Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset

Timothy Bourke[2,3] Lélio Brun[1] Marc Pouzet [4,3,2]

POPL'20 — January 24, 2020

FAC'21 — October 14, 2021

[1]ISAE-SUPAERO

[2]Inria Paris

[3]École normale supérieure – PSL University

[4]Sorbonne University

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

Context: Embedded Systems Design

Embedded systems

- computer systems within physical systems that interact with the real world, often with real-time constraints
- software usually written in low-level languages: C, Ada, Assembly

Embedded systems

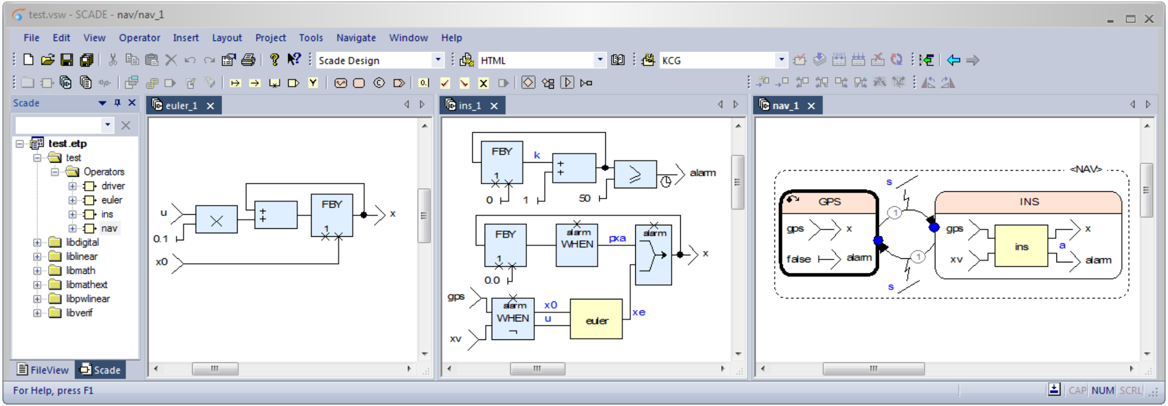
- computer systems within physical systems that interact with the real world, often with real-time constraints
- software usually written in low-level languages: C, Ada, Assembly

Model-Based Design

Executable high-level abstract specifications

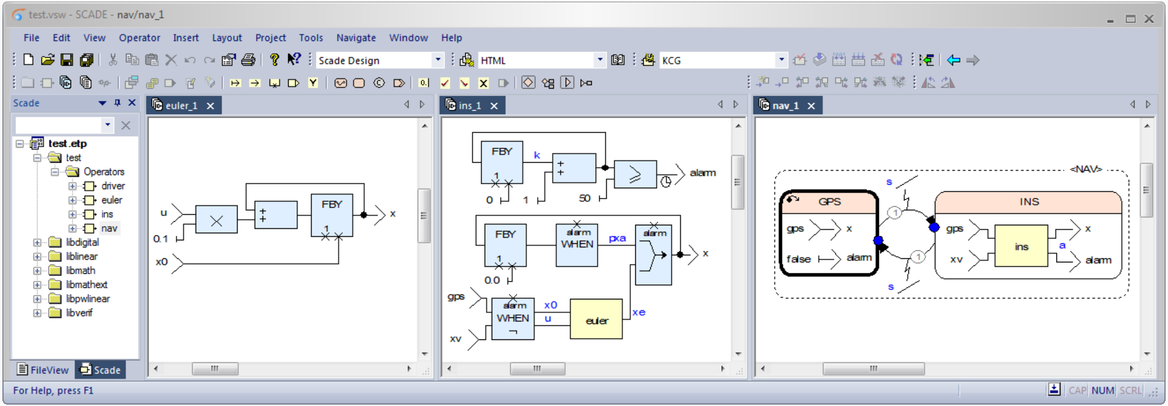
Motivation: Model Based Design in SCADE Suite

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

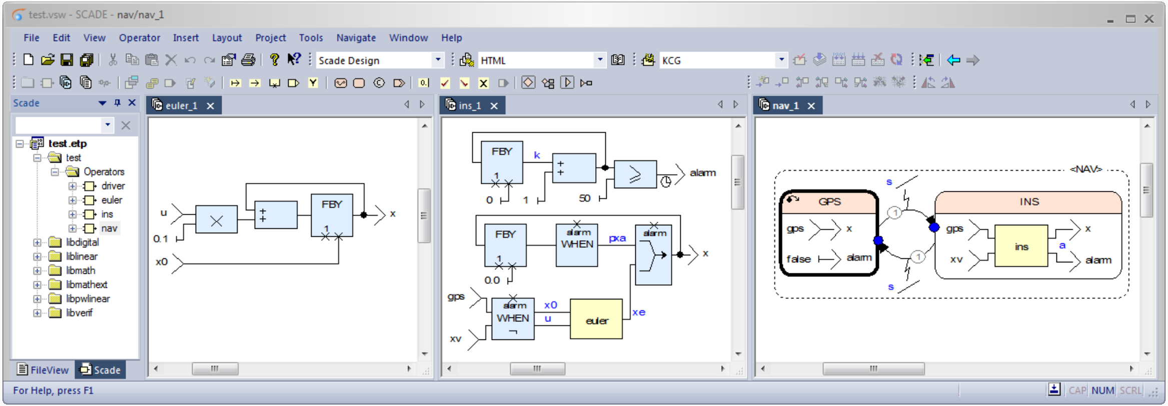


block / node = system

line = signal

Motivation: Model Based Design in SCADE Suite

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

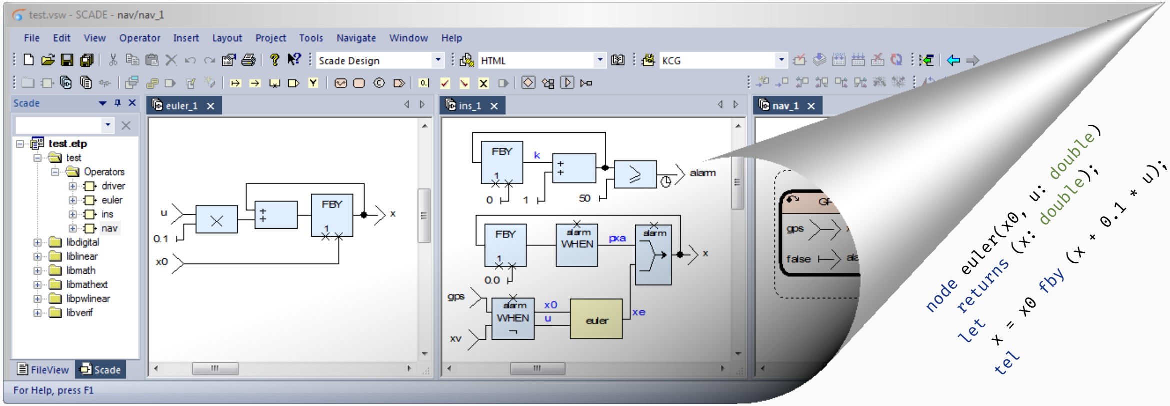


block / node = system = stream function

line = signal = stream of values

Motivation: Model Based Design in SCADE Suite

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

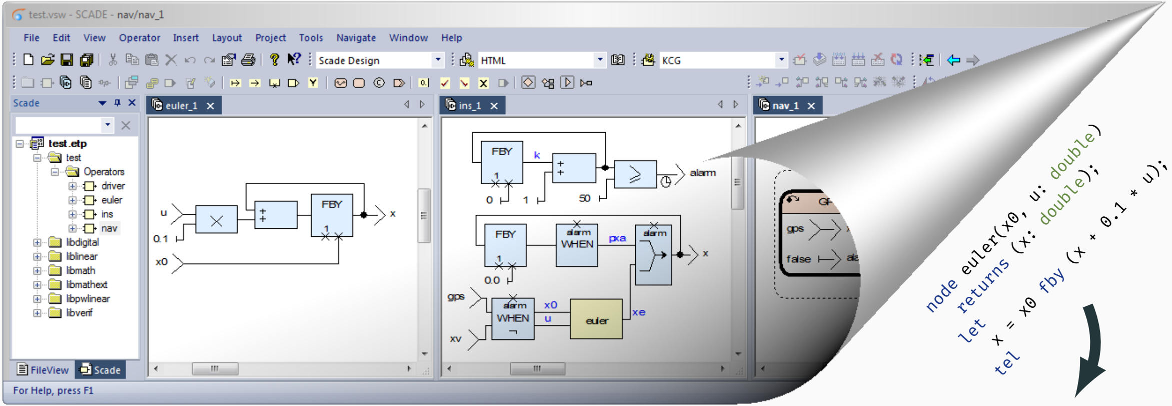


block / node = system = stream function

line = signal = stream of values

Motivation: Model Based Design in SCADE Suite

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



block / node = system = stream function

line = signal = stream of values

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 + Interactive Theorem Provers

Model-Based Design Languages
SCADE, Lustre, Simulink

+

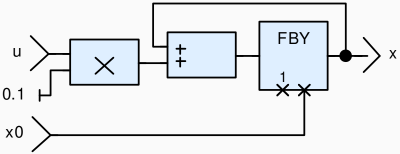
Interactive Theorem Provers
Coq

Challenges

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

Focus: modular reset

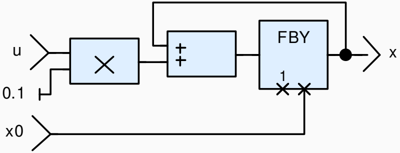
Example




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

Example

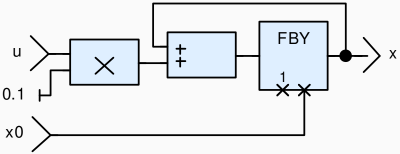
[Caspi et al. (1987); Colaço, Pagano, and Pouzet (2017)]




```
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	$\cdots$
u	15.00	20.00	17.00	12.00	$\cdots$
$x + 0.1 \times u$	1.50	3.50	5.20	6.70	$\cdots$
x	0.00	1.50	3.50	5.20	$\cdots$

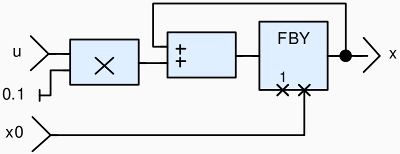
Example




```
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	$\cdots$
u	15.00	20.00	17.00	12.00	$\cdots$
$x + 0.1 \times u$	1.50	3.50	5.20	6.70	$\cdots$
x	0.00	1.50	3.50	5.20	$\cdots$

Example

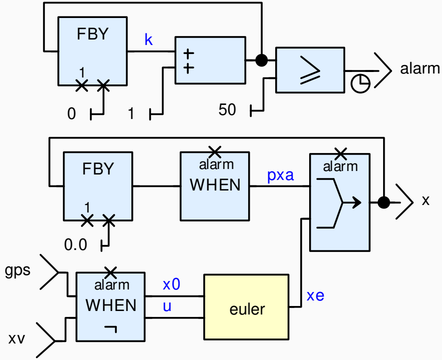



```
node euler(x0, u: double)
  returns (x: double);
let
  x = x0 fby (x + 0.1 * u);
tel
```

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

Example

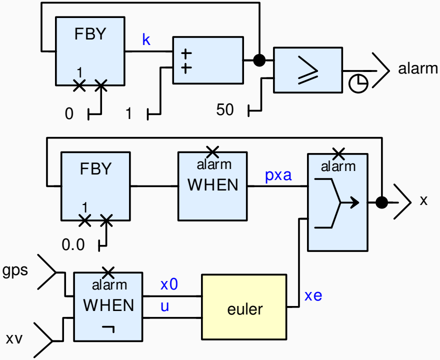
[Caspi et al. (1987); Colaço, Pagano, and Pouzet (2017)]




```
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

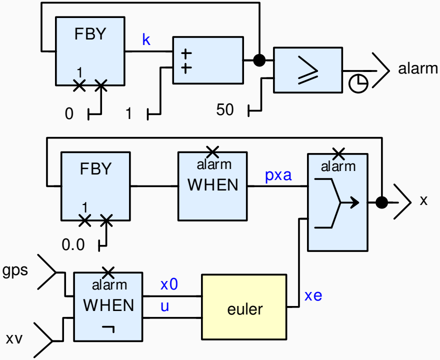



```
node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
let
  k = 0 fby (k + 1);
  alarm = (k ≥ 50);
  xe = euler((gps, xv) when not alarm);
  pxa = (0. fby x) when alarm;
  x = merge alarm pxa xe;
tel
```

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

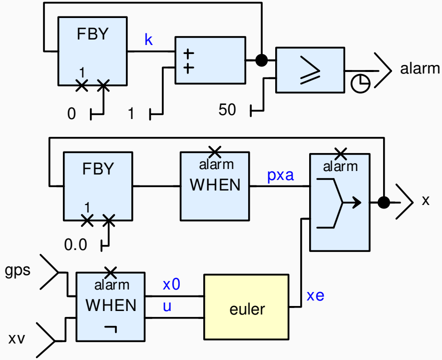
[Caspi et al. (1987); Colaço, Pagano, and Pouzet (2017)]




```
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	···

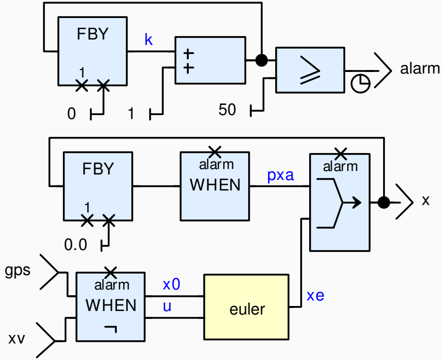
Example




```
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
```

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	…

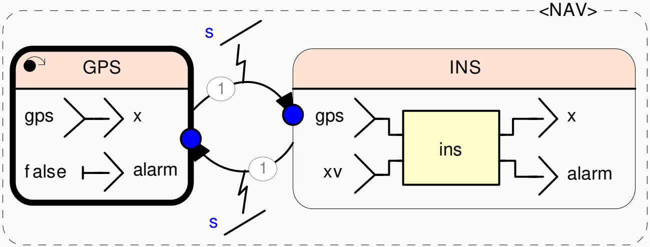
Example




```
node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var pxa, xe: double; k: int;
let
  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
```

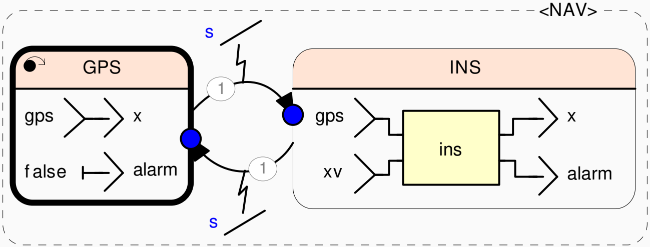
gps	0.00	1.55	3.62	5.46	···	86.52	88.40	90.91	···
xv	15.00	20.00	17.00	12.00	···	18.00	23.00	20.00	···
k	0	1	2	3	···	49	50	51	···
alarm	F	F	F	F	···	F	T	T	···
xe	0.00	1.50	3.50	5.20	···	77.35			···
pxa					···		77.35	77.35	···
x	0.00	1.50	3.50	5.20	···	77.35	77.35	77.35	···

Example



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) whenot alarm));
  alarm = (k ≥ 50);
  k = if r then 0 else 0 fby (k + 1);
tel
...
(x, a) = ins(gps, xv, r);
```

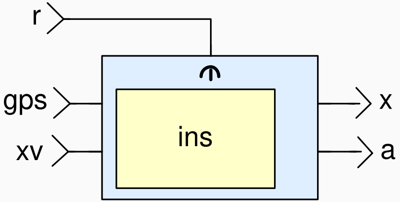
Without modular reset

```
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);
```

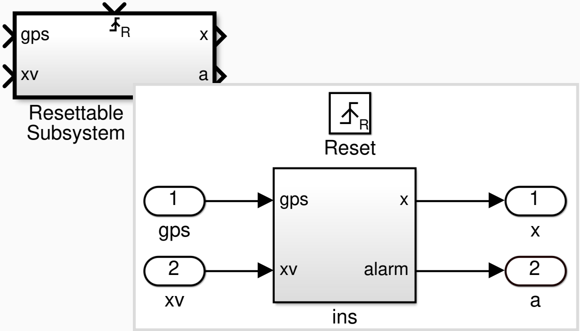
With modular reset

```
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);
```

SCADE

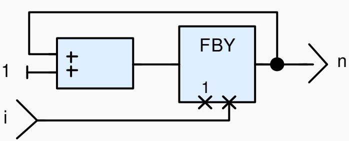


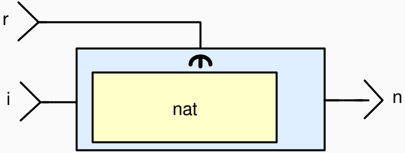
Simulink



A Simpler Example: Intuitive Semantics

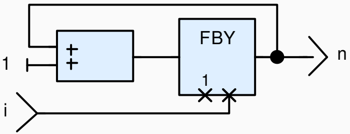
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```

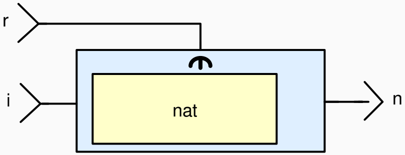




A Simpler Example: Intuitive Semantics

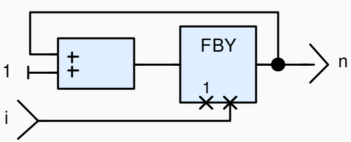
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```

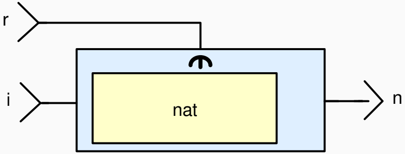




r	F
i	0
$nat(i)$	0
$(\texttt{restart}\ nat\ \texttt{every}\ r)(i)$	0

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

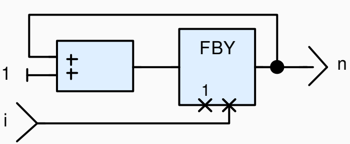


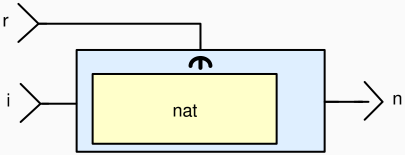


r	F	F
i	0	5
$nat(i)$	0	1
$(\texttt{restart}\ nat\ \texttt{every}\ r)(i)$	0	1

A Simpler Example: Intuitive Semantics

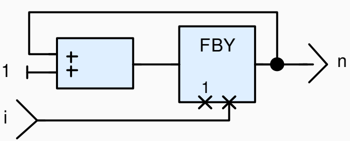
```
node nat(i: int)
  returns (n: int)
let
  n = i fby (n + 1);
tel
```

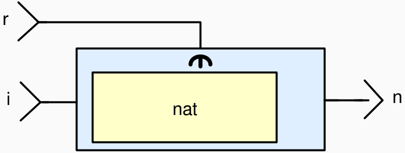




r	F	F	T
i	0	5	10
$nat(i)$	0	1	2
(**restart** nat **every** r)(i)	0	1	10

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

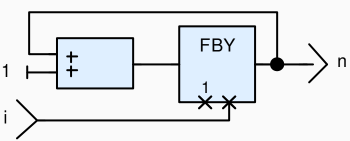


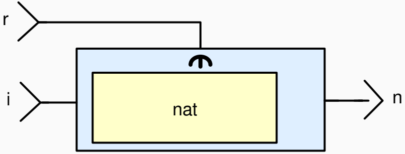


r	F	F	T	F
i	0	5	10	15
$nat(i)$	0	1	2	3
(**restart** nat **every** r)(i)	0	1	10	11

A Simpler Example: Intuitive Semantics

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

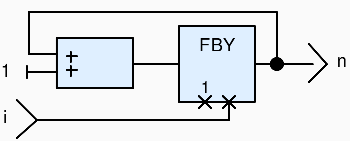


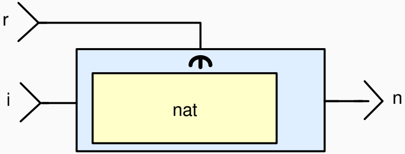


r	F	F	T	F	F
i	0	5	10	15	20
$nat(i)$	0	1	2	3	4
$(\texttt{restart}\ nat\ \texttt{every}\ r)(i)$	0	1	10	11	12

A Simpler Example: Intuitive Semantics

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

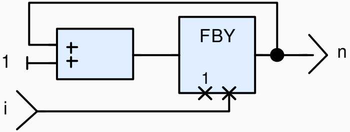


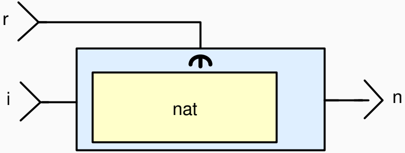


r	F	F	T	F	F	T
i	0	5	10	15	20	25
$nat(i)$	0	1	2	3	4	5
(**restart** nat **every** r)(i)	0	1	10	11	12	25

A Simpler Example: Intuitive Semantics

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

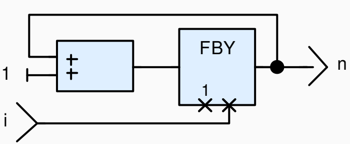


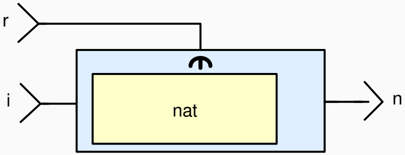


r	F	F	T	F	F	T	F
i	0	5	10	15	20	25	30
$nat(i)$	0	1	2	3	4	5	6
(**restart** *nat* **every** r)(i)	0	1	10	11	12	25	26

A Simpler Example: Intuitive Semantics

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

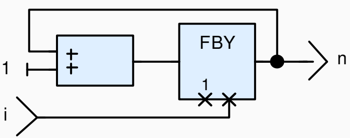


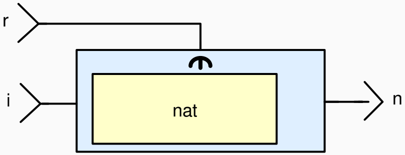


r	F	F	T	F	F	T	F	⋯
i	0	5	10	15	20	25	30	⋯
$nat(i)$	0	1	2	3	4	5	6	⋯
(**restart** nat **every** r)(i)	0	1	10	11	12	25	26	⋯

A Simpler Example: Intuitive Semantics


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





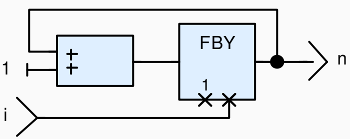
r	F	F	T	F	F	T	F	$\cdots$
i	0	5	10	15	20	25	30	$\cdots$
$nat(i)$	0	1	2	3	4	5	6	$\cdots$
(**restart** *nat* **every** *r*)(*i*)	0	1	10	11	12	25	26	$\cdots$

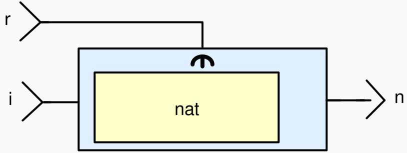
Could be implemented in a higher-order recursive language

A Simpler Example: Intuitive Semantics

[Hamon and Pouzet (2000)]

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





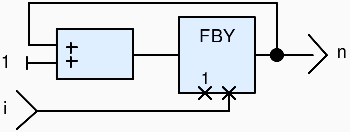
r	F	F	T	F	F	T	F	$\cdots$
i	0	5	10	15	20	25	30	$\cdots$
$nat(i)$	0	1	2	3	4	5	6	$\cdots$
(**restart** nat **every** r)(i)	0	1	10	11	12	25	26	$\cdots$

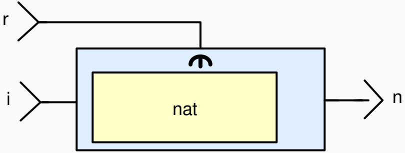
Could be implemented in a higher-order recursive language

A Simpler Example: Intuitive Semantics

[Hamon and Pouzet (2000)]

```
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	...
(**restart** nat **every** r)(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	$\cdots$
i	0	5	10	15	20	25	30	$\cdots$
(`restart` *nat* `every` r)(i)	0	1	10	11	12	25	26	$\cdots$

A Simpler Example: Intuitive Semantics

r	F	F	T	F	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	$\cdots$
i	0	5	10	15	20	25	30	$\cdots$
(`restart` *nat* `every` r)(i)	0	1	10	11	12	25	26	$\cdots$

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

A Simpler Example: Intuitive Semantics

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

A Simpler Example: Intuitive Semantics

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

A Simpler Example: Intuitive Semantics

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

A Simpler Example: Intuitive Semantics

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

A Simpler Example: Intuitive Semantics

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

A Simpler Example: Intuitive Semantics

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

Node instantiation

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

Node instantiation

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

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

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

Modular reset

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

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

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

Node instantiation

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

Modular reset

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

Node instantiation

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

Modular reset

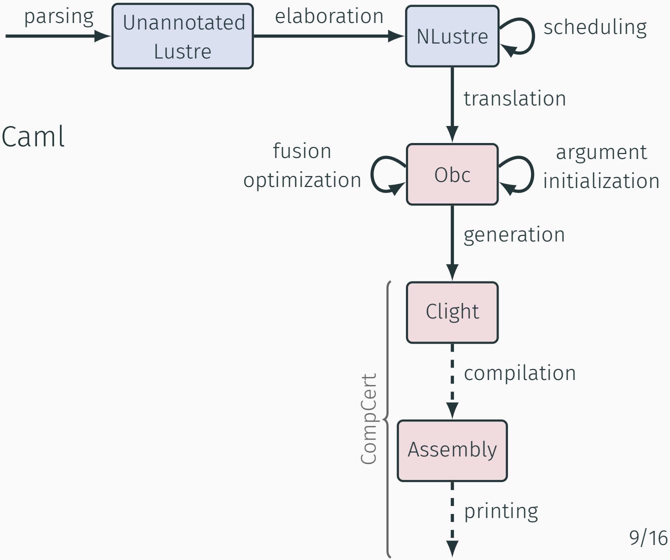
$$\frac{\begin{array}{c} H(y) = rs \quad r = \text{bools-of } rs \\ H \vdash_{\text{exp}} \boldsymbol{e} \Downarrow \boldsymbol{es} \quad \forall k,\ \vdash_{\text{node}} f(\text{mask}_r^k\, \boldsymbol{es}) \Downarrow \text{mask}_r^k\, \boldsymbol{xs} \quad H(\boldsymbol{x}) = \boldsymbol{xs} \end{array}}{H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

Universally quantified relation: unbounded number of constraints

Vélus: a Verified Lustre Compiler

[Bourke et al. (2017)]

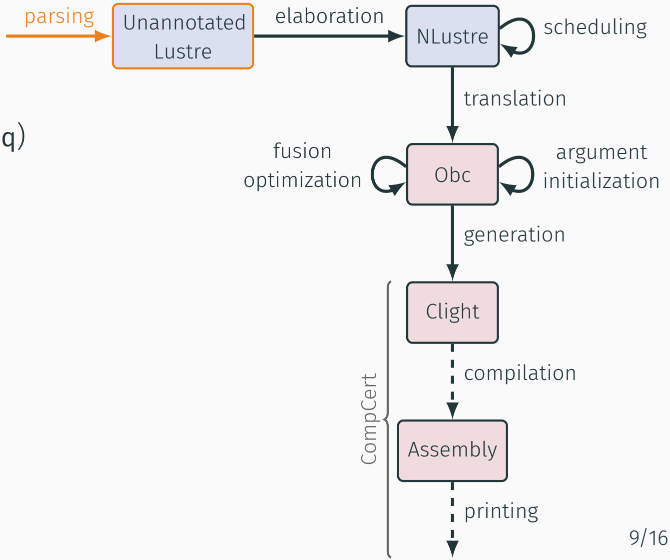
Implemented in Coq and (some) OCaml



Vélus: a Verified Lustre Compiler

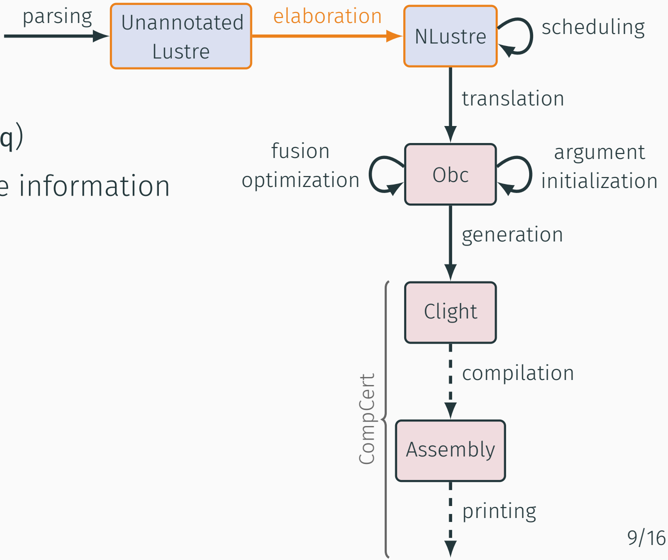
[Jourdan, Pottier, and Leroy (2012)]

- validated parsing (`menhir --coq`)



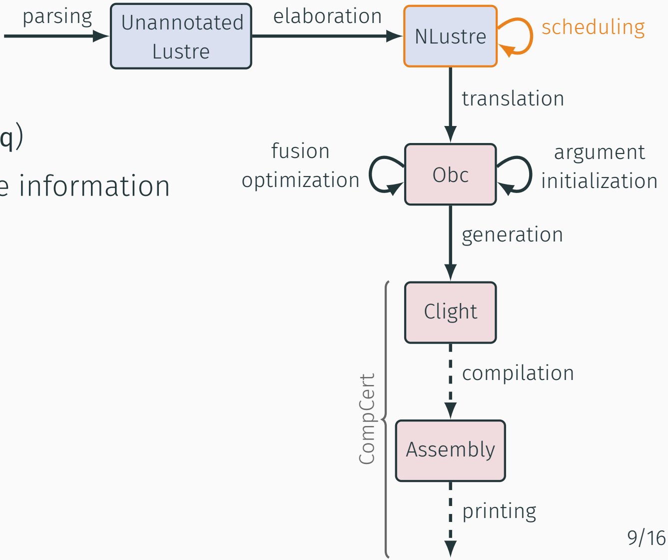
Vélus: a Verified Lustre Compiler

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

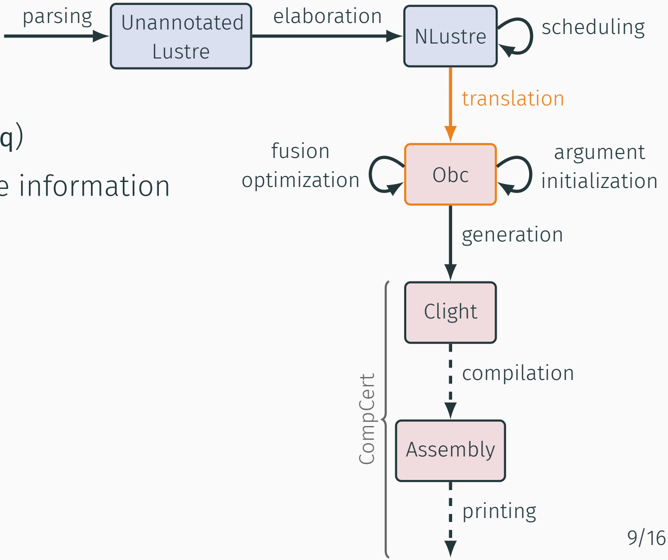


Vélus: a Verified Lustre Compiler

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



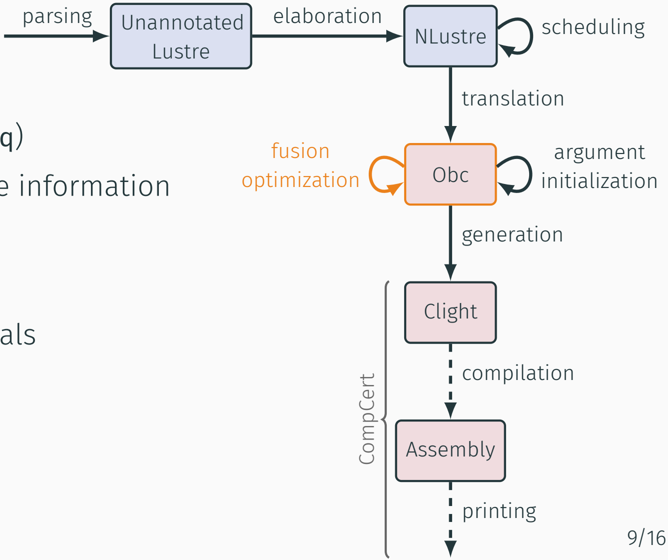
Vélus: a Verified Lustre Compiler



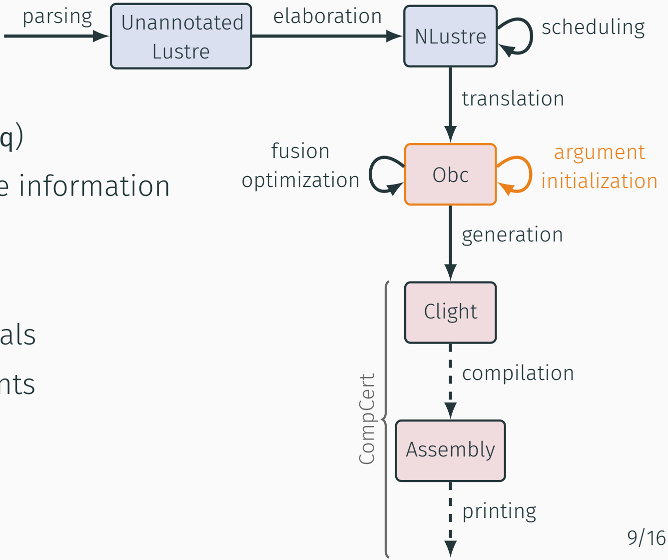
- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- translation to Obc code

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

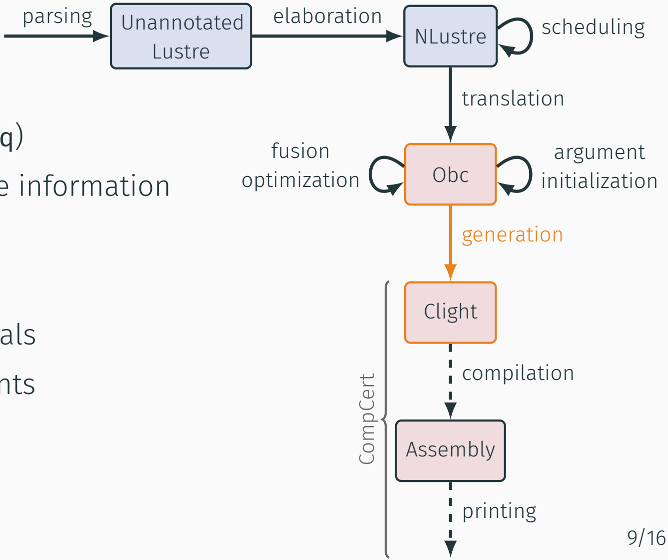


Vélus: a Verified Lustre Compiler



- 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

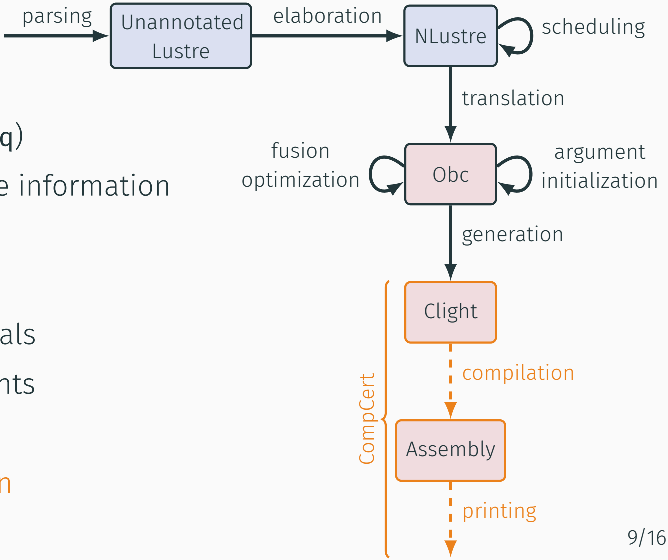


- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- scheduling of NLustre code
- translation to Obc code
- fusion optimization of conditionals
- initialization of variable arguments
- generation of Clight code

Vélus: a Verified Lustre Compiler

[Leroy (2009)]

- 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)
  }
}
```

Stc: Synchronous Transition Code

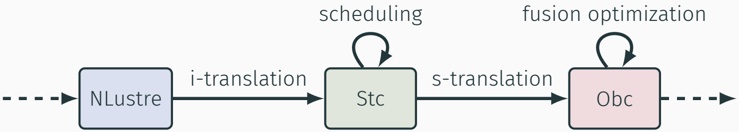
Propose a new intermediate language

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

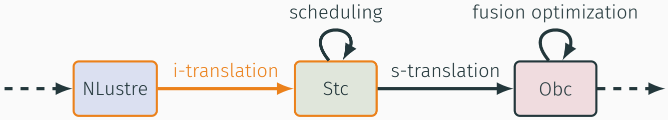
Stc: Synchronous Transition Code

Propose a new intermediate language

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



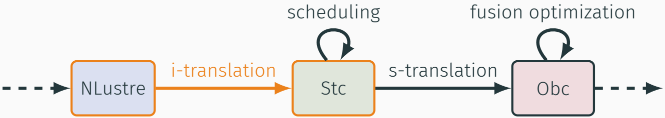
Compilation with Stc




```
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);
  }
}
```

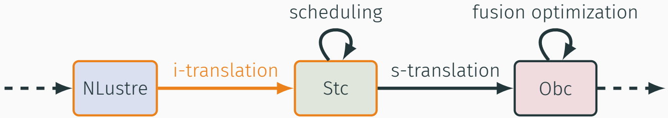
Compilation with Stc




```
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
```

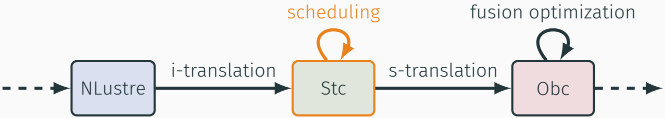
```
system driver {
  sub x: ins, y: ins;

  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



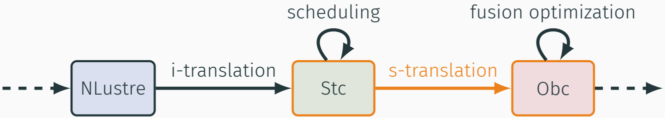

```
node driver(x0, y0, u, v: double, r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

```
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);
  }
}
```




```
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
```

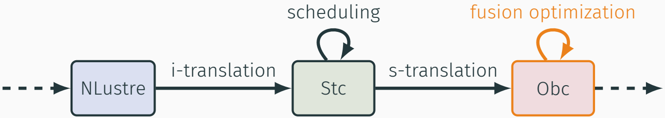
```
system driver {
  sub x: ins, y: ins;

  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);
  }
}
```




```
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

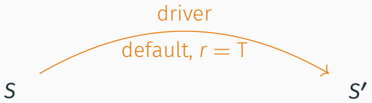
- Start state S, end state S'
- Transition constraints
- Transient state I

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);
  }
}
```

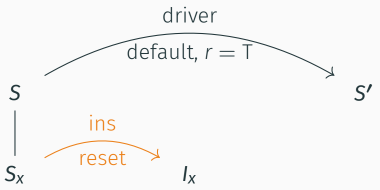


Stc Intuitive Semantics

Transition system

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

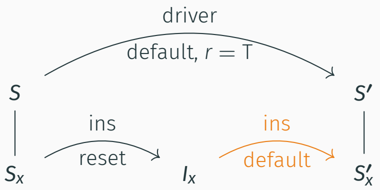
```
system driver {
  sub x: ins, y: ins;

  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);
  }
}
```



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);
  }
}
```

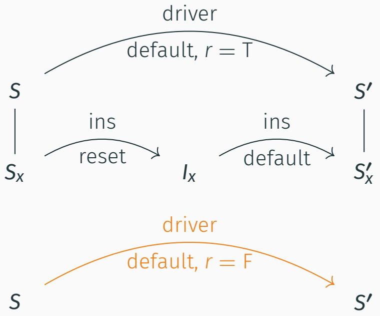


STC INTUITIVE SEMANTICS

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);
  }
}
```

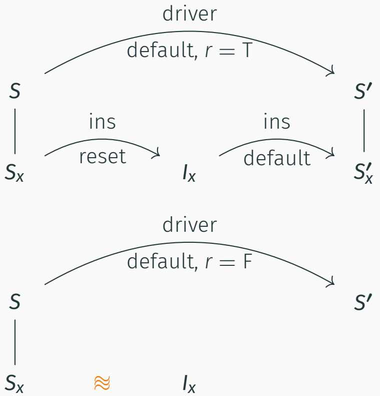


Stc Intuitive Semantics

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);
  }
}
```

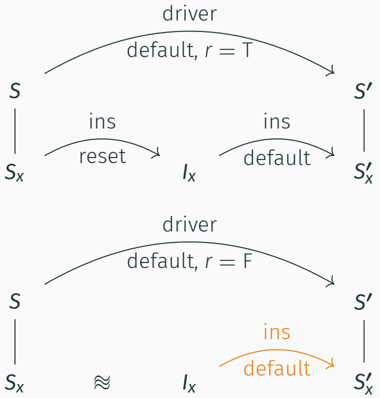


Stc Intuitive Semantics

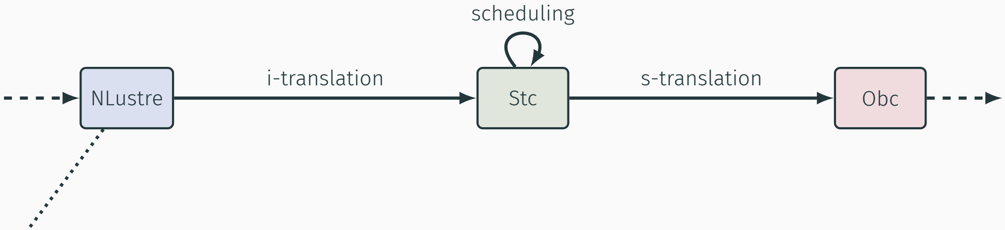
Transition system

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

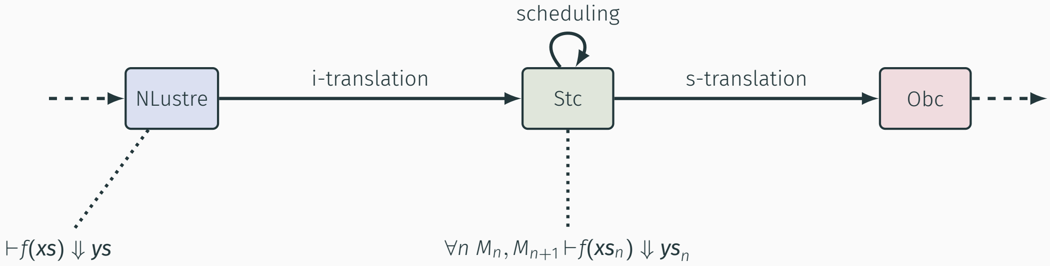
```
system driver {
  sub x: ins, y: ins;

  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);
  }
}
```



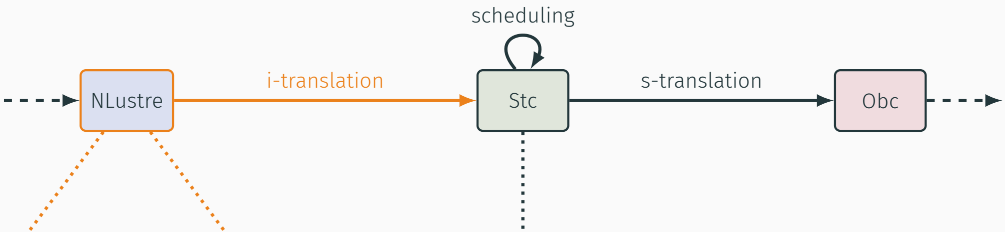
Correctness: Preservation of the Semantics



$\vdash f(xs) \Downarrow ys$


scheduling
NLustre
i-translation
Stc
s-translation
Obc
$\vdash f(xs) \Downarrow ys$
$\forall n\ M_n, M_{n+1} \vdash f(xs_n) \Downarrow ys_n$

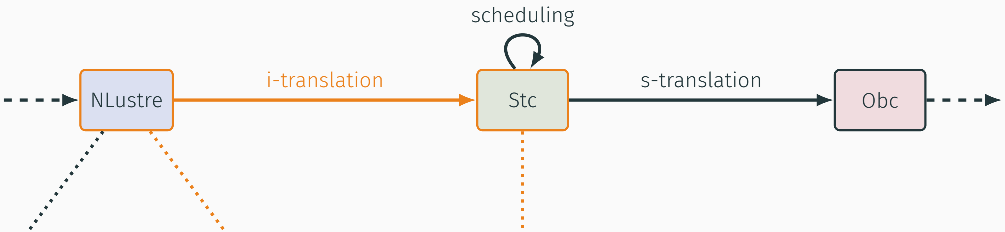
Correctness: Preservation of the Semantics



$\vdash f(xs) \Downarrow ys \Longrightarrow \exists M\ M \vdash f(xs) \Downarrow ys$

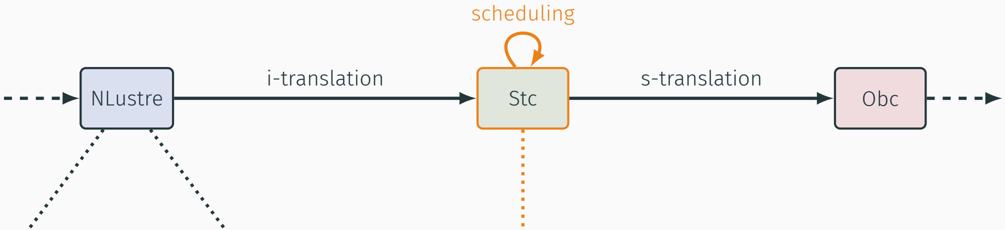
$\forall n\ M_n, M_{n+1} \vdash f(xs_n) \Downarrow ys_n$

Correctness: Preservation of the Semantics

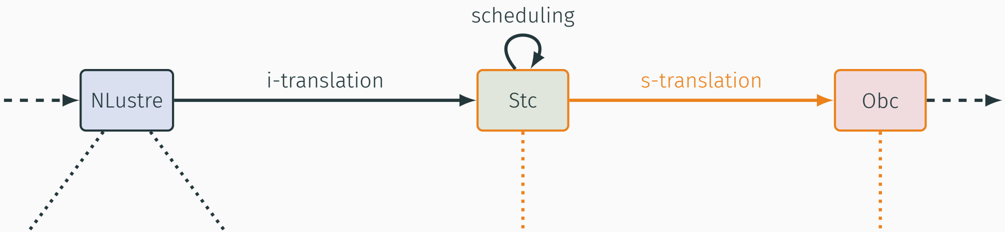


$$\vdash f(xs) \Downarrow ys \implies \exists M\ M \vdash f(xs) \Downarrow ys \implies \forall n\ M_n, M_{n+1} \vdash f(xs_n) \Downarrow ys_n$$

Correctness: Preservation of the Semantics



$$\vdash f(xs) \Downarrow ys \Longrightarrow \exists M\ M \vdash f(xs) \Downarrow ys \Longrightarrow \forall n\ M_n, M_{n+1} \vdash f(xs_n) \Downarrow ys_n$$



$$\vdash f(xs) \Downarrow ys \Longrightarrow \exists M\ M \vdash f(xs) \Downarrow ys \Longrightarrow \forall n\ M_n, M_{n+1} \vdash f(xs_n) \Downarrow ys_n \Longrightarrow \forall n\ M_n \vdash f.\mathit{step}(xs_n) \Downarrow ys_n, M_{n+1}$$

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 $\mathsf{compile}\, D\, f = \mathsf{OK}\,(G, P)$ and $G \vdash f\,(\mathbf{xs}) \Downarrow \mathbf{ys}$, then there exists an infinite trace of events T such that

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

Ultimate Theorem

Theorem (Vélus correctness)

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

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

Theorem (Vélus correctness)

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

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

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* $\mathsf{compile}\, D\, f = \mathrm{OK}\,(G, P)$ *and* $G \vdash f\,(\mathbf{xs}) \Downarrow \mathbf{ys}$*, then there exists an infinite trace of events T such that*

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

Theorem (Vélus correctness)

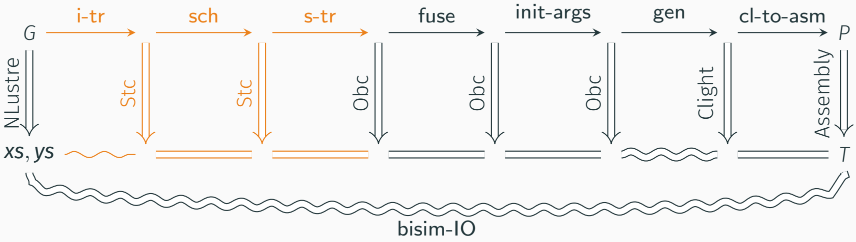
Given a list of declarations D, a name f, lists of streams of values **xs** *and* **ys***, an NLustre program G and an assembly program P such that* $\mathsf{compile}\, D\, f = \mathsf{OK}\,(G, P)$ *and* $G \vdash f\,(\mathbf{xs}) \Downarrow \mathbf{ys}$*, then there exists an infinite trace of events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

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

Theorem (Vélus correctness)

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

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



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, Lélio 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.