Verified Compilation of the Modular Reset, Finally

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

SYNCHRON'19 — November 28, 2019

[1]Inria Paris

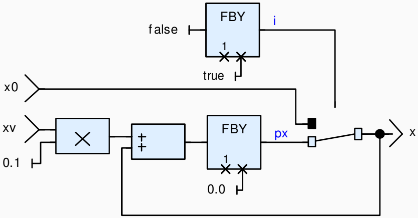
[2]DI ENS – PSL University

[3]Sorbonne University

Adding the modular reset to Vélus

- Compilation and optimization
- Semantic model
- Proof of correctness

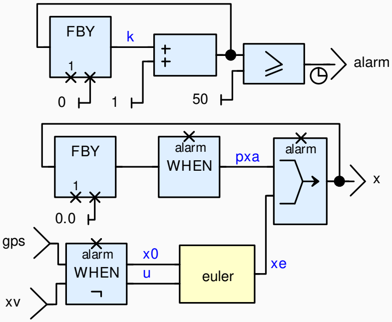
A Normalized Lustre Example




```
node euler(x0, u: double)
  returns (x: double);
  var i: bool, px: double;
let
  i = true fby false;
  x = if i then x0 else px;
  px = 0.0 fby (x + 0.1 * u);
tel
```

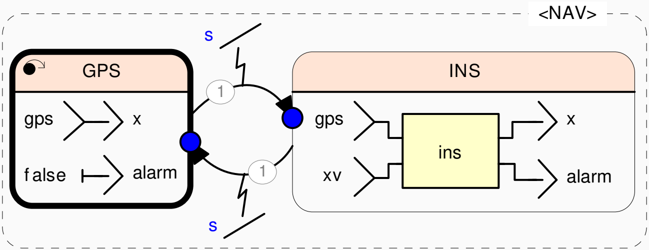
A Normalized Lustre Example

[Cas+87; CPP17]

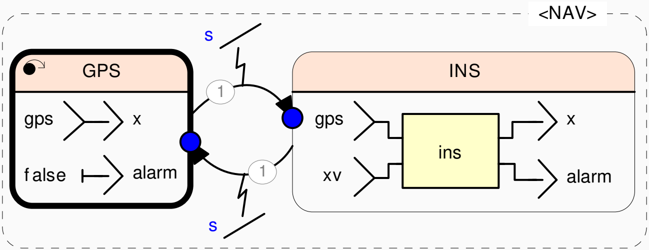



```
node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var k: int, px: double,
      xe: double whenot alarm;
let
  k = 0 fby k + 1;
  alarm = (k ≥ 50);
  xe = euler(gps whenot alarm,
             xv whenot alarm);
  x = merge alarm (px when alarm) xe;
  px = 0. fby x;
tel
```

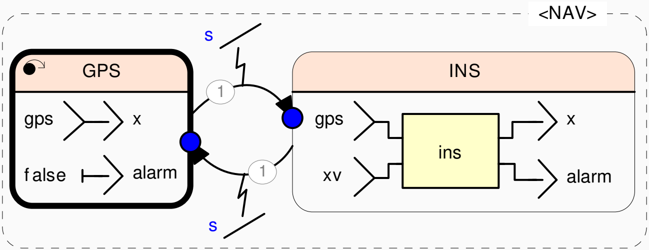
SCADE-like State Machines and Reset Primitive



SCADE-like State Machines and Reset Primitive



Can be compiled into Lustre



Can be compiled into Lustre

Reset:

- Reset the state of a node, ie. reinitialize the `fby`s
- Useful primitive, not only for state machines
- How?

A Simpler Example

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

A Simpler Example

```
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

A Simpler Example

```
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

```
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

A Simpler Example

```
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
$(\texttt{restart}\ nat\ \texttt{every}\ r)(i)$	0	1	10	11

A Simpler Example

```
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
(**restart** nat **every** r)(i)	0	1	10	11	12

A Simpler Example

```
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

```
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

```
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	$\cdots$
$(\texttt{restart}\ nat\ \texttt{every}\ r)(i)$	0	1	10	11	12	25	26	$\cdots$

A Simpler Example

```
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	···
($\texttt{restart}$ nat $\texttt{every}$ r)(i)	0	1	10	11	12	25	26	···

A Simpler Example

```
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	$\cdots$

A Simpler Example

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

A Recursive Intuition

[Cas94; CP97; HP00]

```
node whilenot(r: bool) returns (c: bool)
let
  c = true → if r then false else (pre c);
tel

node reset_nat(i: int, r: bool) returns (y: int)
  var c: bool;
let
  c = whilenot(r);
  y = merge c (nat(i when c))
              (reset_nat((i, r) whenot c));
tel
```

r	F	F	T	F	F	T	F	⋯
c	T	T	F	F	F	F	F	⋯

Infinitely Unrolling the Recursion

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	$\cdots$

Infinitely Unrolling the Recursion

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	$\cdots$

Infinitely Unrolling the Recursion

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	$\cdots$

Infinitely Unrolling the Recursion

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	$\cdots$

Infinitely Unrolling the Recursion

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	$\cdots$

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			$\cdots$

(`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						$\cdots$
$nat(\text{mask}_r^0\ i)$	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	$\cdots$
(`restart` nat `every` r)(i)	0	1	10	11	12	25	26	$\cdots$

Infinitely Unrolling the Recursion

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	$\cdots$

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 application

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

Node application

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

Node application

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

Node application

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

Node application

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

Modular reset

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

Node application

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

Modular reset

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

Node application

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

Modular reset

$$\frac{\vdash_{\text{exp}} e \Downarrow es \qquad \begin{array}{c}\vdash_{\text{var}} y \Downarrow rs\\ \end{array} \qquad \vdash_{\text{var}} x \Downarrow xs}{\vdash_{\text{eqn}} x = (\texttt{restart}\ f\ \texttt{every}\ y)(e)}$$

Node application

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

Modular reset

$$\frac{\begin{array}{c} \vdash_{\text{var}} y \Downarrow rs \quad r = \text{bools-of } rs \\ \vdash_{\text{exp}} \boldsymbol{e} \Downarrow \boldsymbol{es} \qquad\qquad \vdash_{\text{var}} \boldsymbol{x} \Downarrow \boldsymbol{xs} \end{array}}{\vdash_{\text{eqn}} \boldsymbol{x} = (\texttt{restart}\ f\ \texttt{every}\ y)(\boldsymbol{e})}$$

Node application

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

Modular reset

$$\frac{\begin{array}{c}\vdash_{\text{var}} y \Downarrow rs \quad r = \text{bools-of } rs \\ \vdash_{\text{exp}} \boldsymbol{e} \Downarrow \boldsymbol{es} \quad r \vdash_{\text{reset}} f(\boldsymbol{es}) \Downarrow \boldsymbol{xs} \quad \vdash_{\text{var}} \boldsymbol{x} \Downarrow \boldsymbol{xs}\end{array}}{\vdash_{\text{eqn}} \boldsymbol{x} = (\texttt{restart}\ f\ \texttt{every}\ y)(\boldsymbol{e})}$$

Node application

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

Modular reset

$$\frac{\begin{array}{c}\vdash_{\text{var}} y \Downarrow rs \quad r = \text{bools-of } rs \\ \vdash_{\text{exp}} \boldsymbol{e} \Downarrow \boldsymbol{es} \quad r \vdash_{\text{reset}} f(\boldsymbol{es}) \Downarrow \boldsymbol{xs} \quad \vdash_{\text{var}} \boldsymbol{x} \Downarrow \boldsymbol{xs}\end{array}}{\vdash_{\text{eqn}} \boldsymbol{x} = (\texttt{restart}\ f\ \texttt{every}\ y)(\boldsymbol{e})}$$

$$\frac{}{r \vdash_{\text{reset}} f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

Node application

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

Modular reset

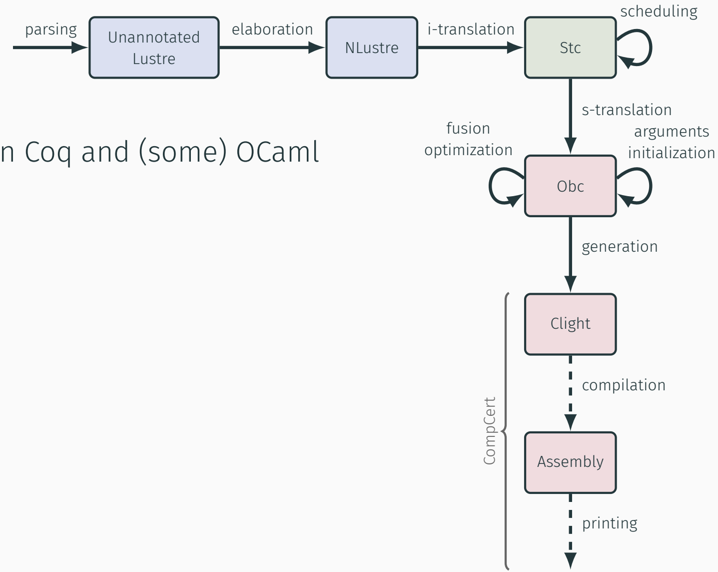
$$\frac{\begin{array}{c}\vdash_{\text{var}} y \Downarrow rs \quad r = \text{bools-of } rs \\ \vdash_{\text{exp}} \boldsymbol{e} \Downarrow \boldsymbol{es} \quad r \vdash_{\text{reset}} f(\boldsymbol{es}) \Downarrow \boldsymbol{xs} \quad \vdash_{\text{var}} \boldsymbol{x} \Downarrow \boldsymbol{xs}\end{array}}{\vdash_{\text{eqn}} \boldsymbol{x} = (\texttt{restart}\ f\ \texttt{every}\ y)(\boldsymbol{e})}$$

Use of an universally quantified relation as a constraint:

$$\frac{\forall k,\ \vdash_{\text{node}} f(\text{mask}_r^k\, \boldsymbol{xs}) \Downarrow \text{mask}_r^k\, \boldsymbol{ys}}{r \vdash_{\text{reset}} f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

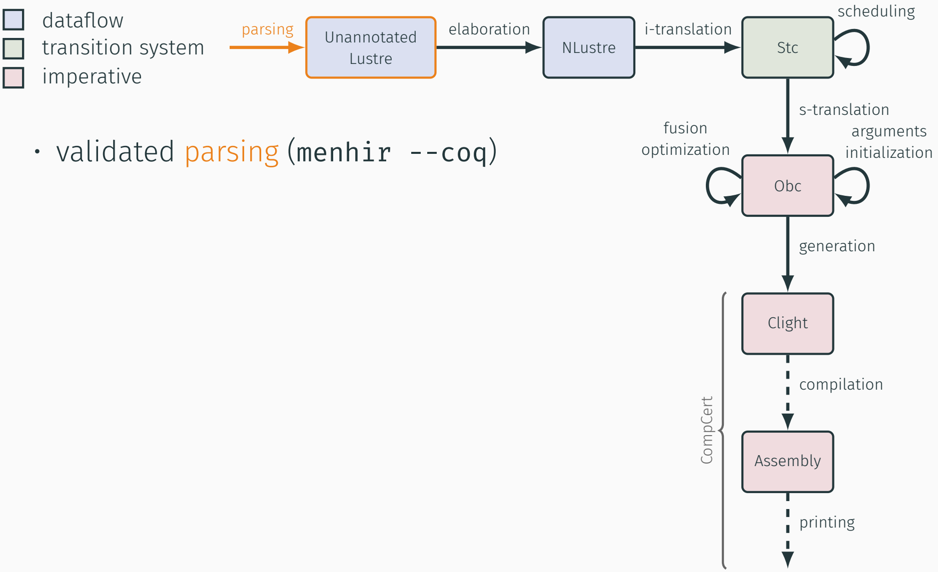
Vélus: a Verified Lustre Compiler

- dataflow
- transition system
- imperative



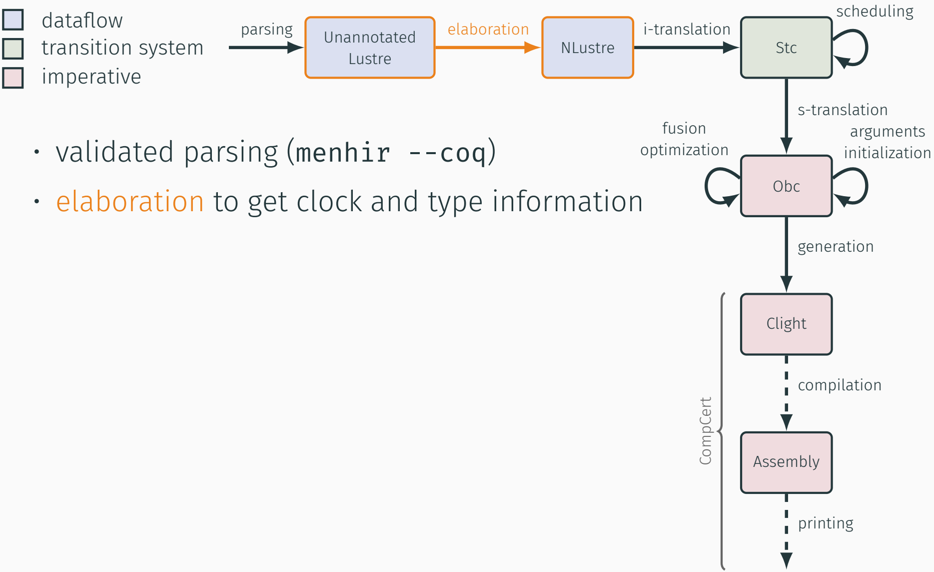
Implemented in Coq and (some) OCaml

VÉLUS: A VERIFIED LUSTRE COMPILER



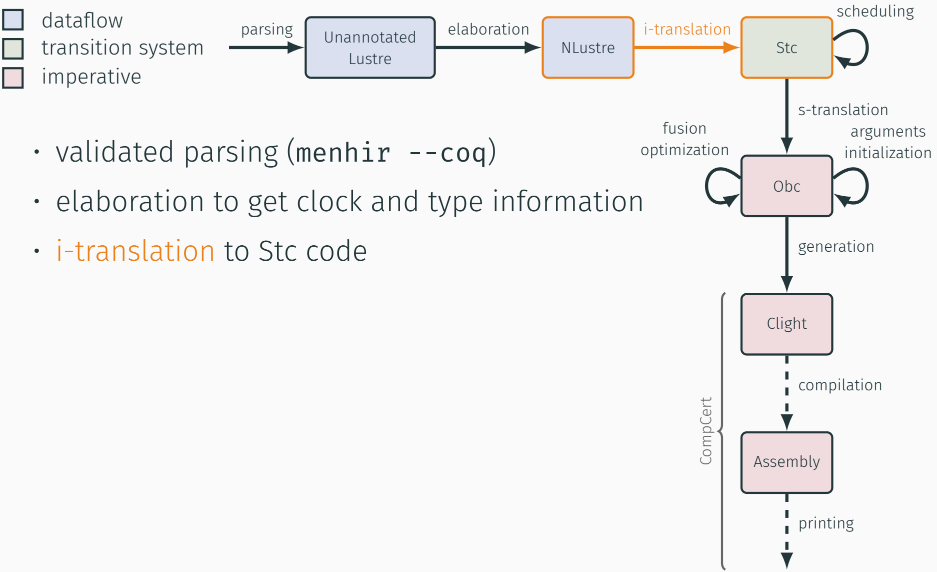
- validated parsing (`menhir --coq`)

Vélus: a Verified Lustre Compiler



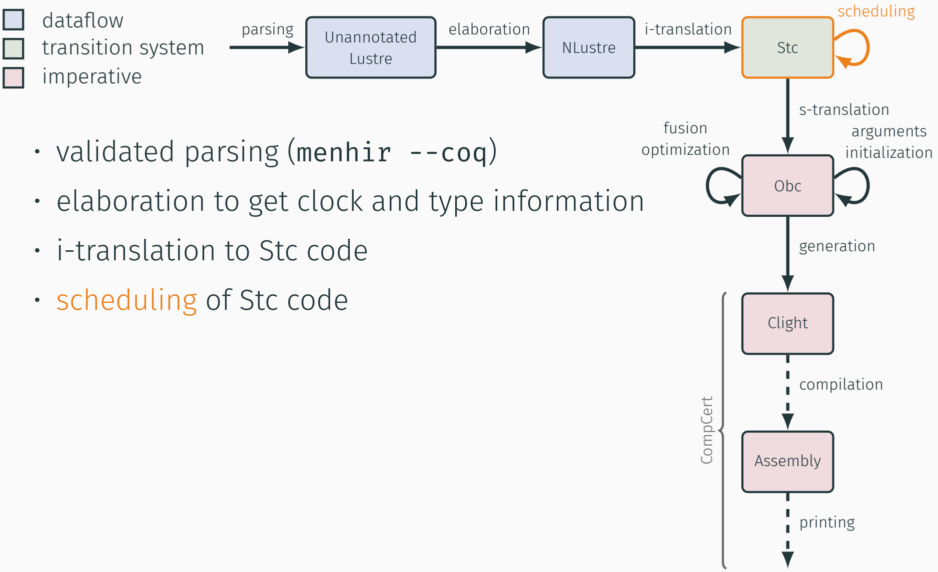
- validated parsing (`menhir --coq`)
- elaboration to get clock and type information

Vélus: a Verified Lustre Compiler



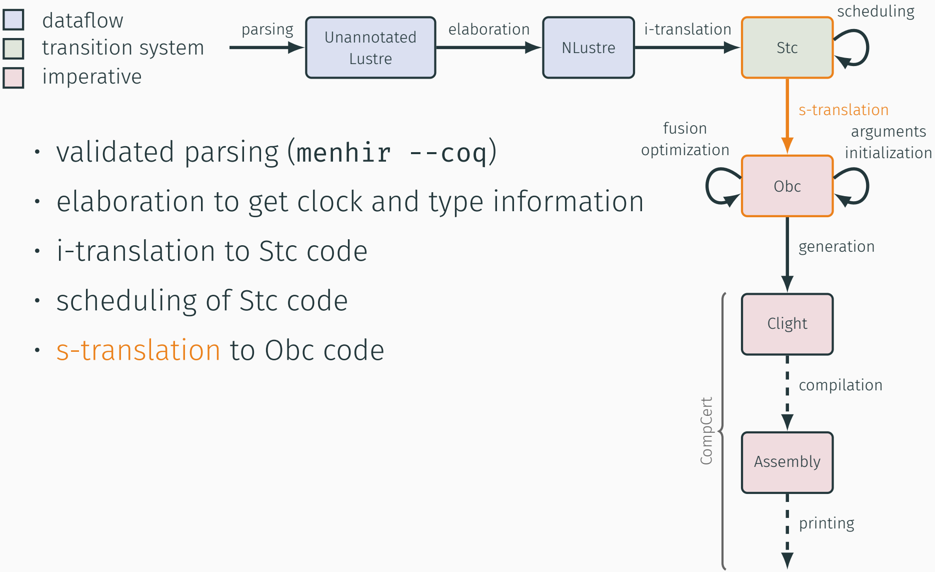
- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- i-translation to Stc code

Vélus: a Verified Lustre Compiler



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

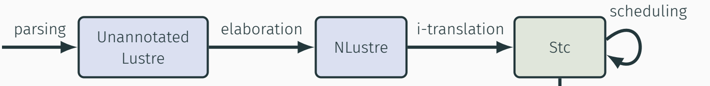
Vélus: a Verified Lustre Compiler

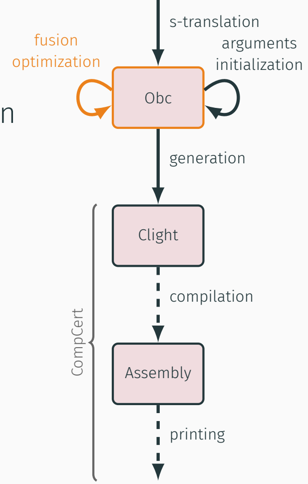


- validated parsing (`menhir --coq`)
- elaboration to get clock and type information
- i-translation to Stc code
- scheduling of Stc code
- s-translation to Obc code

Vélus: a Verified Lustre Compiler

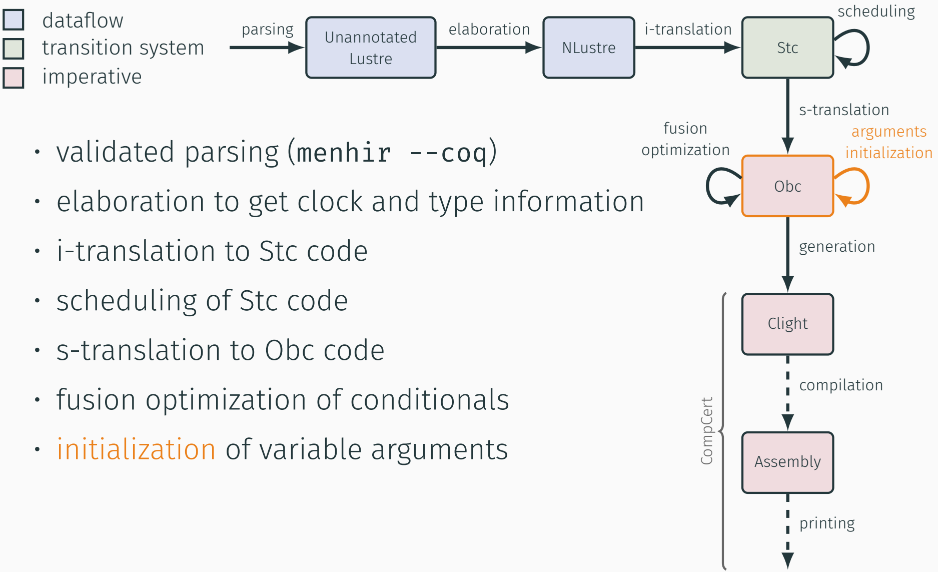
- dataflow
- transition system
- imperative





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

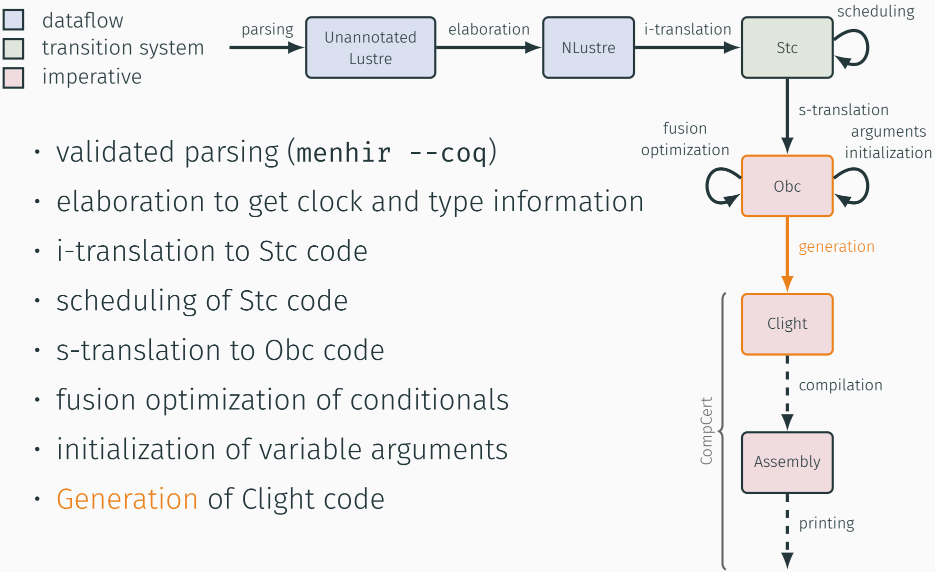
Vélus: a Verified Lustre Compiler



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

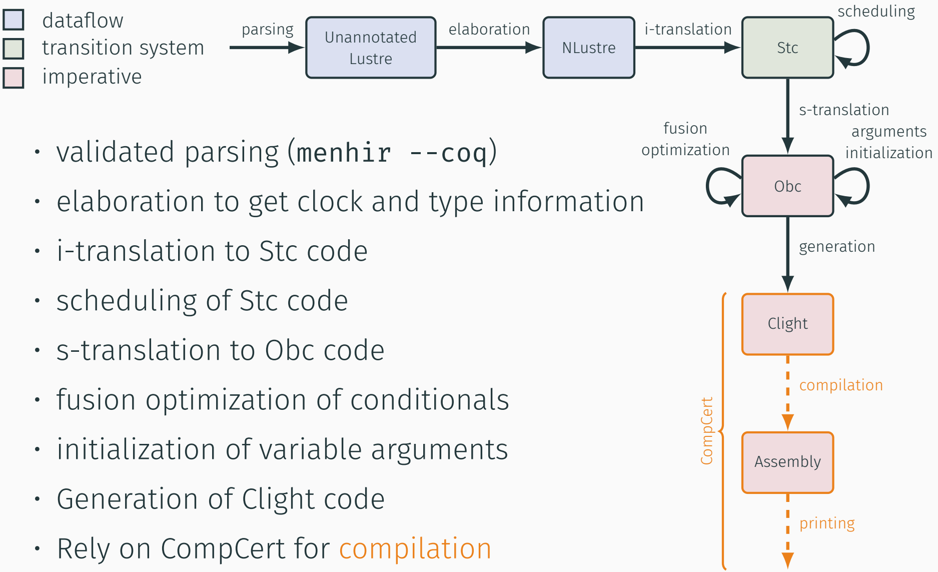
Vélus: a Verified Lustre Compiler

[BL09]



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

Why Stc?

Two issues:

Why Stc?

Two issues:

Syntactic granularity

reset: *schedulable* separate construct

Why Stc?

Two issues:

Syntactic granularity

reset: *schedulable* separate construct

Semantic granularity

transient states

First Issue: Naive Compilation

NLustre

```
x, a = (restart ins every r)(x0, u);
```

Obc

```
if ck_r {
  if r { ins(i).reset() }
};
x, a := ins(i).step(x0, u)
```

First Issue: Naive Compilation

NLustre

```
x, a = (restart ins every r)(x0, u);
```

Obc

```
if ck_r {
  if r { ins(i).reset() }
};
x, a := ins(i).step(x0, u)
```

Problem with fusion optimization:

```
x, ax = (restart ins every r)(x0, u);
y, ay = (restart ins every r)(y0, v);
```

```
if ck_r {
  if r { ins(i).reset() }
};
x, ax := ins(i).step(x0, u);
if ck_r {
  if r { ins(j).reset() }
};
y, ay := ins(j).step(y0, v)
```

NLustre

```
x, a = (restart ins every r)(x0, u);
```

Obc

```
if ck_r {
  if r { ins(i).reset() }
};
x, a := ins(i).step(x0, u)
```

Problem with fusion optimization:

```
x, ax = (restart ins every r)(x0, u);
y, ay = (restart ins every r)(y0, v);
```

```
if ck_r {
  if r {
    ins(i).reset();
    ins(j).reset()
  }
};
x, ax := ins(i).step(x0, u);
y, ay := ins(j).step(y0, v)
```

NLustre

```
x, a = (restart ins every r)(x0, u);
```

Obc

```
if ck_r {
  if r { ins(i).reset() }
};
x, a := ins(i).step(x0, u)
```

Problem with fusion optimization:

```
x, ax = (restart ins every r)(x0, u);
y, ay = (restart ins every r)(y0, v);
```

```
if ck_r {
  if r {
    ins(i).reset();
    ins(j).reset()
  }
};
x, ax := ins(i).step(x0, u);
y, ay := ins(j).step(y0, v)
```

→ Schedule node applications and resets separately

Second Issue: Proof of Correctness

NLustre

$V^\omega \times V^\omega$

Obc

$S \times \mathbf{V} \rightarrow S \times \mathbf{V}$

Second Issue: Proof of Correctness

NLustre

$V^\omega \times V^\omega$

too weak for induction

Obc

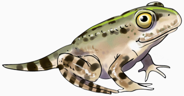
$S \times V \rightarrow S \times V$

Second Issue: Proof of Correctness

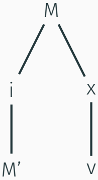
NLustre

$V^\omega \times V^\omega$

"easy": $\exists M$



$V^\omega \times M^\omega \times V^\omega$



Obc

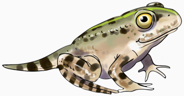
$S \times V \rightarrow S \times V$

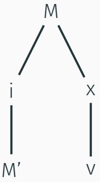
Second Issue: Proof of Correctness

NLustre

$V^{\omega} \times V^{\omega}$

"easy": $\exists M$





$V^{\omega} \times M^{\omega} \times V^{\omega}$

"hard"

Obc

$S \times V \to S \times V$

Second Issue: Proof of Correctness

What about the reset?

- Similar semantics in the memory model

What about the reset?

- Similar semantics in the memory model
- The “easy” proof can be done

Second Issue: Proof of Correctness

What about the reset?

- Similar semantics in the memory model
- The “easy” proof can be done
- The “hard” one failed

Stc: Synchronous Transition Code

Propose a new intermediate language

- Declarative, like NLustre

Stc: Synchronous Transition Code

Propose a new intermediate language

- Declarative, like NLustre
- Reset as a separate construct

Stc: Synchronous Transition Code

Propose a new intermediate language

- Declarative, like NLustre
- Reset as a separate construct
- Explicit state, as in the memory model of NLustre

Stc: Synchronous Transition Code

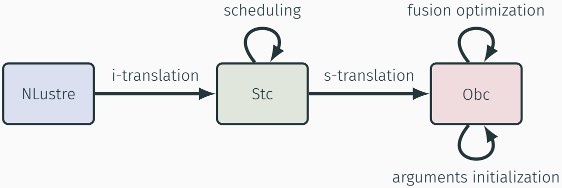
Propose a new intermediate language

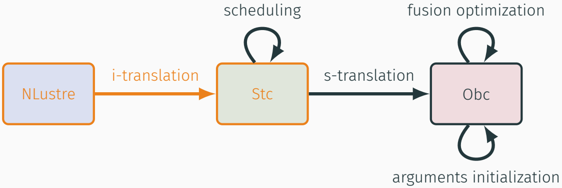
- Declarative, like NLustre
- Reset as a separate construct
- Explicit state, as in the memory model of NLustre
- Transient states

Stc: Synchronous Transition Code

Propose a new intermediate language

- Declarative, like NLustre
- Reset as a separate construct
- Explicit state, as in the memory model of NLustre
- Transient states




scheduling
fusion optimization
NLustre
i-translation
Stc
s-translation
Obc
arguments initialization

NLustre

```
node euler(x0, u: double)
  returns (x: double);
  var i: bool, px: double;
let
  i = true fby false;
  x = if i then x0 else px;
  px = 0.0 fby (x + 0.1 * u);
tel
```

Stc

```
system euler {
  init i = true, px = 0.;

  transition(x0, u: double)
    returns (x: double)
  {
    next i = false;
    x = if i then x0 else px;
    next px = x + 0.1 * u;
  }
}
```

NLustre

```
node euler(x0, u: double)
  returns (x: double);
  var i: bool, px: double;
let
  i = true fby false;
  x = if i then x0 else px;
  px = 0.0 fby (x + 0.1 * u);
tel
```

Stc

```
system euler {
  init i = true, px = 0.;

  transition(x0, u: double)
    returns (x: double)
  {
    next i = false;
    x = if i then x0 else px;
    next px = x + 0.1 * u;
  }
}
```

NLustre

```
node euler(x0, u: double)
  returns (x: double);
  var i: bool, px: double;
let
  i = true fby false;
  x = if i then x0 else px;
  px = 0.0 fby (x + 0.1 * u);
tel
```

Stc

```
system euler {
  init i = true, px = 0.;

  transition(x0, u: double)
    returns (x: double)
  {
    next i = false;
    x = if i then x0 else px;
    next px = x + 0.1 * u;
  }
}
```

```
node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var k: int, px: double,
      xe: double whenot alarm;
let
  k = 0 fby k + 1;
  alarm = (k ≥ 50);
  xe = euler(gps whenot alarm,
             xv whenot alarm);
  x = merge alarm (px when alarm) xe;
  px = 0. fby x;
tel
```

```
system ins {
  init k = 0, px = 0.;
  sub xe: euler;

  transition(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double whenot alarm;
  {
    next k = k + 1;
    alarm = (k ≥ 50);
    xe = euler<xe,0>(gps whenot alarm,
                     xv whenot alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}
```

NLustre

```
node ins(gps, xv: double)
  returns (x: double, alarm: bool)
  var k: int, px: double,
      xe: double whenot alarm;
let
  k = 0 fby k + 1;
  alarm = (k ≥ 50);
  xe = euler(gps whenot alarm,
             xv whenot alarm);
  x = merge alarm (px when alarm) xe;
  px = 0. fby x;
tel
```

Stc

```
system ins {
  init k = 0, px = 0.;
  sub xe: euler;

  transition(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double whenot alarm;
  {
    next k = k + 1;
    alarm = (k ≥ 50);
    xe = euler<xe,0>(gps whenot alarm,
                     xv whenot alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}
```

```
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,1>(x0, u);
    reset ins<x> every (. on r);
    y, ay = ins<y,1>(y0, v);
    reset ins<y> every (. on r);
  }
}
```

NLustre

```
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
```

Stc

```
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,1>(x0, u);
    reset ins<x> every (. on r);
    y, ay = ins<y,1>(y0, v);
    reset ins<y> every (. on r);
  }
}
```

NLustre

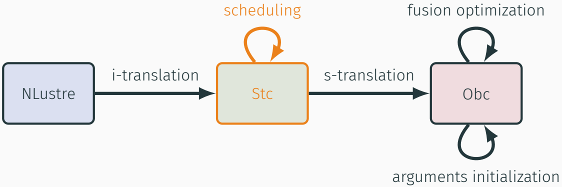
```
node driver(x0, y0, u, v: double,
            r: bool)
  returns (x, y: double)
  var ax, ay: bool;
let
  x, ax = (restart ins every r)(x0, u);
  y, ay = (restart ins every r)(y0, v);
tel
```

Stc

```
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,1>(x0, u);
    reset ins<x> every (. on r);
    y, ay = ins<y,1>(y0, v);
    reset ins<y> every (. on r);
  }
}
```

Scheduling


scheduling
fusion optimization
NLustre
i-translation
Stc
s-translation
Obc
arguments initialization

Scheduling

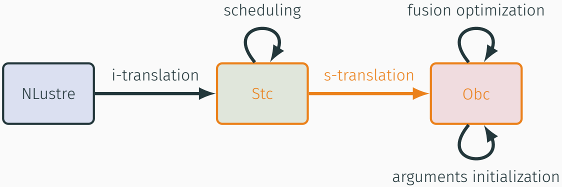
```
system driver {
  sub x: ins, y: ins;

  transition(x0, y0, u, v: double,
             r: bool)
    returns (x, y: double)
    var ax, ay: bool;
  {
    x, ax = ins<x,1>(x0, u);
    reset ins<x> every (. on r);
    y, ay = ins<y,1>(y0, v);
    reset ins<y> every (. on r);
  }
}
```

```
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,1>(x0, u);
    y, ay = ins<y,1>(y0, v);
  }
}
```

Stc → Obc


scheduling
fusion optimization
NLustre
i-translation
Stc
s-translation
Obc
arguments initialization

Stc

```
system euler {
  init i = true, px = 0.;

  transition(x0, u: double)
    returns (x: double)
  {
    x = if i then x0 else px;
    next i = false;
    next px = x + 0.1 * u;
  }
}
```

Obc

```
class euler {
  state i: bool, px: double;

  step(x0, u: double)
    returns (x: double)
  {
    if state(i) { x := x0 }
    else { x := state(px) };
    state(i) := false;
    state(px) := x + 0.1 * u
  }

  reset() { state(i) := true;
            state(px) := 0. }
}
```

Stc

Obc

```
system euler {
  init i = true, px = 0.;

  transition(x0, u: double)
    returns (x: double)
  {
    x = if i then x0 else px;
    next i = false;
    next px = x + 0.1 * u;
  }
}
```

```
class euler {
  state i: bool, px: double;

  step(x0, u: double)
    returns (x: double)
  {
    if state(i) { x := x0 }
    else { x := state(px) };
    state(i) := false;
    state(px) := x + 0.1 * u
  }

  reset() { state(i) := true;
            state(px) := 0. }
}
```

Stc

```
system euler {
  init i = true, px = 0.;

  transition(x0, u: double)
    returns (x: double)
  {
    x = if i then x0 else px;
    next i = false;
    next px = x + 0.1 * u;
  }
}
```

Obc

```
class euler {
  state i: bool, px: double;

  step(x0, u: double)
    returns (x: double)
  {
    if state(i) { x := x0 }
    else { x := state(px) };
    state(i) := false;
    state(px) := x + 0.1 * u
  }

  reset() { state(i) := true;
            state(px) := 0. }
}
```

Stc

```
system ins {
  init k = 0, px = 0.;
  sub xe: euler;

  transition(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double whenot alarm;
  {
    alarm = (k ≥ 50);
    next k = k + 1;
    xe = euler<xe,0>(gps whenot alarm,
                     xv whenot alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}
```

Obc

```
class ins {
  state k: int, px: double;
  instance xe: euler;

  step(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double
  {
    alarm := state(k) ≥ 50;
    state(k) := state(k) + 1;
    if alarm { }
    else { xe := euler(xe).step(gps, xv) };
    if alarm { x := state(px) }
    else { x := xe };
    state(px) := x
  }

  reset() { state(k) := 0;
            state(px) := 0.;
            euler(xe).reset() }
}
```

Stc

```
system ins {
  init k = 0, px = 0.;
  sub xe: euler;

  transition(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double whenot alarm;
  {
    alarm = (k ≥ 50);
    next k = k + 1;
    xe = euler<xe,0>(gps whenot alarm,
                     xv whenot alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}
```

Obc

```
class ins {
  state k: int, px: double;
  instance xe: euler;

  step(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double
  {
    alarm := state(k) ≥ 50;
    state(k) := state(k) + 1;
    if alarm { }
    else { xe := euler(xe).step(gps, xv) };
    if alarm { x := state(px) }
    else { x := xe };
    state(px) := x
  }

  reset() { state(k) := 0;
            state(px) := 0.;
            euler(xe).reset() }
}
```

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,1>(x0, u);
    y, ay = ins<y,1>(y0, v);
  }
}
```

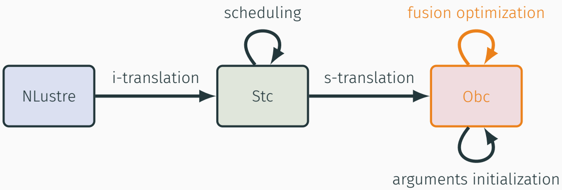
Obc

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

  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)
  }

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

Fusion Optimization



Fusion Optimization

```
class ins {
  state k: int, px: double;
  instance xe: euler;

  step(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double
  {
    alarm := state(k) ≥ 50;
    state(k) := state(k) + 1;
    if alarm { }
    else { xe := euler(xe).step(gps, xv) };
    if alarm { x := state(px) }
    else { x := xe };
    state(px) := x
  }

  reset() { state(k) := 0;
            state(px) := 0.;
            euler(xe).reset() }
}
```

```
class ins {
  state k: int, px: double;
  instance xe: euler;

  step(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double
  {
    alarm := state(k) ≥ 50;
    state(k) := state(k) + 1;
    if alarm { x := state(px) }
    else {
      xe := euler(xe).step(gps, xv);
      x := xe
    };
    state(px) := x
  }

  reset() { state(k) := 0;
            state(px) := 0.;
            euler(xe).reset() }
}
```

Fusion Optimization

```
class ins {
  state k: int, px: double;
  instance xe: euler;

  step(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double
  {
    alarm := state(k) ≥ 50;
    state(k) := state(k) + 1;
    if alarm { }
    else { xe := euler(xe).step(gps, xv) };
    if alarm { x := state(px) }
    else { x := xe };
    state(px) := x
  }

  reset() { state(k) := 0;
            state(px) := 0.;
            euler(xe).reset() }
}
```

```
class ins {
  state k: int, px: double;
  instance xe: euler;

  step(gps, xv: double)
    returns (x: double, alarm: bool)
    var xe: double
  {
    alarm := state(k) ≥ 50;
    state(k) := state(k) + 1;
    if alarm { x := state(px) }
    else {
      xe := euler(xe).step(gps, xv);
      x := xe
    };
    state(px) := x
  }

  reset() { state(k) := 0;
            state(px) := 0.;
            euler(xe).reset() }
}
```

Fusion Optimization

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

  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)
  }

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

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

  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)
  }

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

Fusion Optimization

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

  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)
  }

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

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

  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)
  }

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

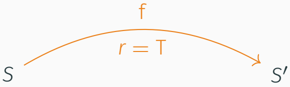
Stc Formal Semantics: Intuition

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```

Transition system

- Three states S, I and S'
- Transition constraints

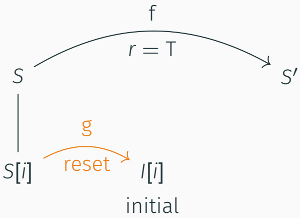
```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```



Transition system

- Three states S, I and S'
- Transition constraints

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```

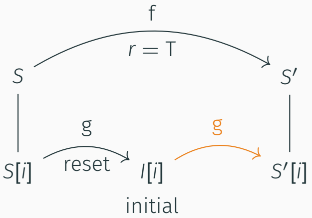


Transition system

- Three states S, I and S'
- Transition constraints

Stc Formal Semantics: Intuition

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```



Transition system

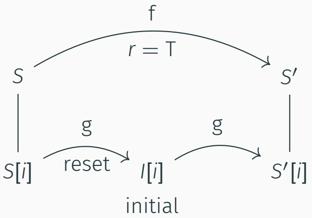
- Three states S, I and S'
- Transition constraints

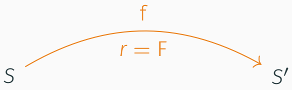
Stc Formal Semantics: Intuition

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```

Transition system

- Three states S, I and S'
- Transition constraints



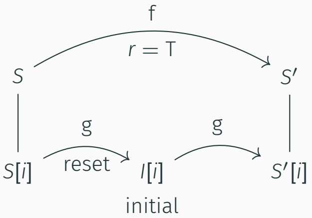


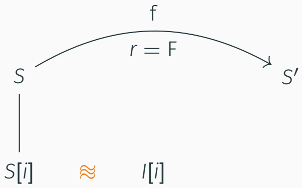
Stc Formal Semantics: Intuition

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```

Transition system

- Three states S, I and S'
- Transition constraints



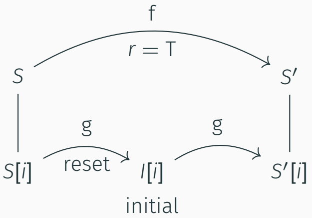


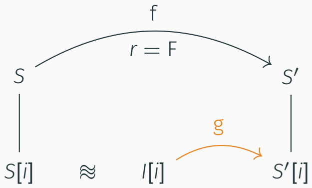
Stc Formal Semantics: Intuition

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    reset g<i> every (. on r);
    y = g<i,1>(x);
  }
}
```

Transition system

- Three states S, I and S'
- Transition constraints



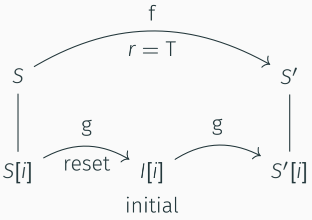


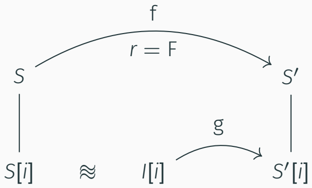
Stc Formal Semantics: Intuition

```
system f {
  sub i: g;
  transition (x: int, r: bool)
    returns (y: int)
  {
    y = g<i,1>(x);
    reset g<i> every (. on r);
  }
}
```

Transition system

- Three states S, I and S'
- Transition constraints
- Declarative





Stc Formal Semantics: Simplified Rules

Basic transition constraint

$$\frac{R, b \vdash e \downarrow R(x)}{P, R, b, S, I, S' \vdash x = e}$$

Basic transition constraint

$$\frac{R, b \vdash e \downarrow R(x)}{P, R, b, S, I, S' \vdash x = e}$$

Next transition constraint

$$\frac{R, b \vdash e \downarrow \langle v \rangle \quad R(x) = \langle S(x) \rangle \quad S'(x) = v}{P, R, b, S, I, S' \vdash \texttt{next}\ x = e}$$

Stc Formal Semantics: Simplified Rules

Basic transition constraint

$$\frac{R, b \vdash e \downarrow R(x)}{P, R, b, S, I, S' \vdash x = e}$$

Next transition constraint

$$\frac{R, b \vdash e \downarrow \langle v \rangle \quad R(x) = \langle S(x) \rangle \quad S'(x) = v}{P, R, b, S, I, S' \vdash \texttt{next}\ x = e}$$

$$\frac{R, b \vdash e \downarrow \langle\,\rangle \quad R(x) = \langle\,\rangle \quad S'(x) = S(x)}{P, R, b, S, I, S' \vdash \texttt{next}\ x = e}$$

Default transition

$$\frac{R, b \vdash \boldsymbol{e} \downarrow \boldsymbol{v} \quad P, I[i]\,, S'[i] \vdash f\,(\boldsymbol{v}) \Downarrow R(\boldsymbol{x}) \qquad \text{if } \ (k = 0) \ \text{ then } I[i] \approx S[i]}{P, R, b, S, I, S' \vdash \boldsymbol{x} \texttt{ = } f\texttt{<}i, k\texttt{>(}\,\boldsymbol{e}\,\texttt{)}}$$

Default transition

$$\frac{R, b \vdash \boldsymbol{e} \downarrow \boldsymbol{v} \quad P, I[i]\,, S'[i] \vdash f\,(\boldsymbol{v}) \Downarrow R(\boldsymbol{x}) \quad \text{if } (k = 0) \text{ then } I[i] \approx S[i]}{P, R, b, S, I, S' \vdash \boldsymbol{x} = f\texttt{<}i, k\texttt{>}(\,\boldsymbol{e}\,)}$$

Reset transition

$$\frac{R, b \vdash ck \downarrow \text{true} \quad \text{initial-state}\, P\; f\; I[i]}{P, R, b, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

Default transition

$$\frac{R, b \vdash \boldsymbol{e} \downarrow \boldsymbol{v} \quad P, I[i], S'[i] \vdash f(\boldsymbol{v}) \Downarrow R(\boldsymbol{x}) \quad \text{if } (k = 0) \text{ then } I[i] \approx S[i]}{P, R, b, S, I, S' \vdash \boldsymbol{x} = f\texttt{<}i, k\texttt{>}(\boldsymbol{e})}$$

Reset transition

$$\frac{R, b \vdash ck \downarrow \text{true} \quad \text{initial-state}\, P\, f\, I[i]}{P, R, b, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

$$\frac{R, b \vdash ck \downarrow \text{false} \quad I[i] \approx S[i]}{P, R, b, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

System

$$\frac{\text{system}(P, f) = s \quad R(s.\mathsf{in}) = \boldsymbol{xs} \quad R(s.\mathsf{out}) = \boldsymbol{ys} \qquad \forall tc \in s.\mathsf{tcs},\ P, R, \text{base-of}\ \boldsymbol{xs}, S, I, S' \vdash tc}{P, S, S' \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

System

$$\frac{\text{system}(P, f) = s \quad R(s.\textbf{in}) = \boldsymbol{xs} \quad R(s.\textbf{out}) = \boldsymbol{ys} \qquad \forall tc \in s.\textbf{tcs},\ P, R, \text{base-of}\ \boldsymbol{xs}, S, I, S' \vdash tc}{P, S, S' \vdash f(\boldsymbol{xs}) \downdownarrows \boldsymbol{ys}}$$

Loop

$$\frac{P, S, S' \vdash f(xs_n) \downdownarrows ys_n \quad P, S' \vdash f(xs) \overset{n+1}{\circlearrowleft} ys}{P, S \vdash f(xs) \overset{n}{\circlearrowleft} ys}$$

Theorem (i-translation correctness)
Given a program G, a name f, streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f(xs) \Downarrow ys$*, then*

$$\text{initial-state}\,(\text{i-tr}\,G)\;\; f\, M_0 \quad \textit{and} \quad \text{i-tr}\,G, M_0 \vdash f\,(xs) \overset{0}{\circlearrowleft} ys$$

Theorem (i-translation correctness)
Given a program G, a name f, streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f(xs) \Downarrow ys$*, then*

$$\text{initial-state}\,(\text{i-tr}\,G)\;\; f\, M_0 \quad \textit{and} \quad \text{i-tr}\,G, M_0 \vdash f\,(xs) \overset{0}{\circlearrowleft} ys$$

Proof.

1. **equation/transition constraint:** by cases, showing the existence of I

□

Theorem (i-translation correctness)
Given a program G, a name f, streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f(xs) \Downarrow ys$, *then*

$$\text{initial-state}\,(\text{i-tr}\, G)\ \ f\, M_0 \quad \textit{and} \quad \text{i-tr}\, G, M_0 \vdash f(xs) \overset{0}{\circlearrowleft} ys$$

Proof.

1. **equation/transition constraint:** by cases, showing the existence of I
2. **node/system:** induction on the equations, then on G

□

Theorem (i-translation correctness)
Given a program G, a name f, streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f(xs) \Downarrow ys$*, then*

$$\text{initial-state}\,(\text{i-tr}\,G)\;\; f\, M_0 \quad \textit{and} \quad \text{i-tr}\,G, M_0 \vdash f\,(xs) \overset{0}{\circlearrowleft} ys$$

Proof.

1. **equation/transition constraint:** by cases, showing the existence of I
2. **node/system:** induction on the equations, then on G
3. **loop:** coinduction

□

Theorem (s-translation correctness)
Given a program P, a name f, streams of lists of values xs and ys, a state S such that initial-state $P\ f\ S$ *and* $P, S \vdash f(xs) \overset{0}{\circlearrowleft} ys$*, then there exists a memory tree* $me \approx S$ *such that:*

$$\textsf{s-tr}\, P, \{\varnothing\} \vdash f.reset\,() \Downarrow me \quad \textit{and} \quad \textsf{s-tr}\, P, me \vdash f.step\,(xs) \overset{0}{\circlearrowleft} ys$$

Theorem (s-translation correctness)
Given a program P, a name f, streams of lists of values xs and ys, a state S such that initial-state $P\ f\ S$ *and* $P, S \vdash f(xs) \overset{0}{\circlearrowleft} ys$, *then there exists a memory tree* $me \approx S$ *such that:*

$$\textsf{s-tr}\, P, \{\varnothing\} \vdash f.reset\,() \Downarrow me \quad \textit{and} \quad \textsf{s-tr}\, P, me \vdash f.step\,(xs) \overset{0}{\circlearrowleft} ys$$

Proof.

1. **expressions:** by induction

□

Theorem (s-translation correctness)
Given a program P, a name f, streams of lists of values xs and ys, a state S such that initial-state $P\ f\ S$ *and* $P, S \vdash f(xs) \overset{0}{\circlearrowleft} ys$*, then there exists a memory tree* $me \approx S$ *such that:*

$$\textsf{s-tr}\, P, \{\varnothing\} \vdash f.reset\,() \Downarrow me \quad \textit{and} \quad \textsf{s-tr}\, P, me \vdash f.step\,(xs) \overset{0}{\circlearrowleft} ys$$

Proof.

1. **expressions:** by induction
2. **transition constraint/statement:** by cases, using correspondence relations

□

From Stc to Obc

Theorem (s-translation correctness)
Given a program P, a name f, streams of lists of values xs and ys, a state S such that $\text{initial-state}\, P\, f\, S$ *and* $P, S \vdash f\,(xs) \overset{0}{\circlearrowleft} ys$*, then there exists a memory tree* $me \approx S$ *such that:*

$$\text{s-tr}\, P, \{\varnothing\} \vdash f.reset\,() \Downarrow me \quad \text{and} \quad \text{s-tr}\, P, me \vdash f.step\,(xs) \overset{0}{\circlearrowleft\!\!\circlearrowleft} ys$$

Proof.

1. **expressions:** by induction
2. **transition constraint/statement:** by cases, using correspondence relations
3. **system/step method:** induction on the transition constraints, then on P

□

Theorem (s-translation correctness)
Given a program P, a name f, streams of lists of values xs and ys, a state S such that initial-state $P\ f\ S$ *and* $P, S \vdash f(xs) \overset{0}{\circlearrowleft} ys$*, then there exists a memory tree* $me \approx S$ *such that:*

$$\textsf{s-tr}\ P, \{\varnothing\} \vdash f.reset\,() \Downarrow me \quad and \quad \textsf{s-tr}\ P, me \vdash f.step\,(xs) \overset{0}{\circledcirc} ys$$

Proof.

1. **expressions:** by induction
2. **transition constraint/statement:** by cases, using correspondence relations
3. **system/step method:** induction on the transition constraints, then on *P*
4. **loop:** coinduction

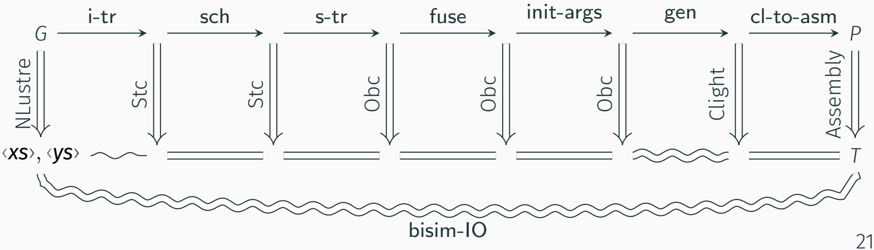
□

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* $G \vdash f\,(\langle \mathbf{xs} \rangle) \Downarrow \langle \mathbf{ys} \rangle$*, 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* $G \vdash f(\langle \mathbf{xs} \rangle) \Downarrow \langle \mathbf{ys} \rangle$*, then there exists an infinite trace of events T such that*

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



Summary

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

Summary

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

Future Work

- Normalization
- Extend Stc
- State machines

[Cas+87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Alexander Plaice. "LUSTRE: A Declarative Language for Programming Synchronous Systems". In: *In 14th Symposium on Principles of Programming Languages (POPL'87). ACM*. 1987.

[Cas94] Paul Caspi. "Towards Recursive Block Diagrams". In: *Annual Review in Automatic Programming* 18 (Jan. 1, 1994), pp. 81–85.

[CP97] Paul Caspi and Marc Pouzet. *A Co-Iterative Characterization of Synchronous Stream Functions*. VERIMAG, Oct. 1997.

[HP00] Grégoire Hamon and Marc Pouzet. "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, 2000, pp. 289–300.

References II

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “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, 2005, pp. 173–182.

[BL09] Sandrine Blazy and Xavier Leroy. “Mechanized Semantics for the Clight Subset of the C Language”. In: *Journal of Automated Reasoning* 43.3 (Oct. 1, 2009), pp. 263–288.

[Ler09] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115.

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. “Validating LR(1) Parsers”. In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 397–416.

[Bou+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. "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, 2017, pp. 586–601.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "SCADE 6: A Formal Language for Embedded Critical Software Development (Invited Paper)". In: *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE). Sept. 2017, pp. 1–11.