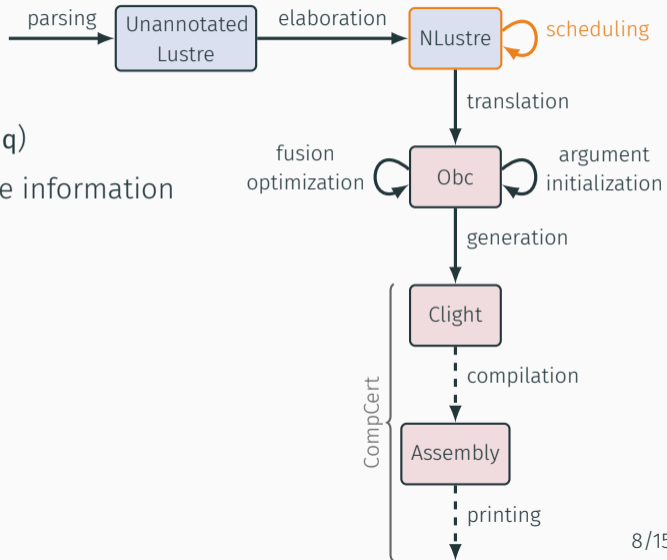
VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- **elaboration** to get clock and type information



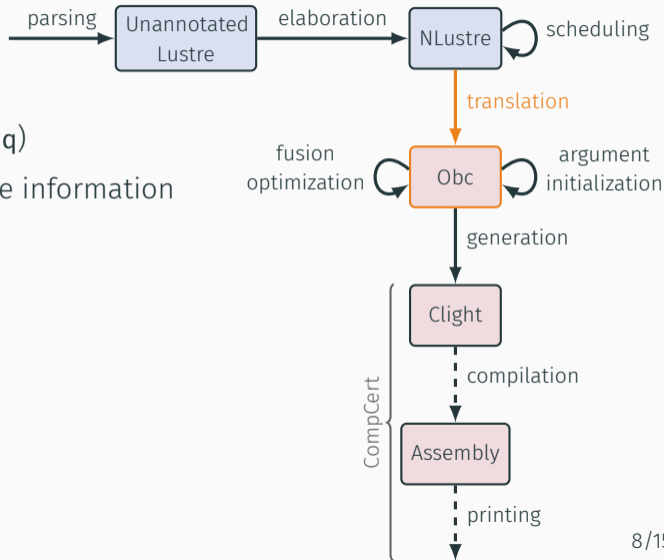
VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- **scheduling** of NLustre code



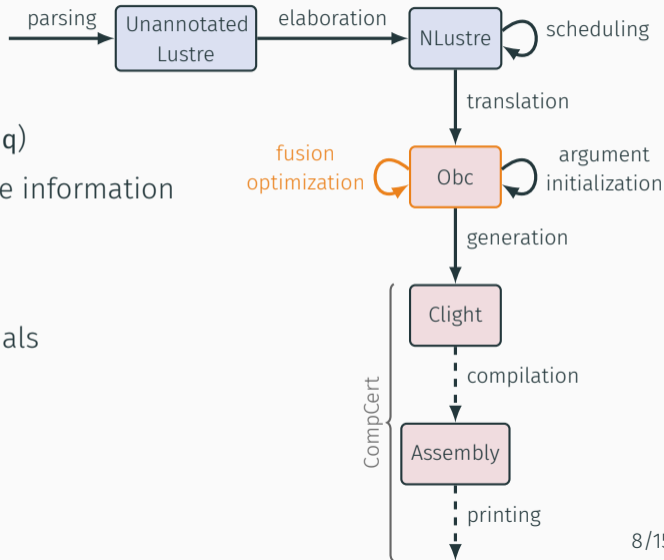
VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- **translation** to Obc code



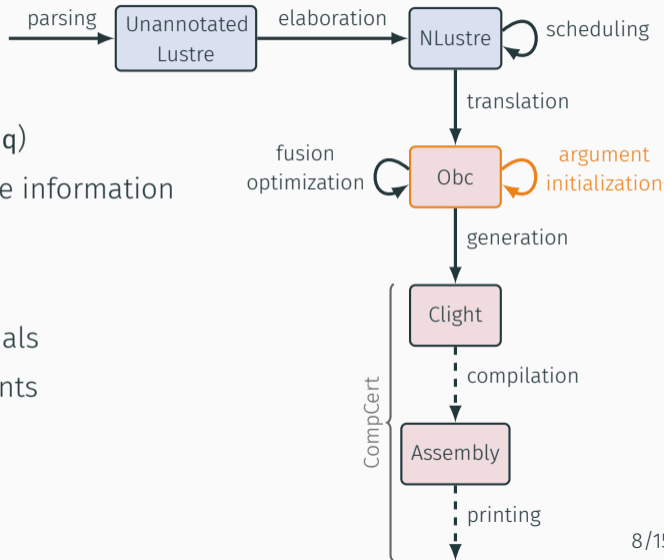
VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- translation to Obc code
- **fusion optimization** of conditionals



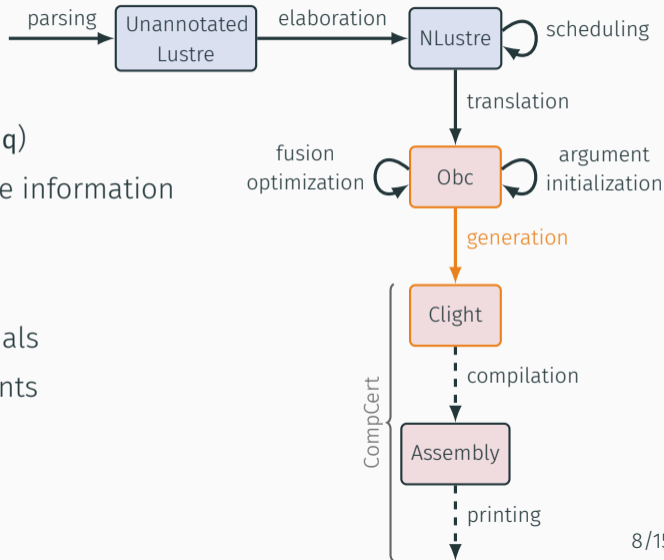
VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- translation to Obc code
- fusion optimization of conditionals
- **initialization** of variable arguments



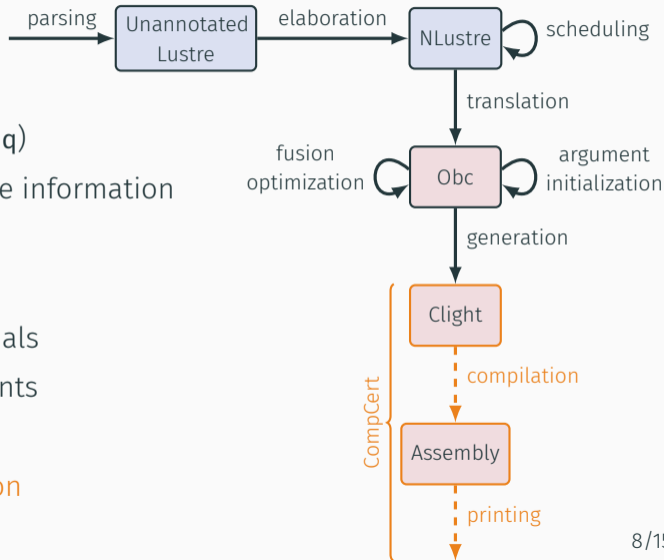
VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- translation to Obc code
- fusion optimization of conditionals
- initialization of variable arguments
- **Generation** of Clight code



VÉLUS: A VERIFIED LUSTRE COMPILER

- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- translation to Obc code
- fusion optimization of conditionals
- initialization of variable arguments
- Generation of Clight code
- Rely on CompCert for **compilation**



A PROBLEM WITH THE COMPILATION FROM NLUSTRE TO OBC

```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
class driver {
  instance x: ins, y: ins;

  reset() { ins(x).reset();
           ins(y).reset() }

  step(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool
  {
    if r { ins(x).reset() };
    x, ax := ins(x).step(x0, u);
    if r { ins(y).reset() };
    y, ay := ins(y).step(y0, v)
  }
}
```

A PROBLEM WITH THE COMPILATION FROM NLUSTRE TO OBC

```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
class driver {
  instance x: ins, y: ins;

  reset() { ins(x).reset();
           ins(y).reset() }

  step(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool
  {
    if r { ins(x).reset() };
    x, ax := ins(x).step(x0, u);
    if r { ins(y).reset() };
    y, ay := ins(y).step(y0, v)
  }
}
```

A PROBLEM WITH THE COMPILATION FROM NLUSTRE TO OBC

```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
class driver {
  instance x: ins, y: ins;

  reset() { ins(x).reset();
           ins(y).reset() }

  step(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool
  {
    if r { ins(x).reset() };
    x, ax := ins(x).step(x0, u);
    if r { ins(y).reset() };
    y, ay := ins(y).step(y0, v)
  }
}
```

A PROBLEM WITH THE COMPILATION FROM NLUSTRE TO OBC

```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
class driver {
  instance x: ins, y: ins;

  reset() { ins(x).reset();
           ins(y).reset() }

  step(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool
  {
    if r { ins(x).reset() };
    x, ax := ins(x).step(x0, u);
    if r { ins(y).reset() };
    y, ay := ins(y).step(y0, v)
  }
}
```

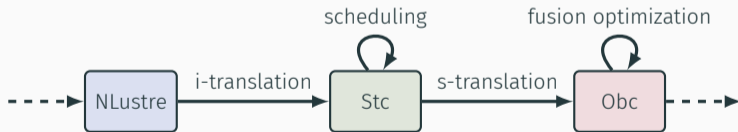
Propose a new intermediate language

- **Invariant semantics** under permutation
- **Separate reset** construct
- **Explicit state**: state variables and instances

STC: SYNCHRONOUS TRANSITION CODE

Propose a new intermediate language

- **Invariant semantics** under permutation
- **Separate reset** construct
- **Explicit state**: state variables and instances



COMPILATION WITH STC



```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
system driver {
  sub x: ins, y: ins;

  transition(x0, y0, u, v: double, r: bool)
    returns (x, y: double)
    var ax, ay: bool;
    {
      x, ax = ins<x>(x0, u);
      reset ins<x> every (. on r);
      y, ay = ins<y>(y0, v);
      reset ins<y> every (. on r);
    }
}
```

COMPILATION WITH STC



```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
system driver {
  sub x: ins, y: ins;

  transition(x0, y0, u, v: double, r: bool)
    returns (x, y: double)
    var ax, ay: bool;
    {
      x, ax = ins<x>(x0, u);
      reset ins<x> every (. on r);
      y, ay = ins<y>(y0, v);
      reset ins<y> every (. on r);
    }
}
```

COMPILATION WITH STC



```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
system driver {
  sub x: ins, y: ins;

  transition(x0, y0, u, v: double, r: bool)
    returns (x, y: double)
    var ax, ay: bool;
  {
    x, ax = ins<x>(x0, u);
    reset ins<x> every (. on r);
    y, ay = ins<y>(y0, v);
    reset ins<y> every (. on r);
  }
}
```

COMPILATION WITH STC

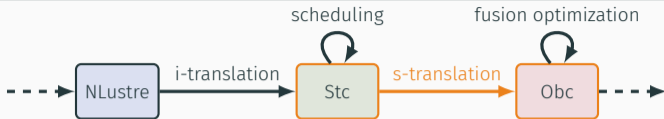


```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
system driver {
  sub x: ins, y: ins;

  transition(x0, y0, u, v: double, r: bool)
    returns (x, y: double)
    var ax, ay: bool;
  {
    reset ins<x> every (. on r);
    reset ins<y> every (. on r);
    x, ax = ins<x>(x0, u);
    y, ay = ins<y>(y0, v);
  }
}
```

COMPILATION WITH STC



```
system driver {  
  sub x: ins, y: ins;
```

```
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    reset ins<x> every (. on r);  
    reset ins<y> every (. on r);  
    x, ax = ins<x>(x0, u);  
    y, ay = ins<y>(y0, v);  
  }  
}
```

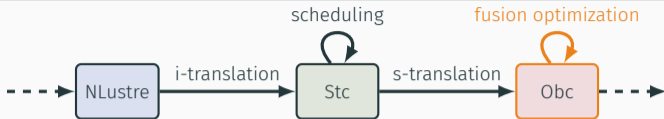
```
class driver {
```

```
  instance x: ins, y: ins;
```

```
  reset() { ins(x).reset();  
           ins(y).reset() }
```

```
  step(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool  
  {  
    if r { ins(x).reset() };  
    if r { ins(y).reset() };  
    x, ax := ins(x).step(x0, u);  
    y, ay := ins(y).step(y0, v)  
  }  
}
```

COMPILATION WITH STC



```
system driver {  
  sub x: ins, y: ins;
```

```
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    reset ins<x> every (. on r);  
    reset ins<y> every (. on r);  
    x, ax = ins<x>(x0, u);  
    y, ay = ins<y>(y0, v);  
  }  
}
```

```
class driver {
```

```
  instance x: ins, y: ins;
```

```
  reset() { ins(x).reset();  
           ins(y).reset() }
```

```
  step(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool  
  {  
    if r { ins(x).reset();  
          ins(y).reset() };  
    x, ax := ins(x).step(x0, u);  
    y, ay := ins(y).step(y0, v)  
  }  
}
```

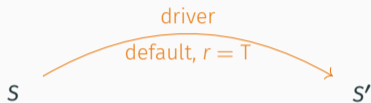
Transition system

- Start state S , end state S'
- Transition constraints
- Transient state l

Transition system

- Start state S , end state S'
- Transition constraints
- Transient state l

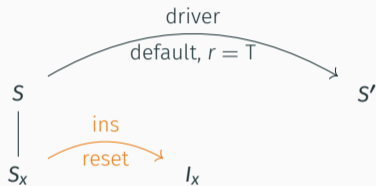
```
system driver {  
  sub x: ins, y: ins;  
  
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    x, ax = ins<x>(x0, u);  
    reset ins<x> every (. on r);  
    y, ay = ins<y>(y0, v);  
    reset ins<y> every (. on r);  
  }  
}
```



Transition system

- Start state S , end state S'
- Transition constraints
- Transient state I

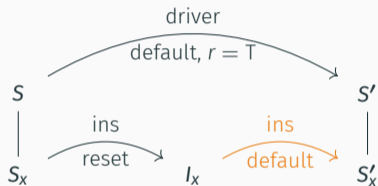
```
system driver {  
  sub x: ins, y: ins;  
  
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    x, ax = ins<x>(x0, u);  
    reset ins<x> every (. on r);  
    y, ay = ins<y>(y0, v);  
    reset ins<y> every (. on r);  
  }  
}
```



Transition system

- Start state S , end state S'
- Transition constraints
- Transient state I

```
system driver {  
  sub x: ins, y: ins;  
  
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    x, ax = ins<x>(x0, u);  
    reset ins<x> every (. on r);  
    y, ay = ins<y>(y0, v);  
    reset ins<y> every (. on r);  
  }  
}
```



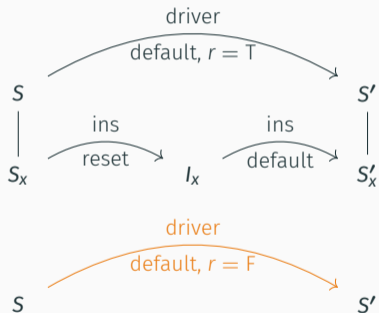
Transition system

- Start state S , end state S'
- Transition constraints
- Transient state I

```

system driver {
  sub x: ins, y: ins;

  transition(x0, y0, u, v: double, r: bool)
    returns (x, y: double)
    var ax, ay: bool;
  {
    x, ax = ins<x>(x0, u);
    reset ins<x> every (. on r);
    y, ay = ins<y>(y0, v);
    reset ins<y> every (. on r);
  }
}
    
```

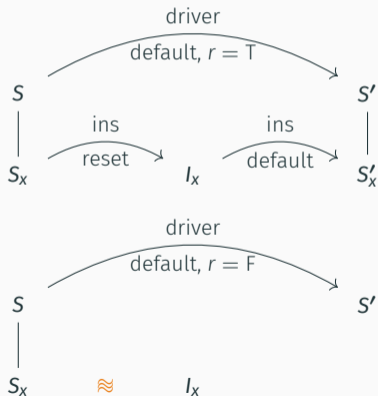


STC INTUITIVE SEMANTICS

Transition system

- Start state S , end state S'
- Transition constraints
- Transient state I

```
system driver {  
  sub x: ins, y: ins;  
  
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    x, ax = ins<x>(x0, u);  
    reset ins<x> every (. on r);  
    y, ay = ins<y>(y0, v);  
    reset ins<y> every (. on r);  
  }  
}
```

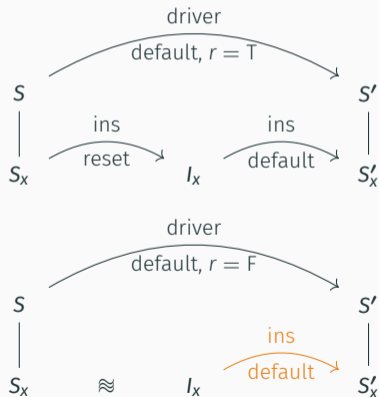


STC INTUITIVE SEMANTICS

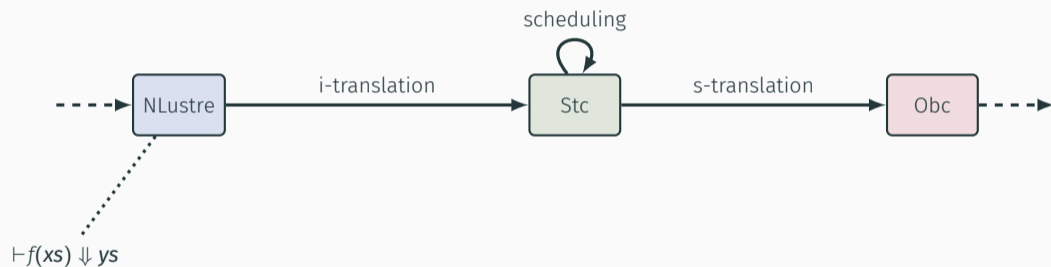
Transition system

- Start state S , end state S'
- Transition constraints
- Transient state I

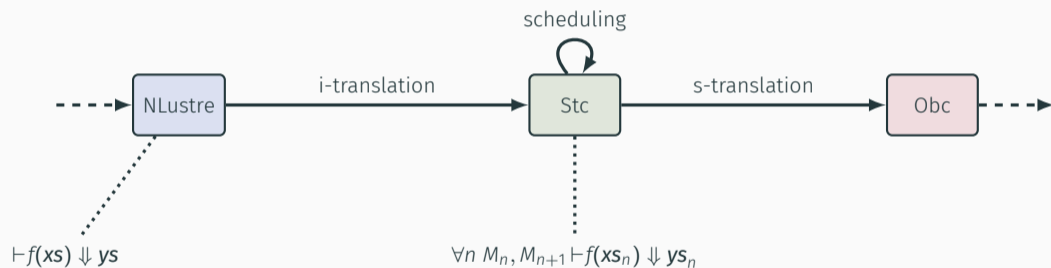
```
system driver {  
  sub x: ins, y: ins;  
  
  transition(x0, y0, u, v: double, r: bool)  
    returns (x, y: double)  
    var ax, ay: bool;  
  {  
    x, ax = ins<x>(x0, u);  
    reset ins<x> every (. on r);  
    y, ay = ins<y>(y0, v);  
    reset ins<y> every (. on r);  
  }  
}
```



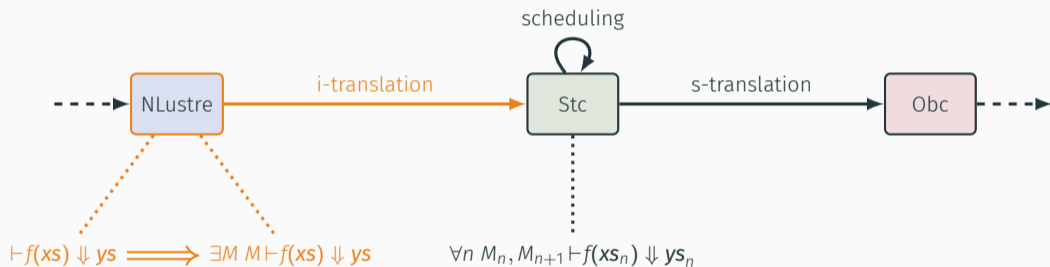
CORRECTNESS: PRESERVATION OF THE SEMANTICS



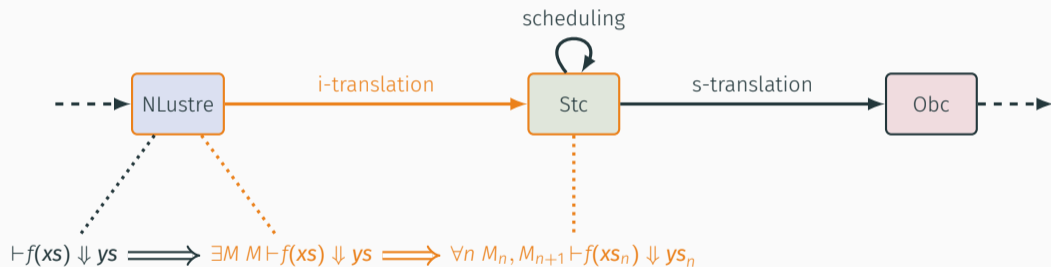
CORRECTNESS: PRESERVATION OF THE SEMANTICS



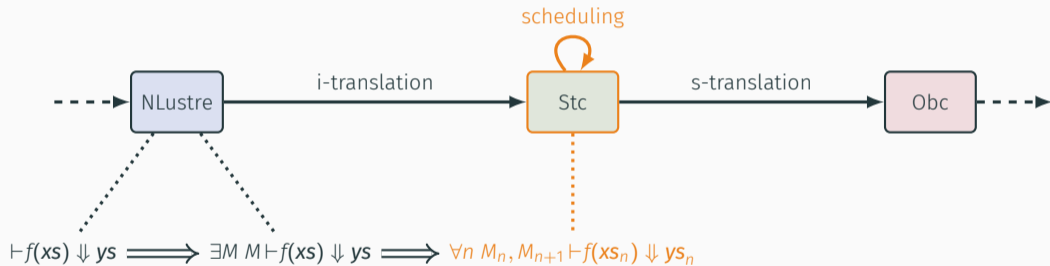
CORRECTNESS: PRESERVATION OF THE SEMANTICS



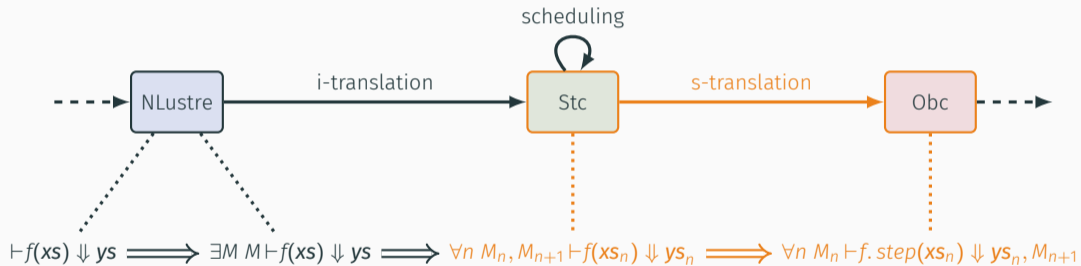
CORRECTNESS: PRESERVATION OF THE SEMANTICS



CORRECTNESS: PRESERVATION OF THE SEMANTICS



CORRECTNESS: PRESERVATION OF THE SEMANTICS



Theorem (Vélus correctness)

Given a list of declarations D , a name f , lists of streams of values \mathbf{xs} and \mathbf{ys} , an NLustre program G and an assembly program P such that $\text{compile } D \ f = \text{OK } (G, P)$ and $G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$, then there exists an infinite trace of events T such that

$$P \Downarrow_{ASM} \text{Reacts}(T) \quad \text{and} \quad \text{bisim-IO}^G f \ \mathbf{xs} \ \mathbf{ys} \ T$$

Theorem (Vélus correctness)

Given a list of declarations D , a name f , lists of streams of values \mathbf{xs} and \mathbf{ys} , an NLustre program G and an assembly program P such that $\text{compile } D \ f = \text{OK } (G, P)$ and $G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$, then there exists an infinite trace of events T such that

$$P \Downarrow_{ASM} \text{Reacts}(T) \quad \text{and} \quad \text{bisim-IO}^G f \ \mathbf{xs} \ \mathbf{ys} \ T$$

Theorem (Vélus correctness)

Given a list of declarations D , a name f , lists of streams of values \mathbf{xs} and \mathbf{ys} , an NLustre program G and an assembly program P such that $\text{compile } D \ f = \text{OK } (G, P)$ and $G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$, then there exists an infinite trace of events T such that

$$P \Downarrow_{ASM} \text{Reacts}(T) \quad \text{and} \quad \text{bisim-IO}^G f \ \mathbf{xs} \ \mathbf{ys} \ T$$

Theorem (Vélus correctness)

Given a list of declarations D , a name f , lists of streams of values \mathbf{xs} and \mathbf{ys} , an NLustre program G and an assembly program P such that $\text{compile } D \ f = \text{OK } (G, P)$ and $G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$, then there exists an infinite trace of events T such that

$$P \Downarrow_{ASM} \text{Reacts}(T) \quad \text{and} \quad \text{bisim-IO}^G f \ \mathbf{xs} \ \mathbf{ys} \ T$$

Theorem (Vélus correctness)

Given a list of declarations D , a name f , lists of streams of values \mathbf{xs} and \mathbf{ys} , an NLustre program G and an assembly program P such that $\text{compile } D \ f = \text{OK } (G, P)$ and $G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$, then there exists an infinite trace of events T such that

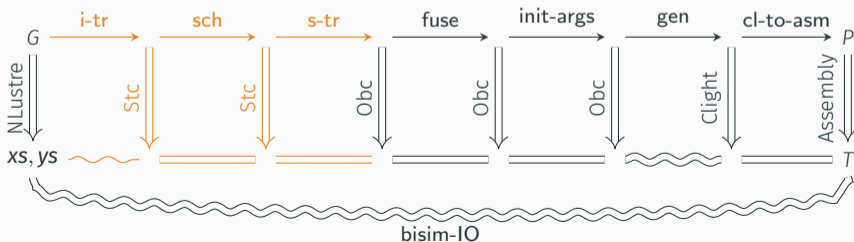
$$P \Downarrow_{ASM} \text{Reacts}(T) \quad \text{and} \quad \text{bisim-IO}^G f \ \mathbf{xs} \ \mathbf{ys} \ T$$

ULTIMATE THEOREM

Theorem (Vélus correctness)

Given a list of declarations D , a name f , lists of streams of values xs and ys , an NLustre program G and an assembly program P such that $\text{compile } D \ f = \text{OK } (G, P)$ and $G \vdash f(xs) \Downarrow ys$, then there exists an infinite trace of events T such that

$$P \Downarrow_{ASM} \text{Reacts}(T) \quad \text{and} \quad \text{bisim-IO}^G \ f \ xs \ ys \ T$$



Contributions:

- A verified compiler for Lustre with reset
- A single additional semantic rule for the reset
- An intermediate transition system language: Stc

Next goal: State machines

`velus.inria.fr`
`github.com/INRIA/velus`

REFERENCES I

- Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Alexander Plaice (1987). “LUSTRE: A Declarative Language for Programming Synchronous Systems”. In: *In 14th Symposium on Principles of Programming Languages (POPL'87)*. ACM.
- Paul Caspi (Jan. 1, 1994). “Towards Recursive Block Diagrams”. In: *Annual Review in Automatic Programming* 18, pp. 81–85.
- Grégoire Hamon and Marc Pouzet (2000). “Modular Resetting of Synchronous Data-Flow Programs”. In: *Proceedings of the 2Nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '00. New York, NY, USA: ACM, pp. 289–300.
- Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet (2005). “A Conservative Extension of Synchronous Data-Flow with State Machines”. In: *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT '05. New York, NY, USA: ACM, pp. 173–182.
- Sandrine Blazy and Xavier Leroy (Oct. 1, 2009). “Mechanized Semantics for the Clight Subset of the C Language”. In: *Journal of Automated Reasoning* 43.3, pp. 263–288.

REFERENCES II

- Xavier Leroy (July 2009). “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7, pp. 107–115.
- Jacques-Henri Jourdan, François Pottier, and Xavier Leroy (2012). “Validating LR(1) Parsers”. In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 397–416.
- Timothy Bourke, Lélío Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg (2017). “A Formally Verified Compiler for Lustre”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM, pp. 586–601.
- Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet (Sept. 2017). “SCADE 6: A Formal Language for Embedded Critical Software Development”. In: *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 1–11.