

Verified Compilation of a Synchronous Dataflow Language with State Machines

Basile Pesin

Inria Paris

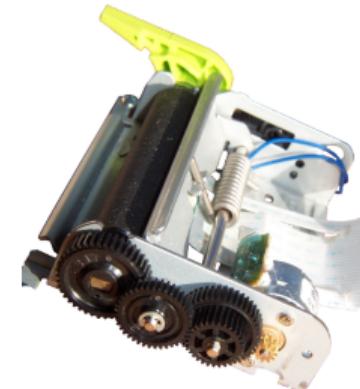
École normale supérieure, CNRS, PSL University

Friday, October 13

Programming embedded systems



cc Cjp24



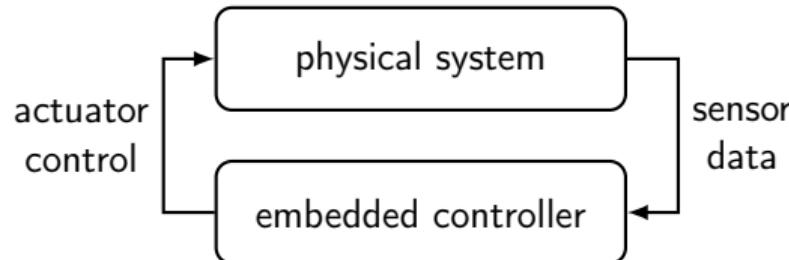
cc Akiry



Programming embedded systems



cc Cjp24



cc Akiry



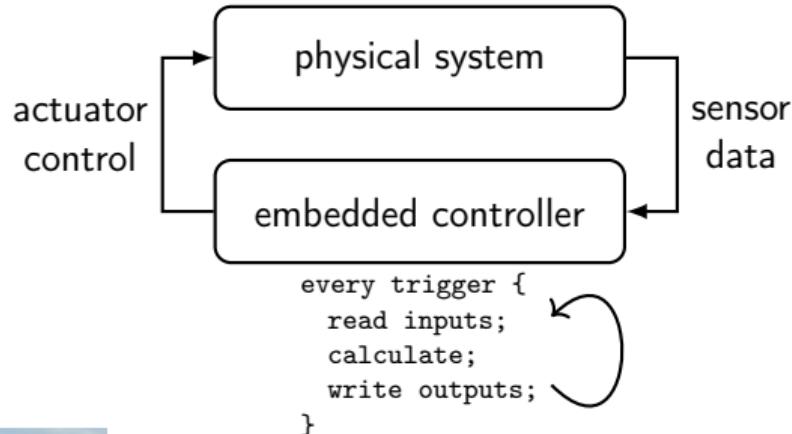
Programming embedded systems



cc Cjp24



cc Akiry



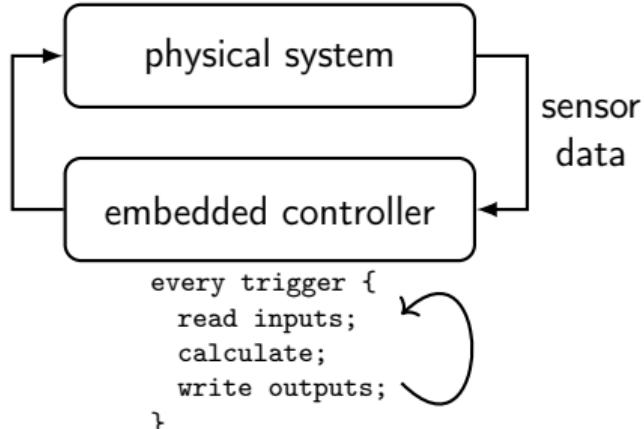
Programming embedded systems



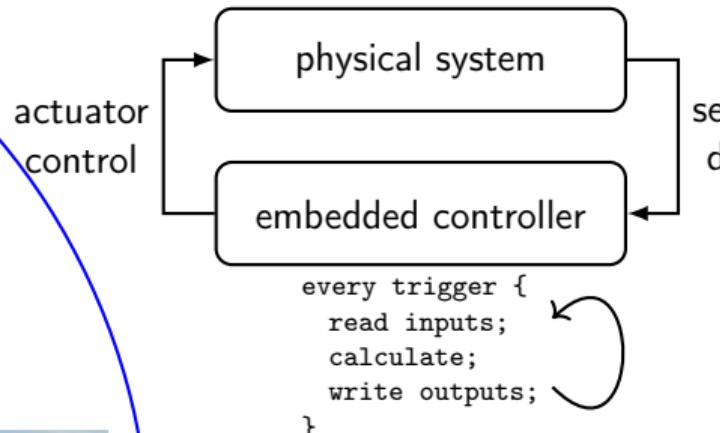
safety-critical



actuator
control

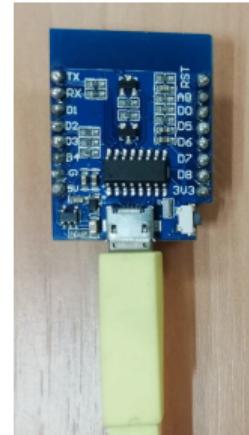


Programming embedded systems



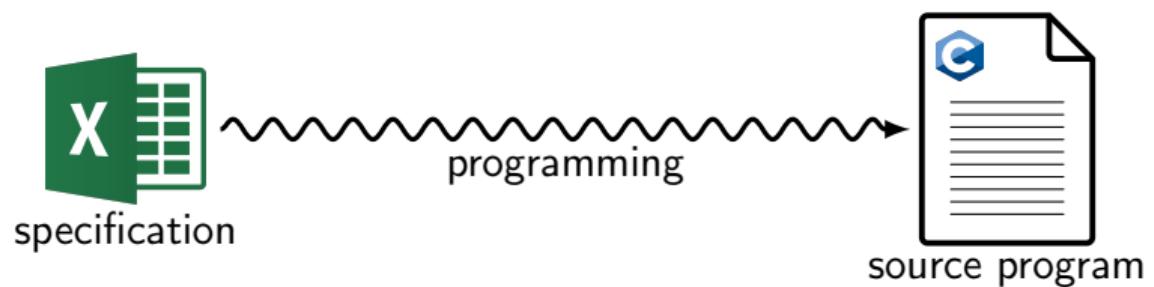
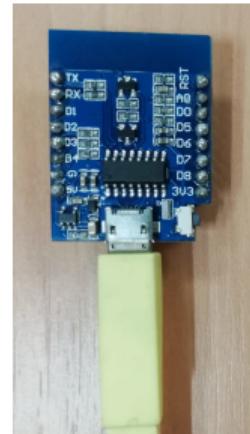
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system



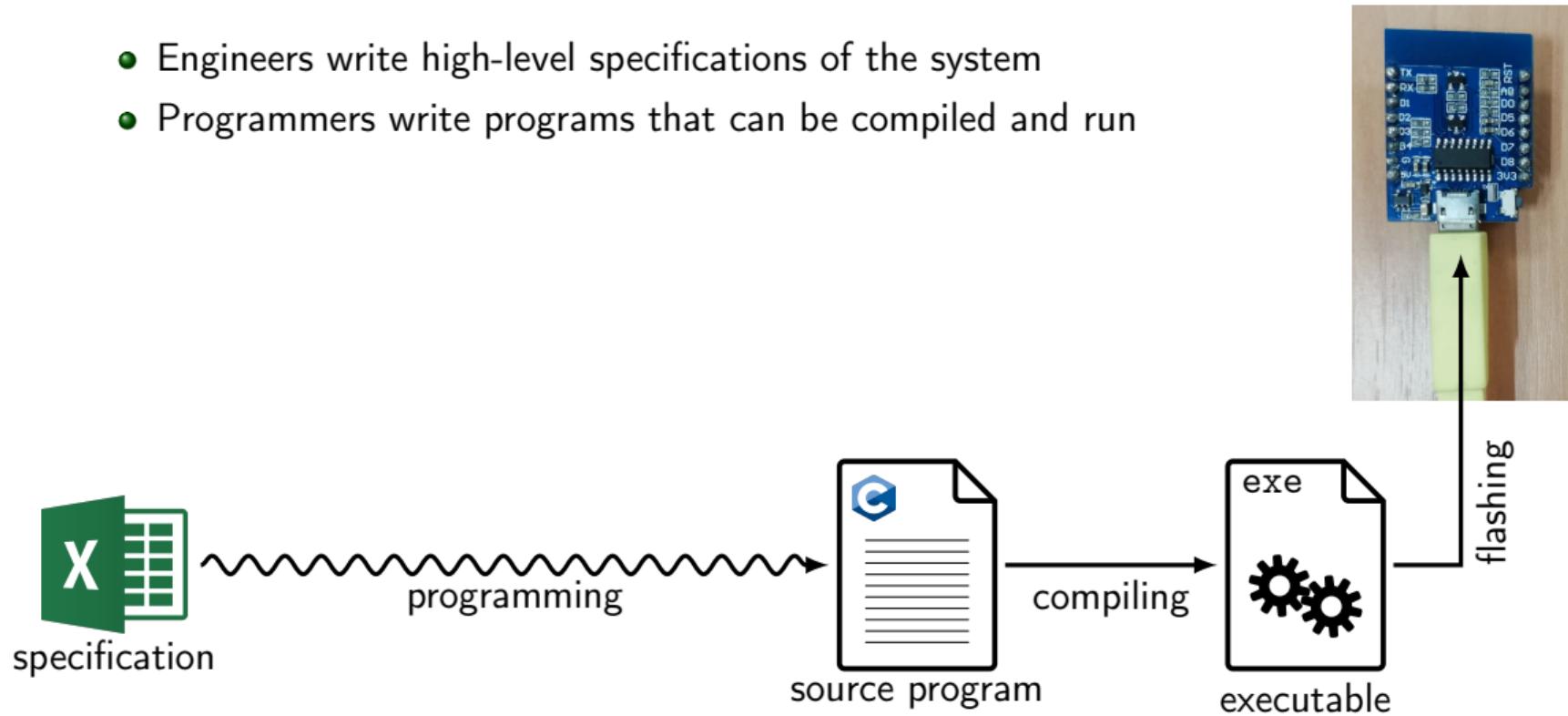
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run



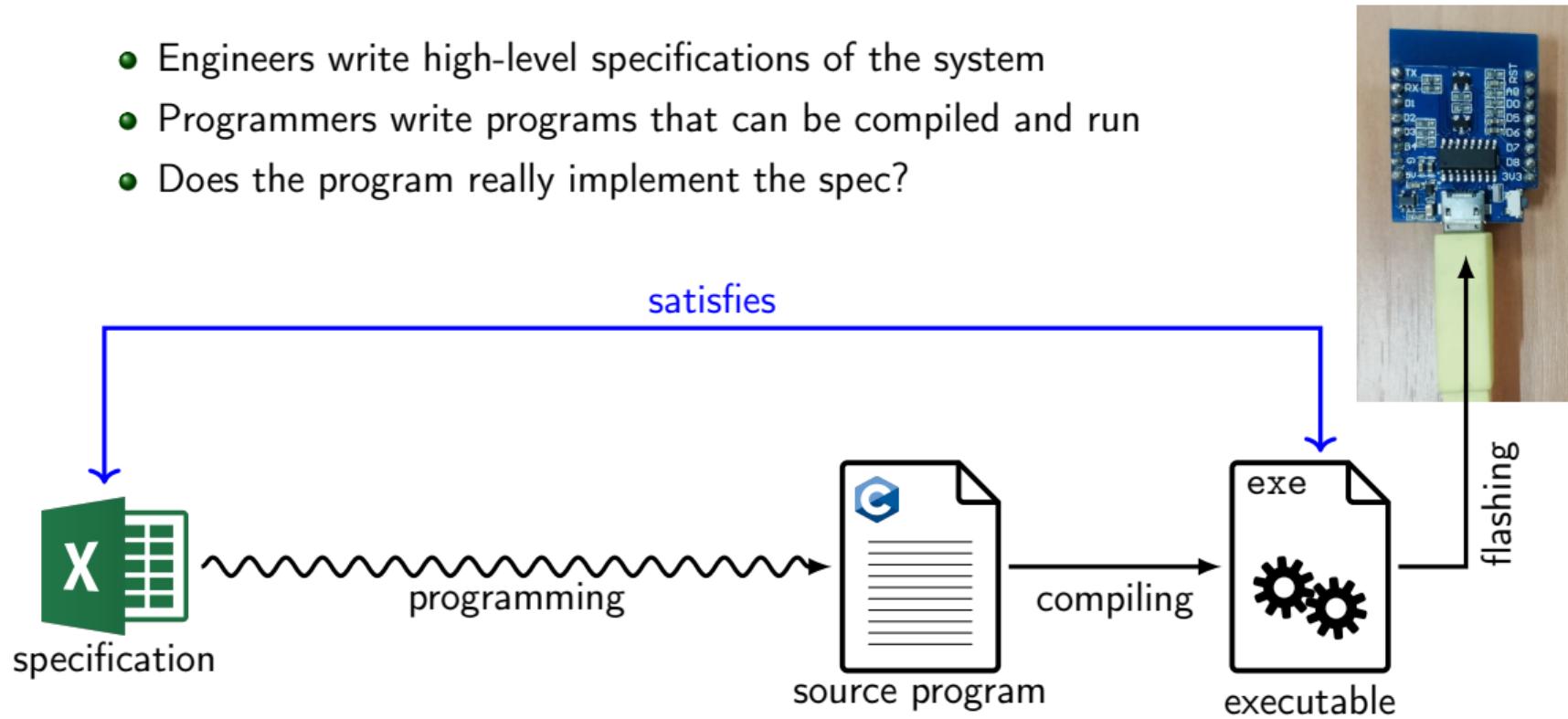
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run



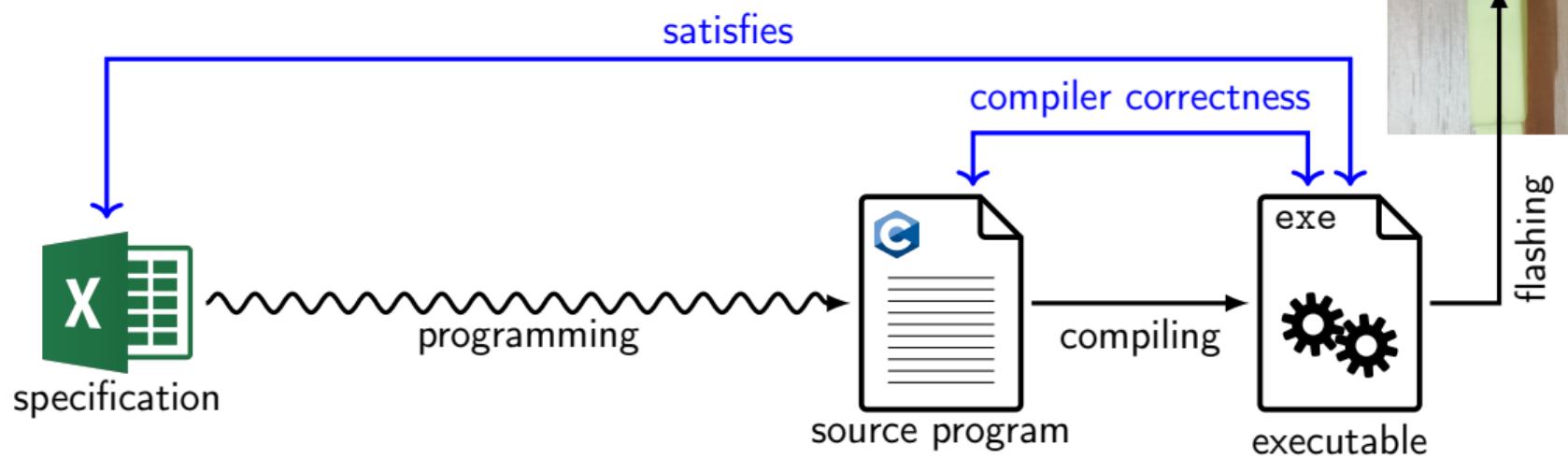
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?



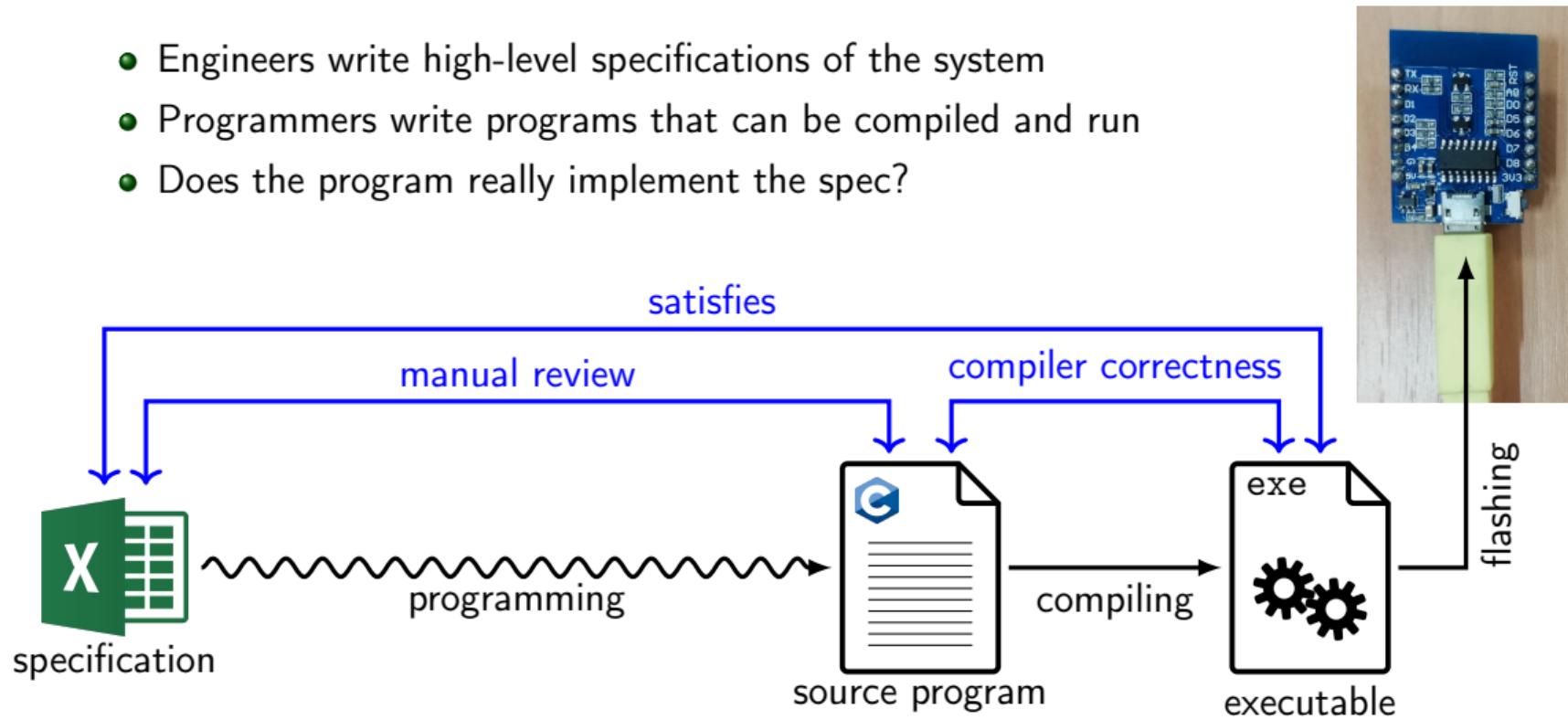
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?



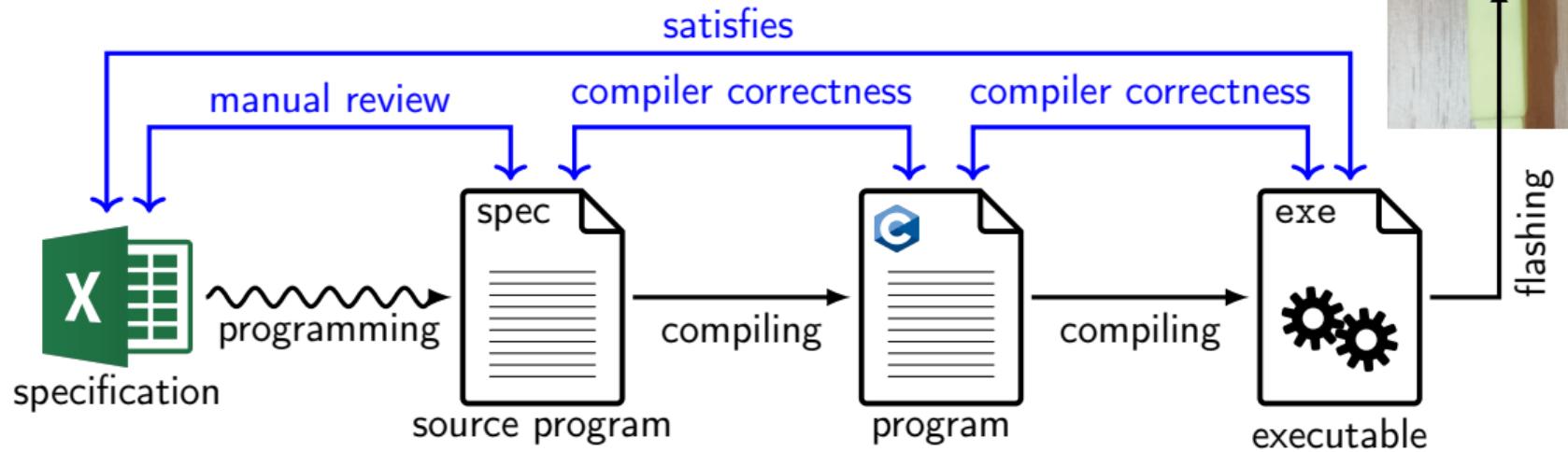
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?



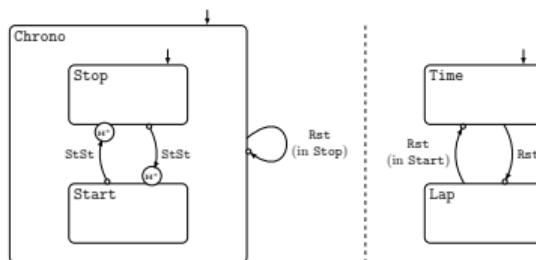
Low-level languages and high-level specifications

- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?
- Reduce the gap by programming in a language closer to the spec



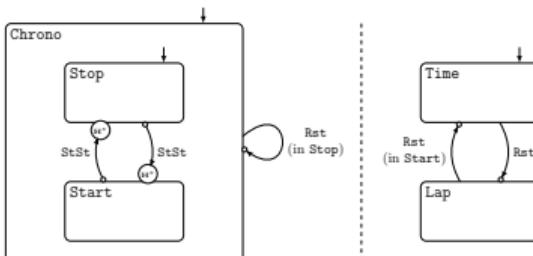
Programming Embedded Systems with State Machines

- **Statecharts** [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]



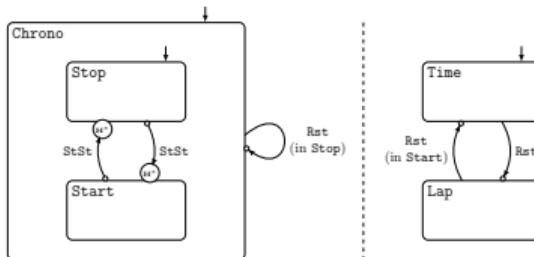
Programming Embedded Systems with State Machines

- **Statecharts** [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- **SyncCharts** [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- **Mode-Automata** [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]



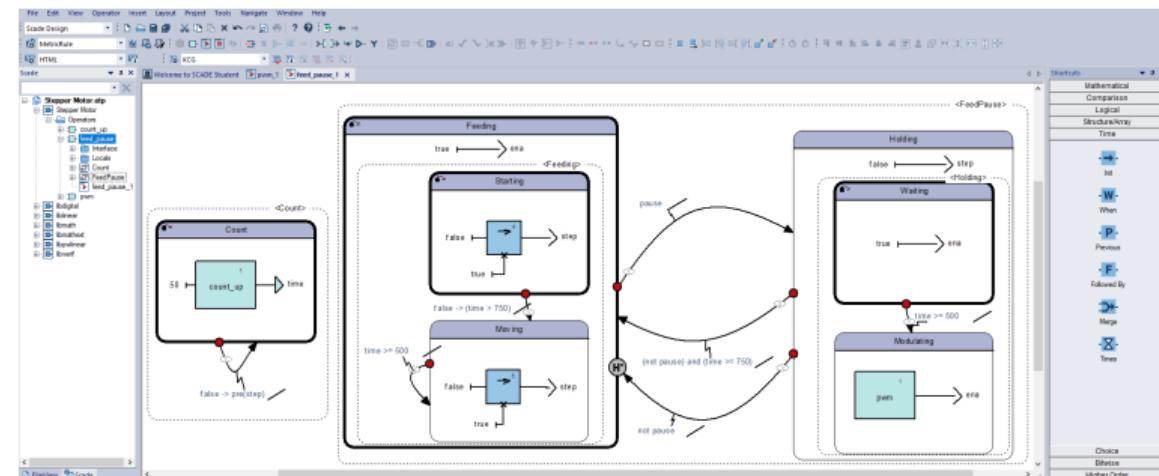
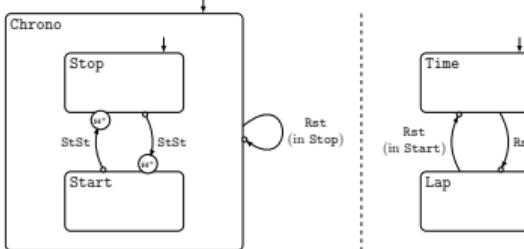
Programming Embedded Systems with State Machines

- **Statecharts** [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- **SyncCharts** [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- **Mode-Automata** [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]
- **Lucid Synchrone** [Pouzet (2006): Lucid Synchrone, v. 3.]
[Tutorial and reference manual]



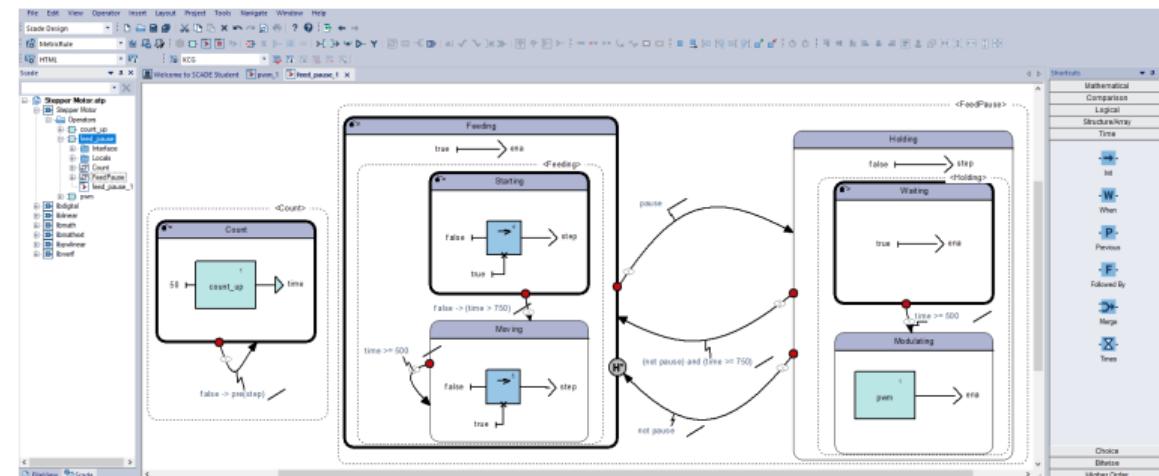
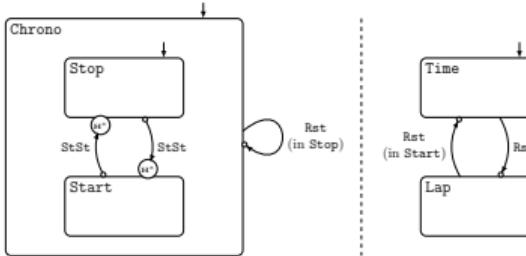
Programming Embedded Systems with State Machines

- **Statecharts** [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- **SyncCharts** [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- **Mode-Automata** [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]
- **Lucid Synchrone** [Pouzet (2006): Lucid Synchrone, v. 3.]
[Tutorial and reference manual]
- **Scade 6** [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]



Programming Embedded Systems with State Machines

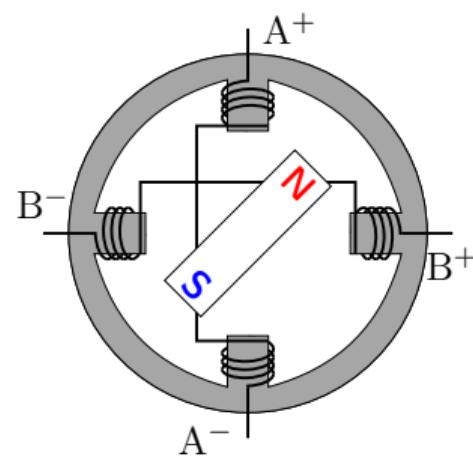
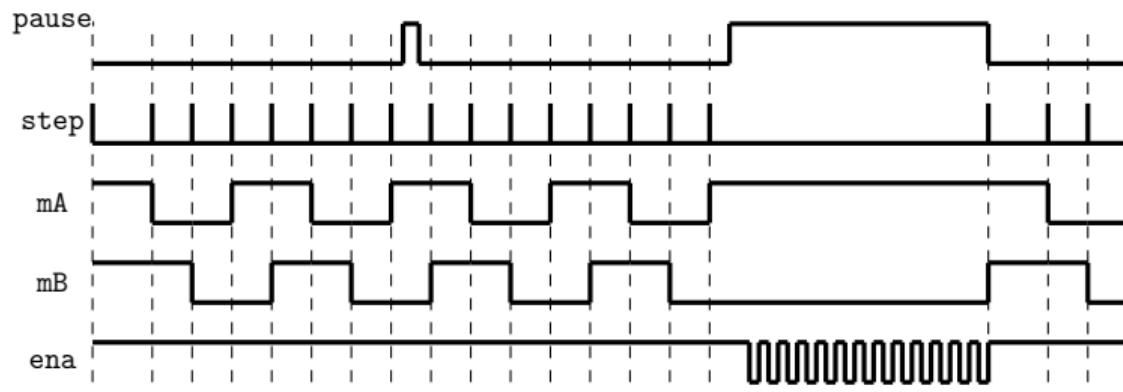
- **Statecharts** [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- **SyncCharts** [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- **Mode-Automata** [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]
- **Lucid Synchrone** [Pouzet (2006): Lucid Synchrone, v. 3.]
[Tutorial and reference manual]
- **Scade 6** [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
- **Vélus: A subset of Scade 6**



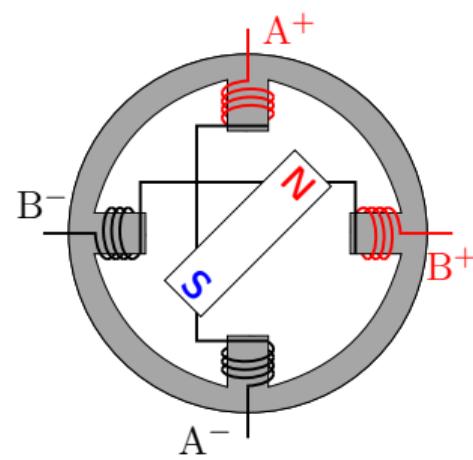
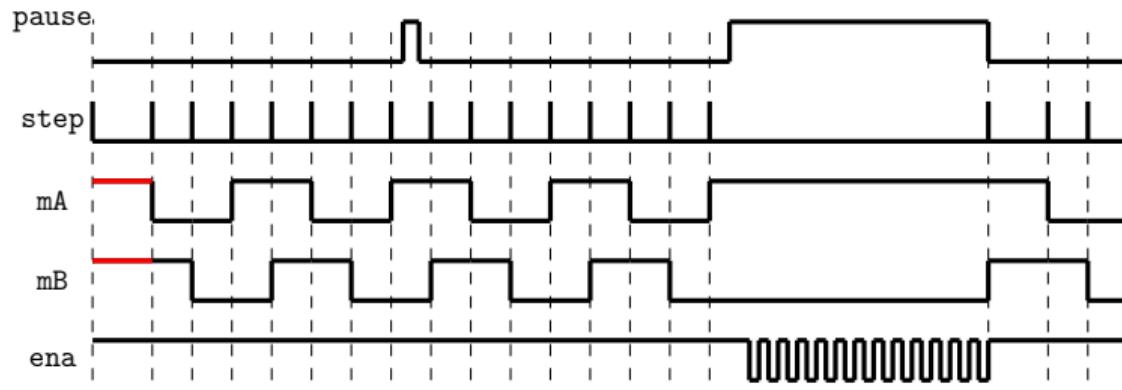
An embedded example: stepper motor for a small printer



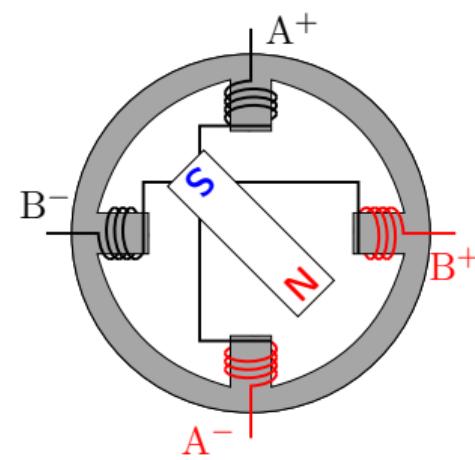
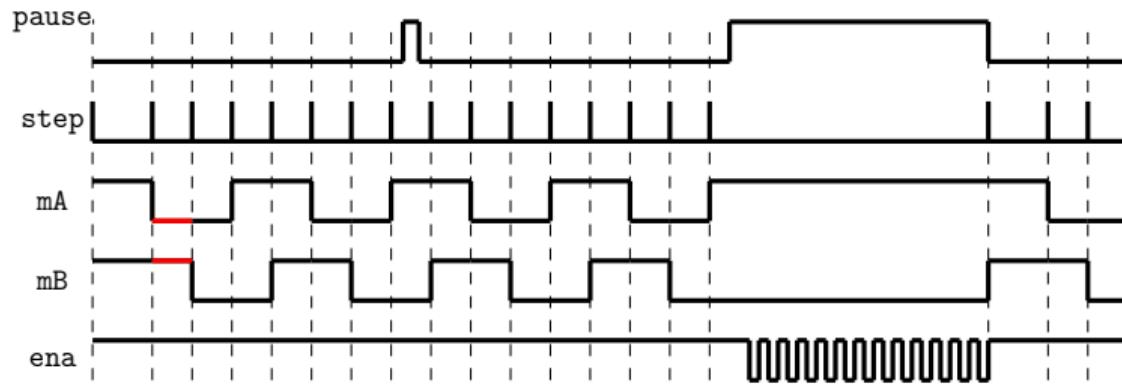
An embedded example: stepper motor for a small printer



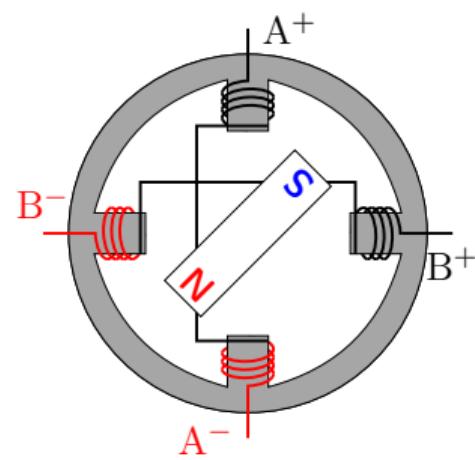
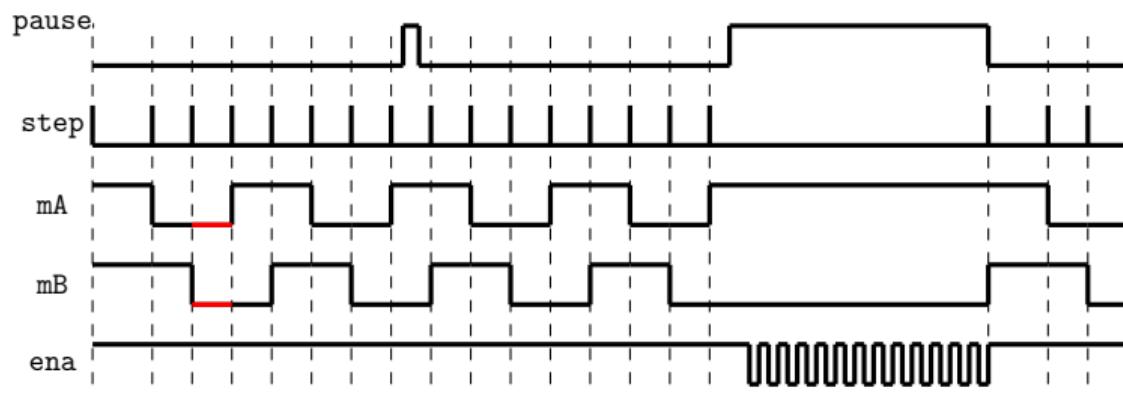
An embedded example: stepper motor for a small printer



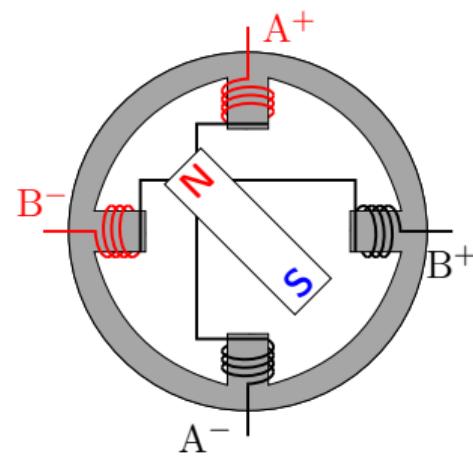
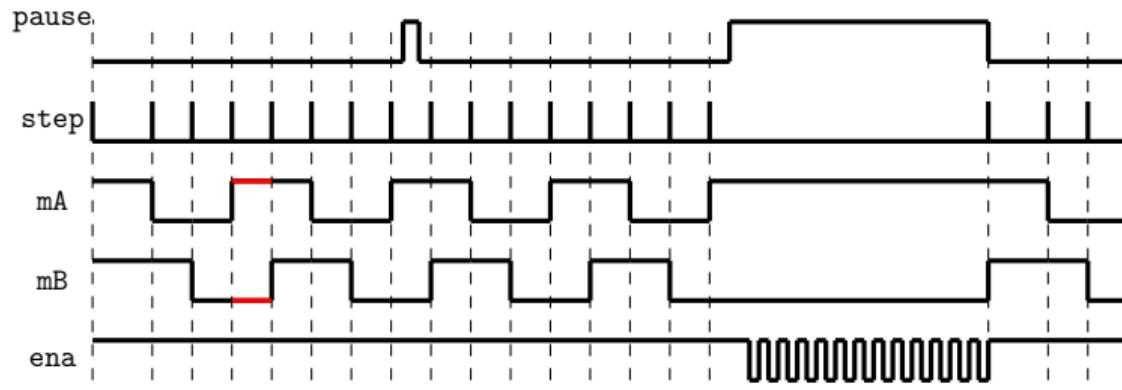
An embedded example: stepper motor for a small printer



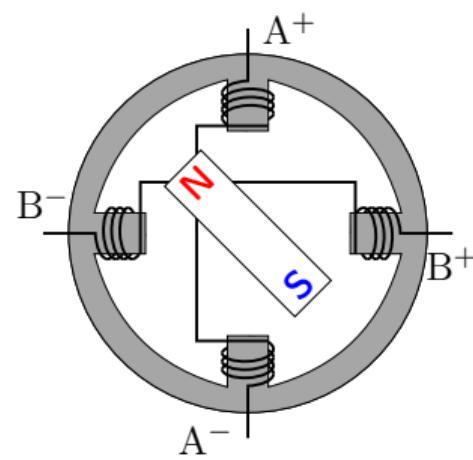
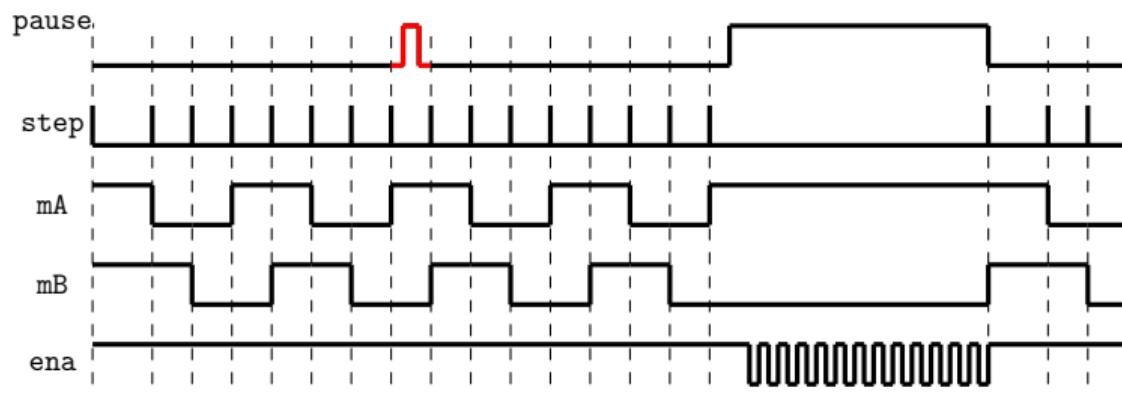
An embedded example: stepper motor for a small printer



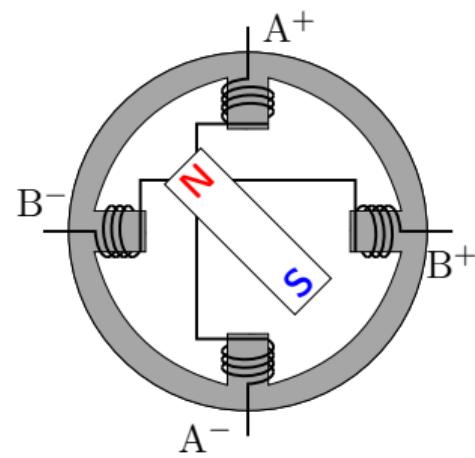
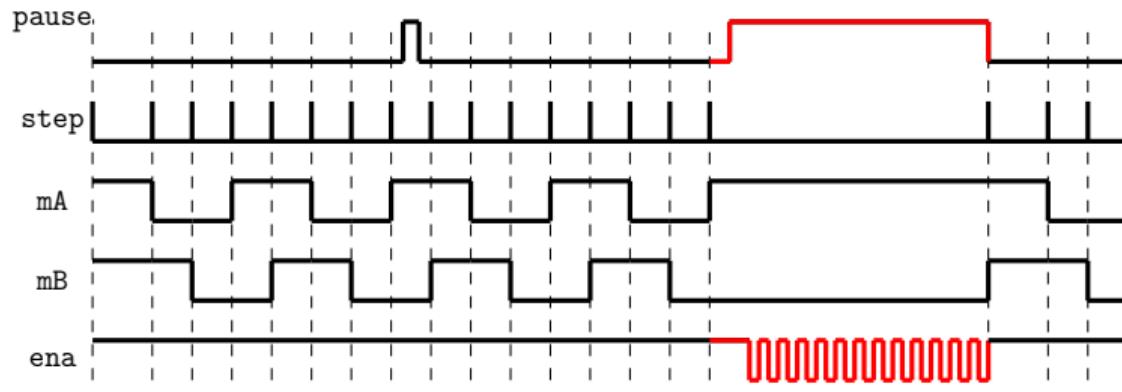
An embedded example: stepper motor for a small printer



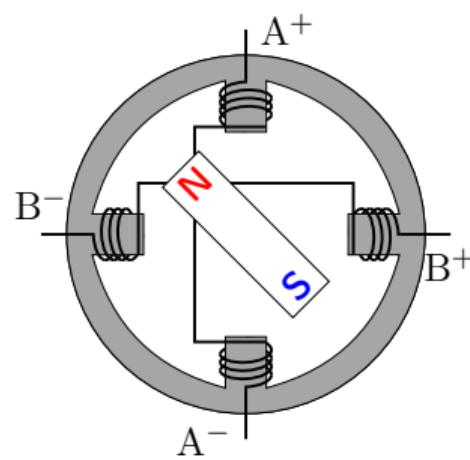
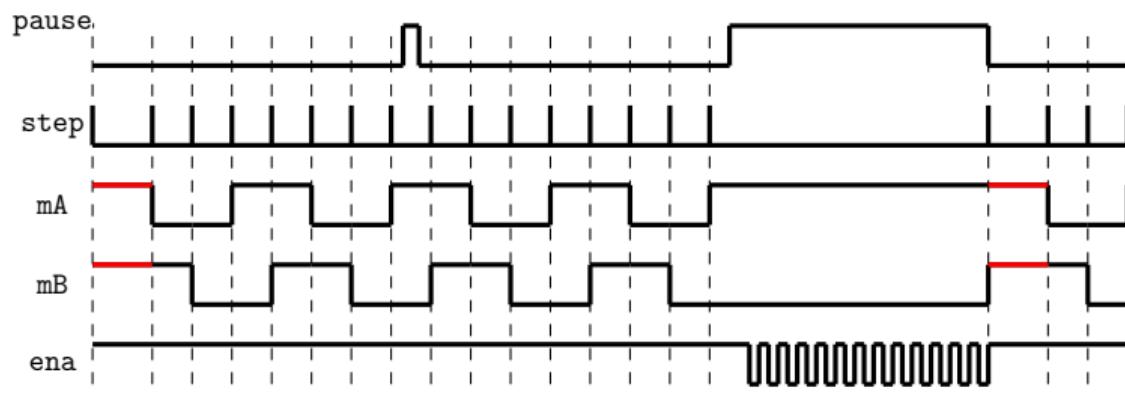
An embedded example: stepper motor for a small printer



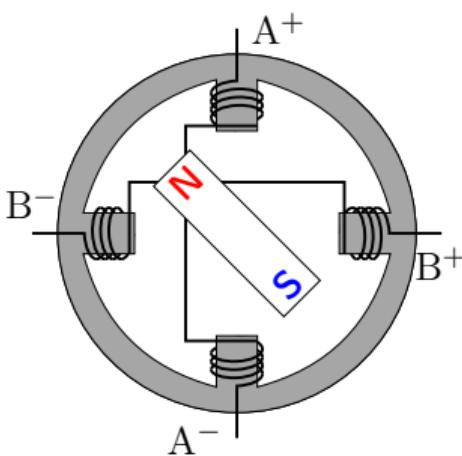
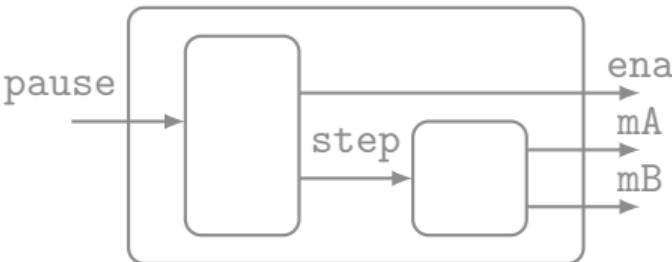
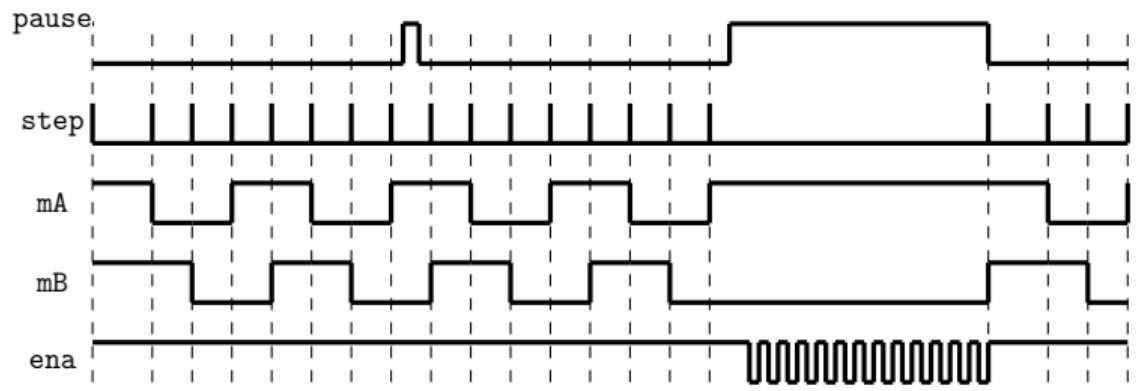
An embedded example: stepper motor for a small printer



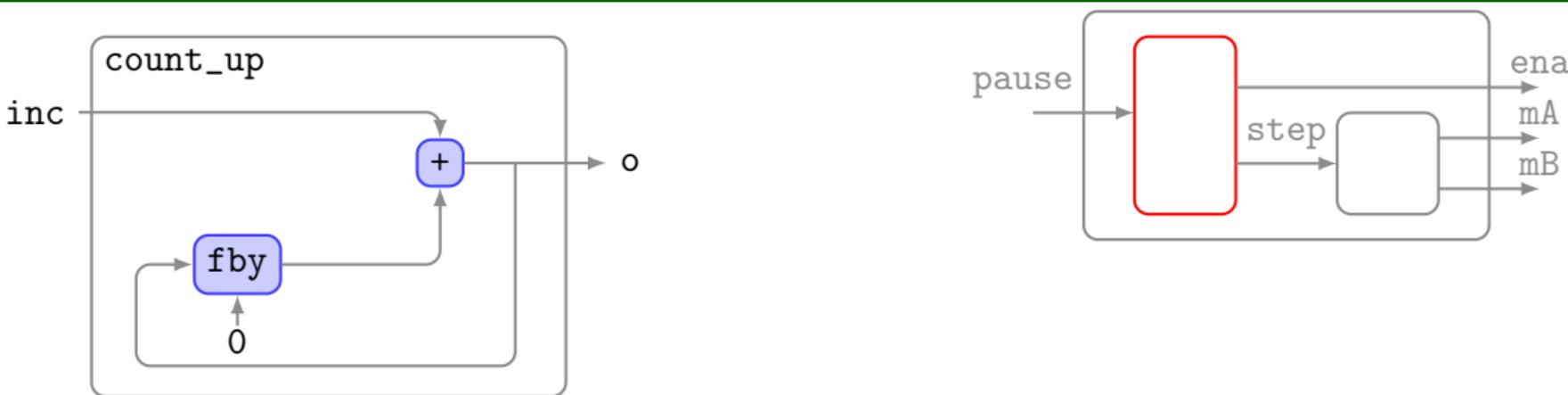
An embedded example: stepper motor for a small printer



An embedded example: stepper motor for a small printer

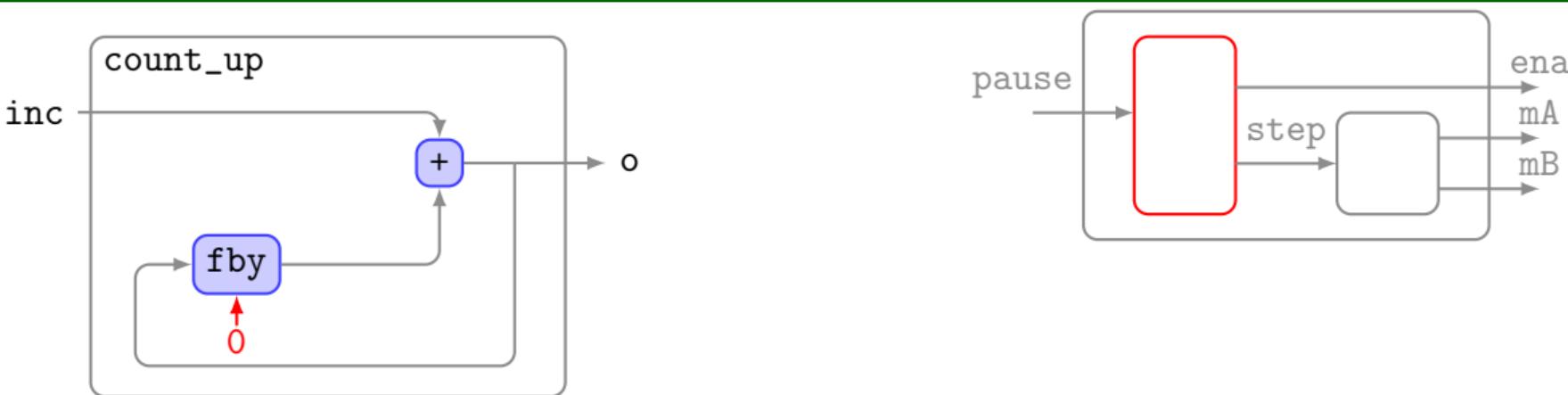


A simple dataflow program



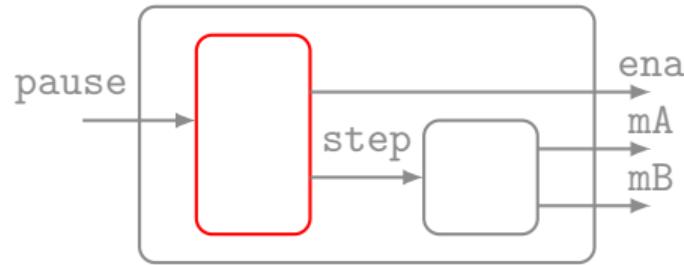
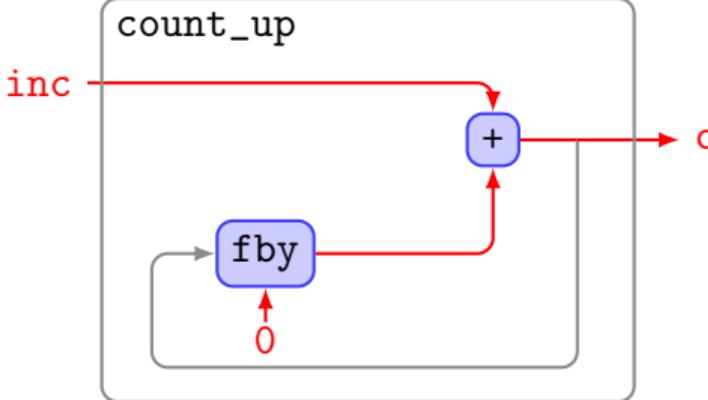
<code>inc</code>	5	4	1	3	2	8	3	...
<code>0 fby o</code>								
<code>o</code>								

A simple dataflow program



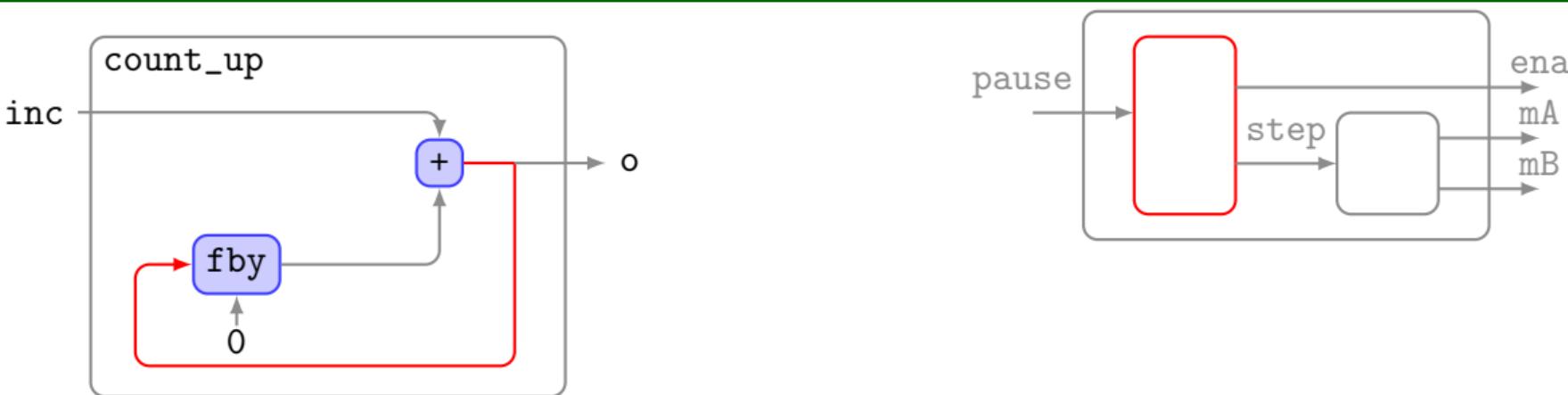
inc	5	4	1	3	2	8	3	...
0	fby	o	0					
			o					

A simple dataflow program



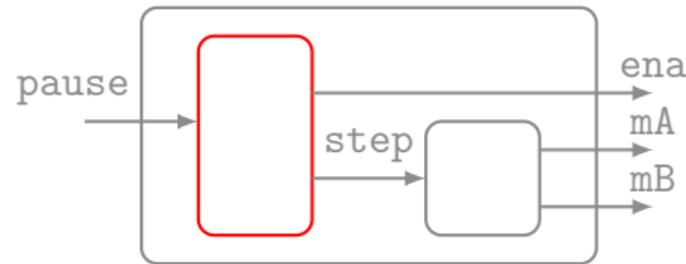
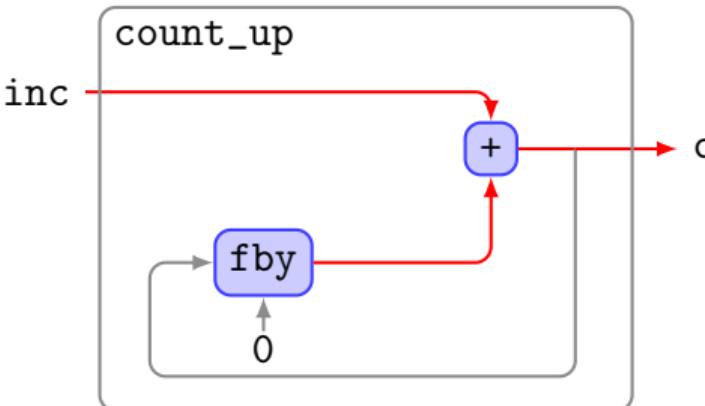
inc	5	4	1	3	2	8	3	...
0	fby	0	0	0	0	0	0	0
o				5				

A simple dataflow program



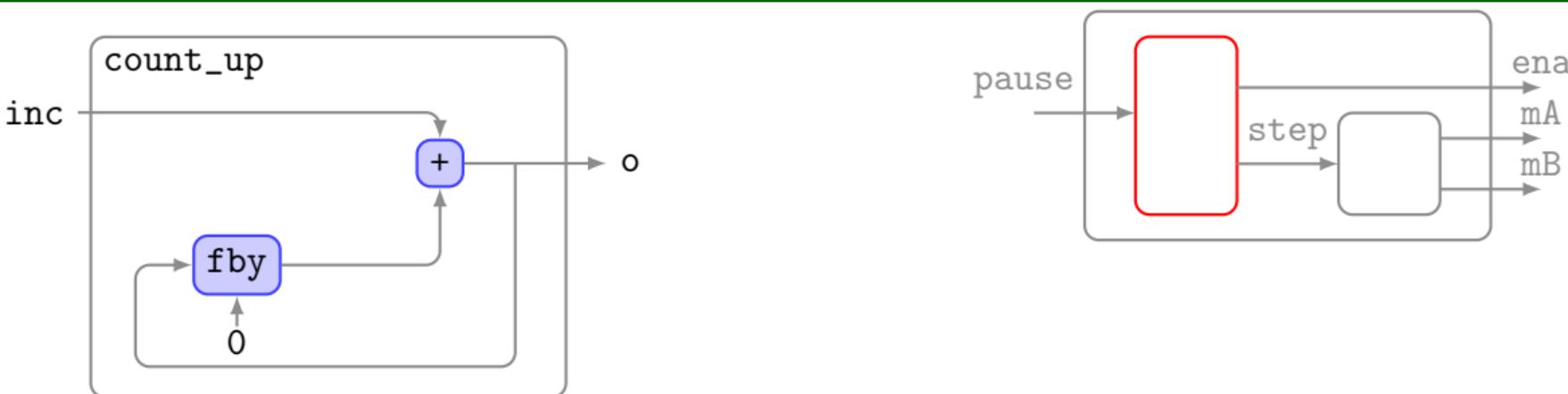
inc	5	4	1	3	2	8	3	...
0	fby	0	5					
o				5				

A simple dataflow program



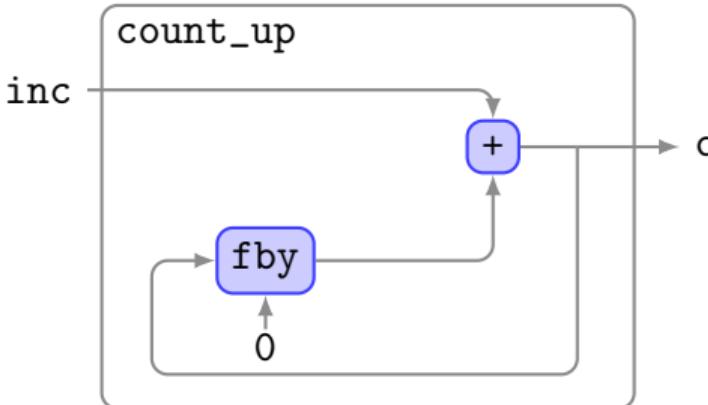
inc	5	4	1	3	2	8	3	...
0 fby o	0	5						
o	5	9						

A simple dataflow program

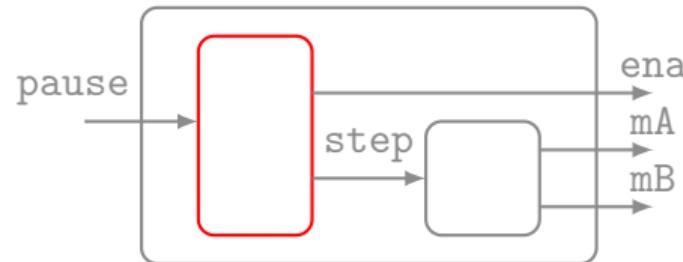


inc	5	4	1	3	2	8	3	...
0 fby o	0	5	9	10	13	15	23	...
o	5	9	10	13	15	23	26	...

A simple dataflow program



```
node count_up(inc : int)
returns (o : int)
let
    o = (0 fby o) + inc;
tel
```

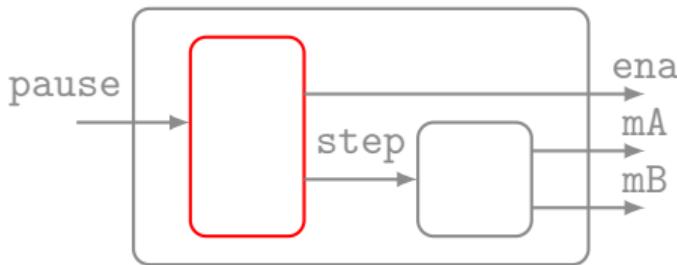


inc	5	4	1	3	2	8	3	...
0 fby o	0	5	9	10	13	15	23	...
o	5	9	10	13	15	23	26	...

Modular resetting of equations

reset

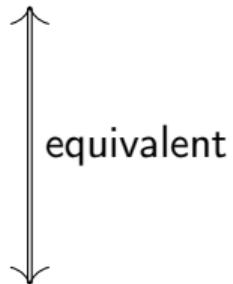
```
time = count_up(50)
every (false fby step)
```



Modular resetting of equations

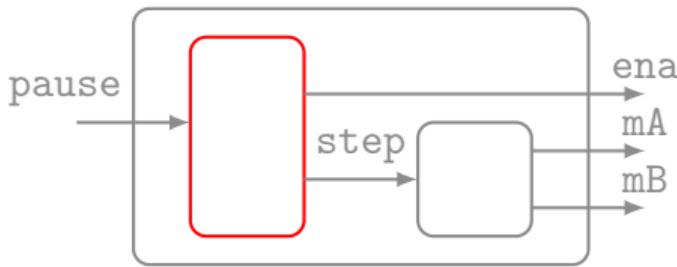
```
reset
```

```
  time = count_up(50)  
every (false fby step)
```

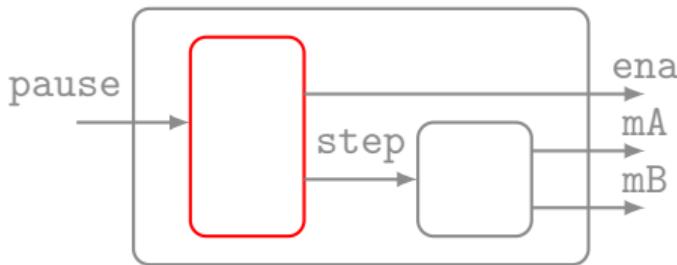


```
reset
```

```
  time = (0 fby time) + 50  
every (false fby step)
```

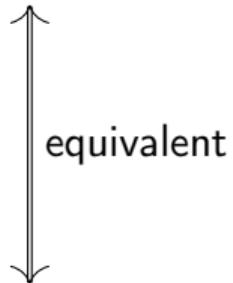


Modular resetting of equations



reset

```
time = count_up(50)  
every (false fby step)
```

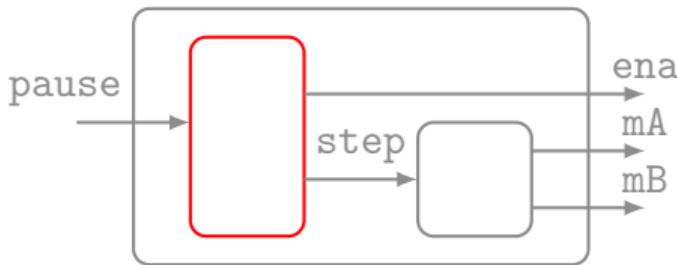


reset

```
time = (0 fby time) + 50  
every (false fby step)
```

step	F	F	T	...
time	50	100	150	...

Modular resetting of equations



reset

```
time = count_up(50)
every (false fby step)
```

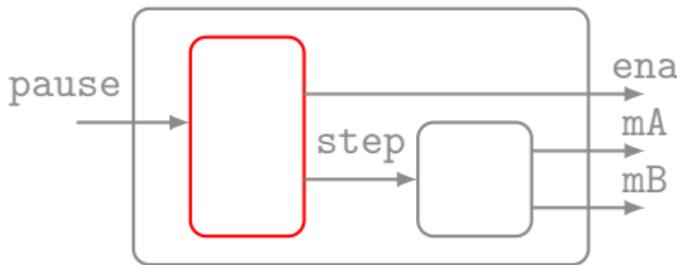
↑
equivalent
↓

reset

```
time = (0 fby time) + 50
every (false fby step)
```

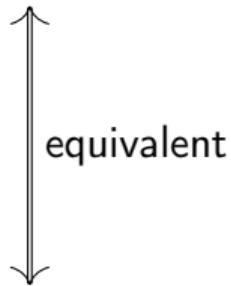
step	F	F	T	F	F	F	T	...
time				50	100	150	200	...

Modular resetting of equations



reset

```
time = count_up(50)  
every (false fby step)
```

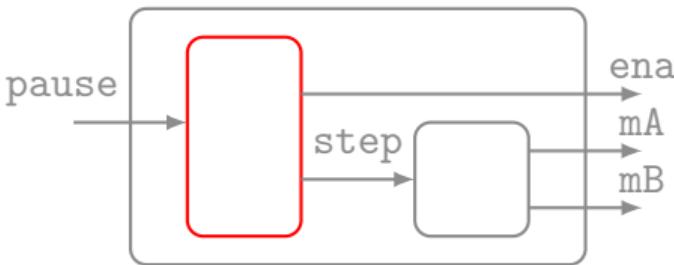


reset

```
time = (0 fby time) + 50  
every (false fby step)
```

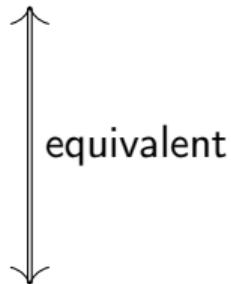
step	F	F	T	F	F	F	T	F	...
time								50	...

Modular resetting of equations



reset

```
time = count_up(50)
every (false fby step)
```

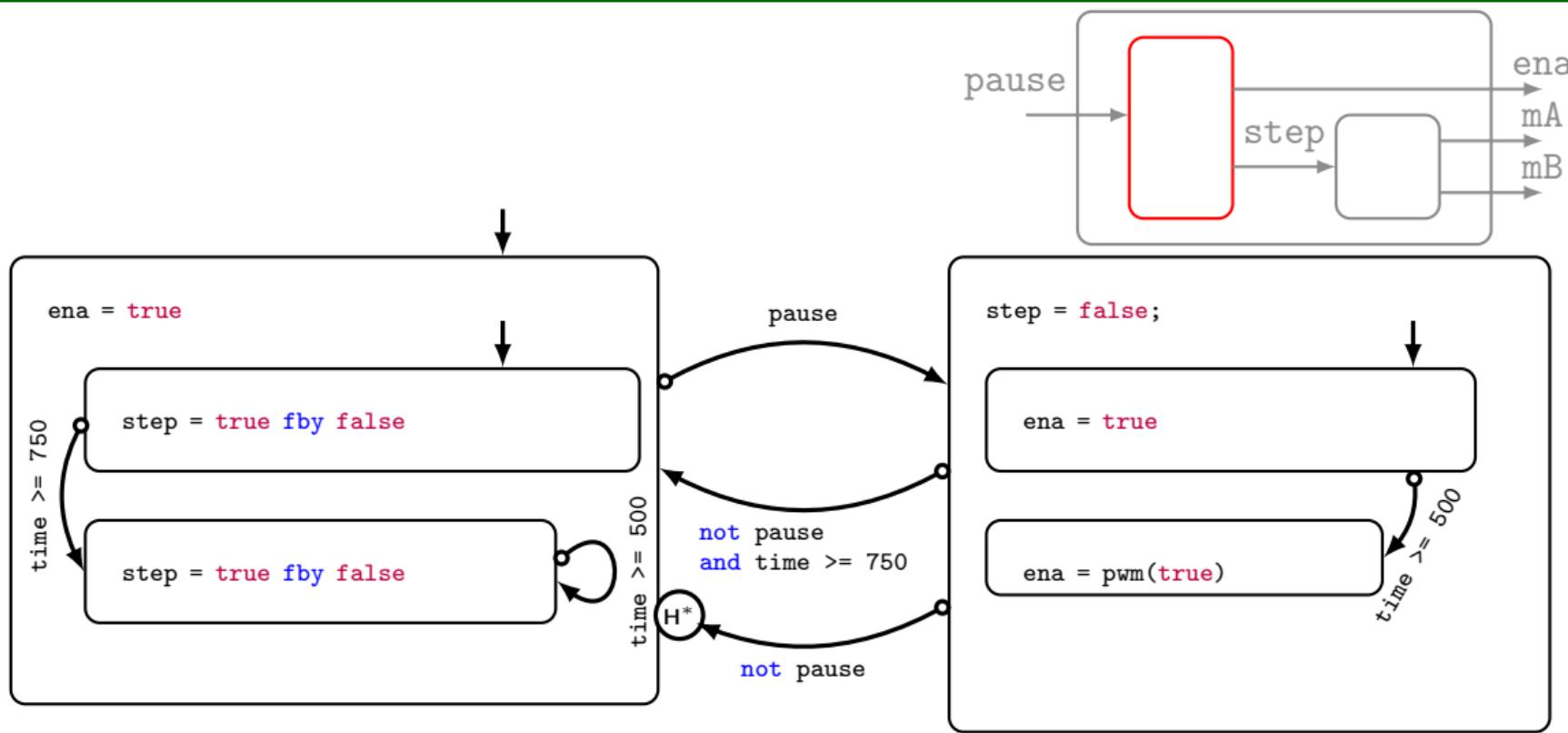


reset

```
time = (0 fby time) + 50
every (false fby step)
```

step	F	F	T	F	F	F	T	F	...
time	50	100	150	50	100	150	200	50	...

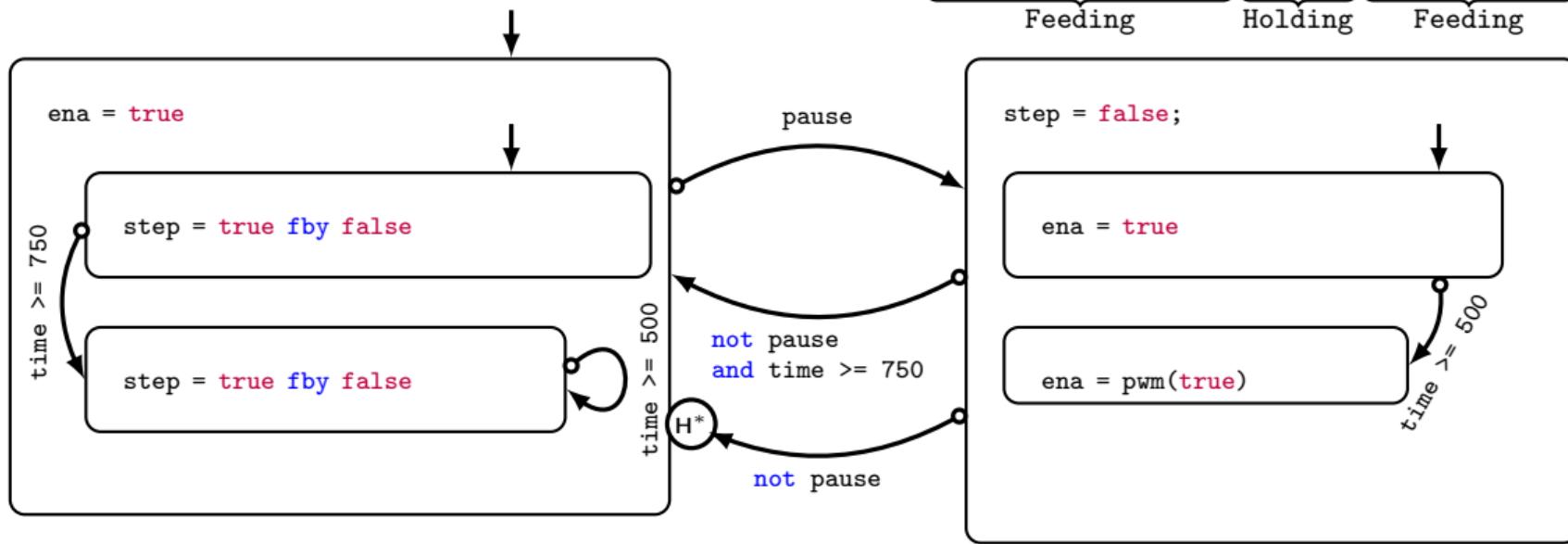
Hierarchical State Machines



Hierarchical State Machines

pause	F F F ... F F ... T ... F ... F ...
time	0 0 50 ... 750 0 ... 150 ... 350 ... 500 ...
step	T F F ... T F ... F ... F ... T ...
ena	T T T ... T T ... T ... T ... T ...

Feeding Holding Feeding



Hierarchical State Machines

```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);

```

automaton initially Feeding

```
state Feeding do
  ena = true;
  automaton initially Starting
    state Starting do
      step = true fby false
      unless time >= 750 then Moving
    end;
    state Moving do
      step = true fby false
      unless time >= 500 then Moving
    end;
    unless pause then Holding
  end
tel
```

pause	F F F ... F F ... T ... F ... F ...
time	0 0 50 ... 750 0 ... 150 ... 350 ... 500 ...
step	T F F ... T F F ... T F ... T F ... T ...
ena	T T T ...

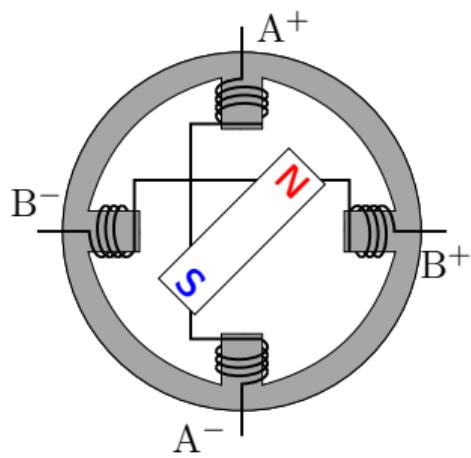
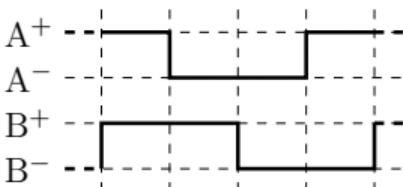
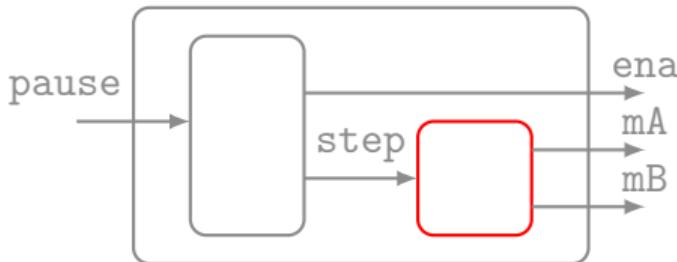
Feeding Holding Feeding

```
state Holding do
  step = false;
  automaton initially Waiting
    state Waiting do
      ena = true
      unless time >= 500 then Modulating
    end;
    state Modulating do
      ena = pwm(true)
    end;
    unless
      | not pause and time >= 750 then Feeding
      | not pause continue Feeding
    end
  end
end
```

Switch blocks

```
mA = not (last mB);  
mB = last mA;
```

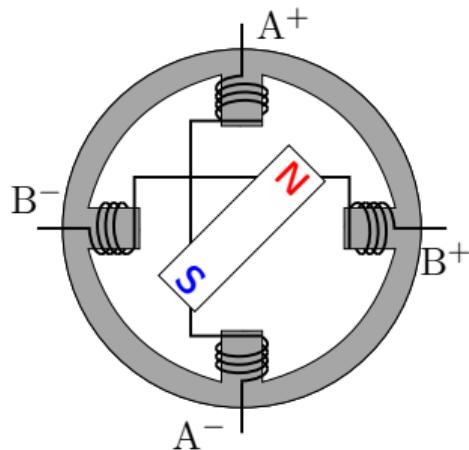
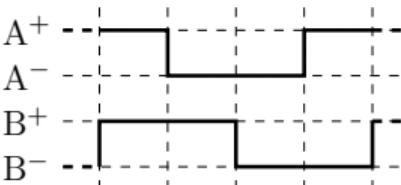
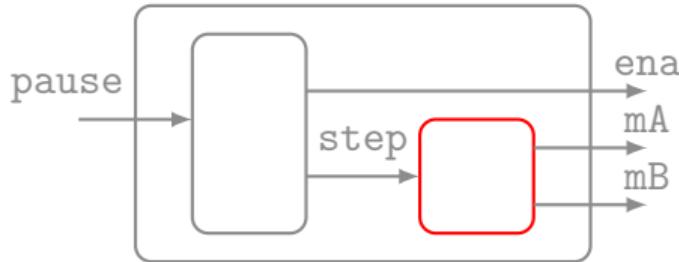
```
last mA = true;  
last mB = false;
```



Switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
```

step	F T T F F T F T F T F ...
last mA	T ...
last mB	F ...
mA	T ...
mB	F ...

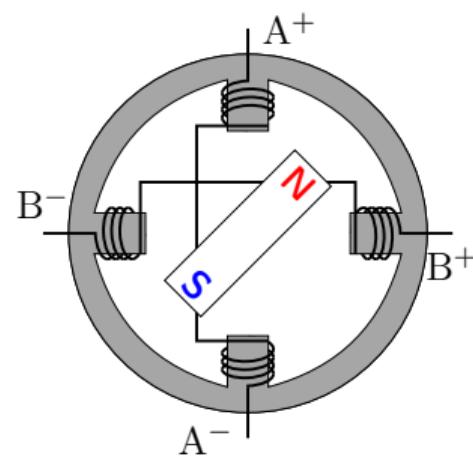
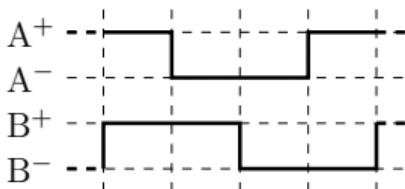
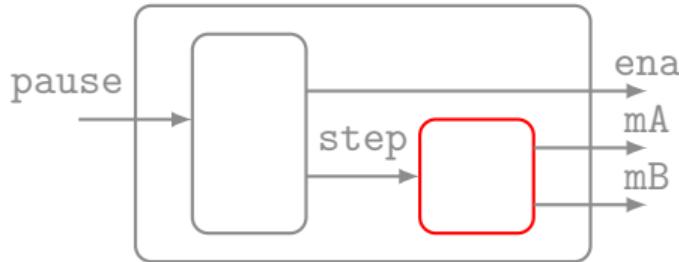


Switch blocks

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
  
```

step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T	T	T									...
last mB	F	F	T									...
mA	T	T	F									...
mB	F	T	T									...

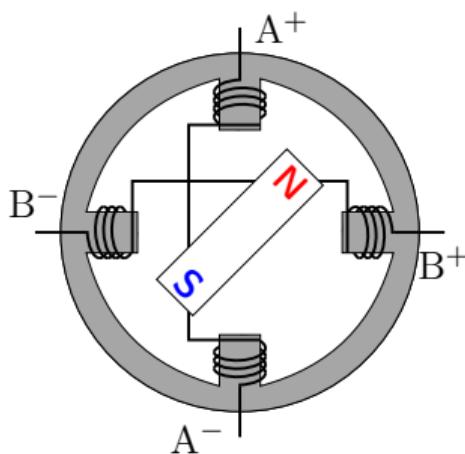
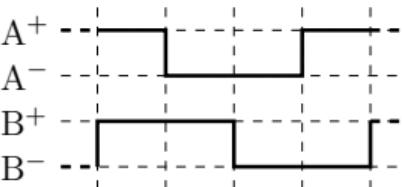
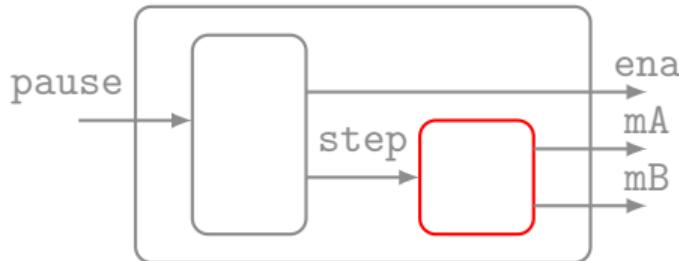


Switch blocks

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
  
```

step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T	T	T	F	F							...
last mB	F	F	T	T	T							...
mA	T	T	F	F	F							...
mB	F	T	T	T	T							...

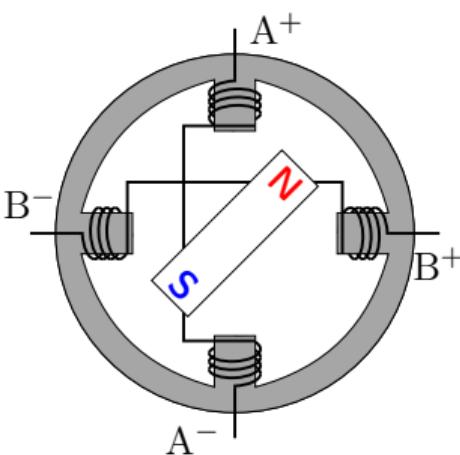
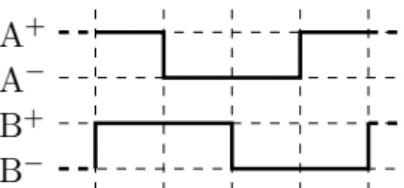
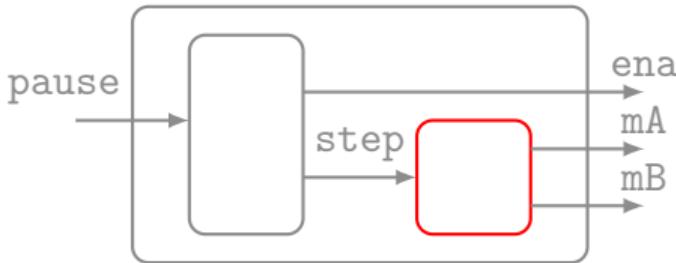


Switch blocks

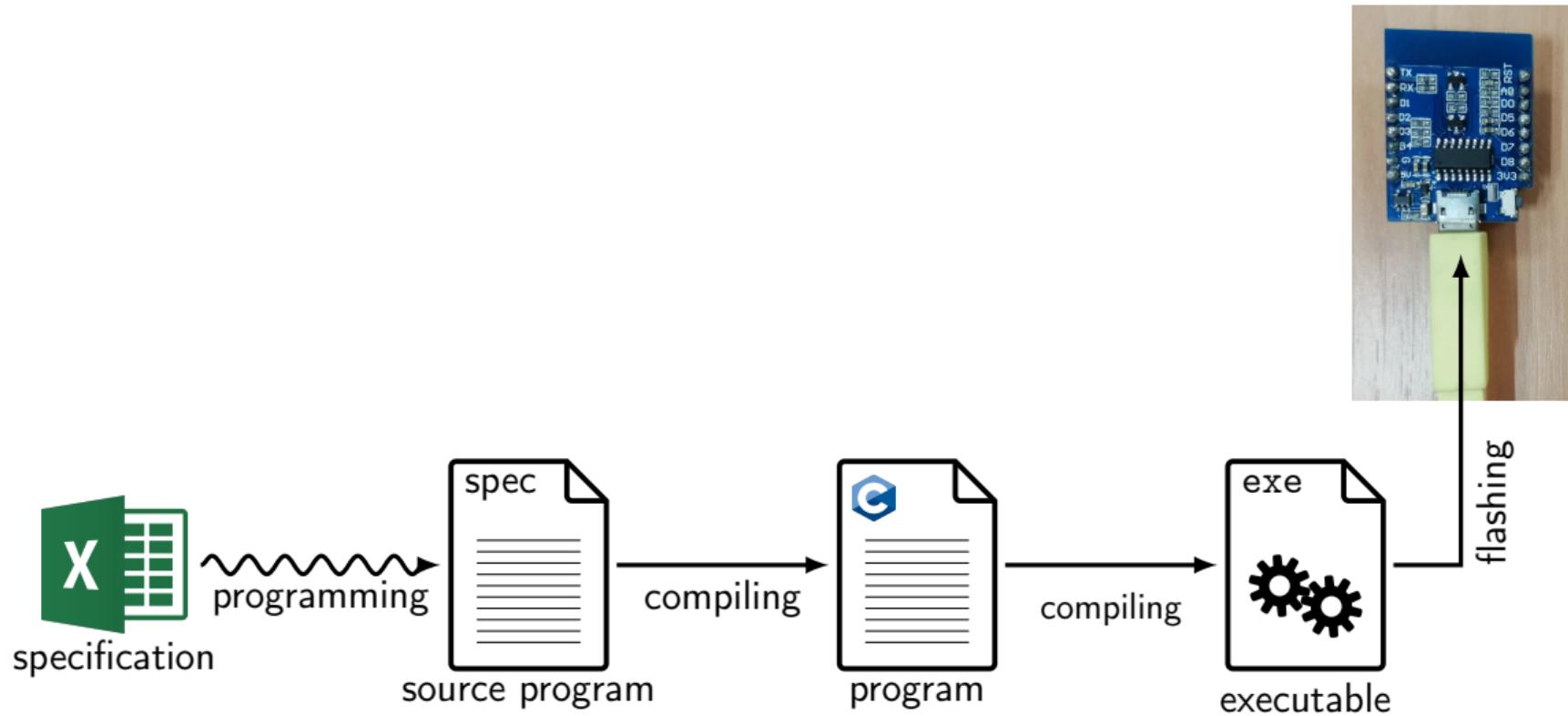
```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
  
```

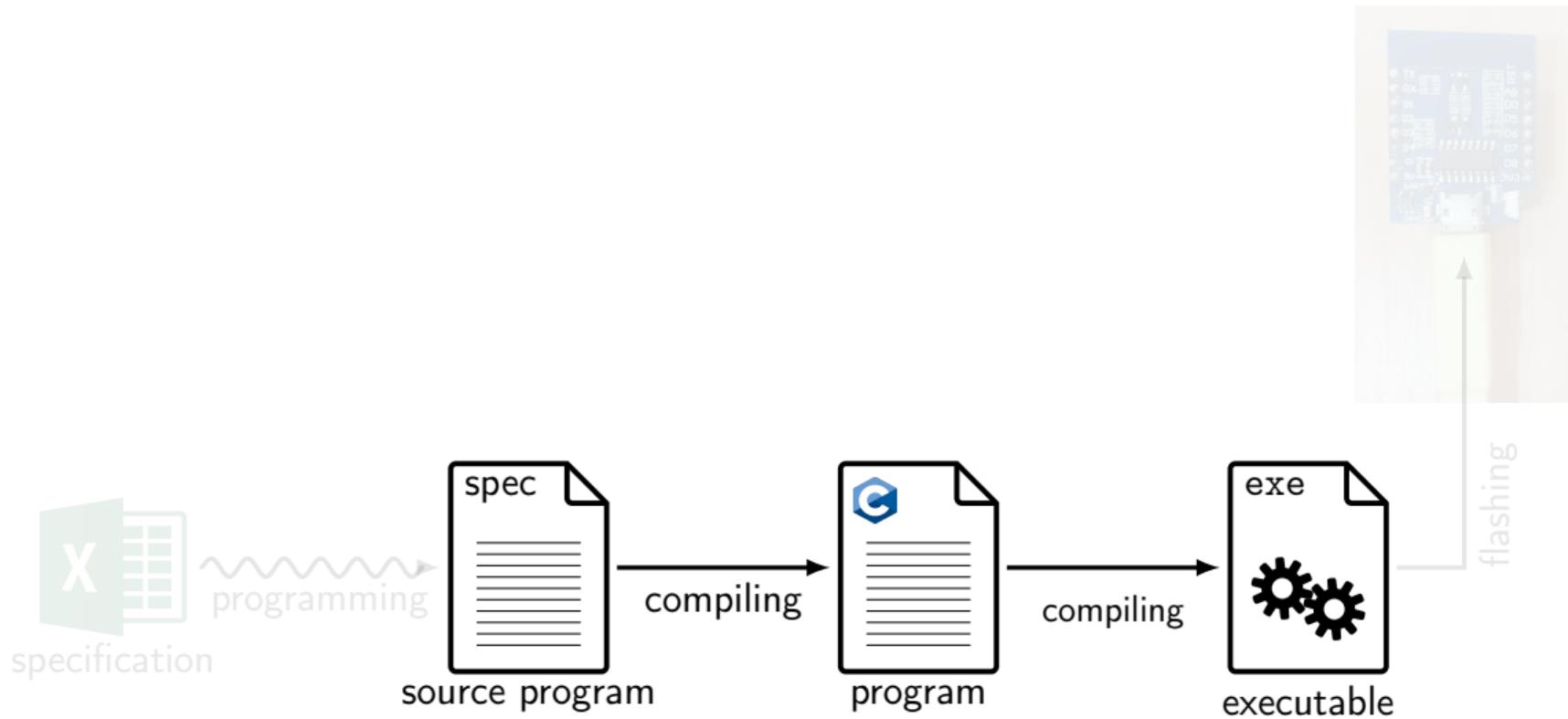
step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T	T	T	F	F	F	F	F	T	T	T	...
last mB	F	F	T	T	T	T	F	F	F	F	T	...
mA	T	T	F	F	F	F	F	T	T	T	T	...
mB	F	T	T	T	T	F	F	F	T	T	T	...



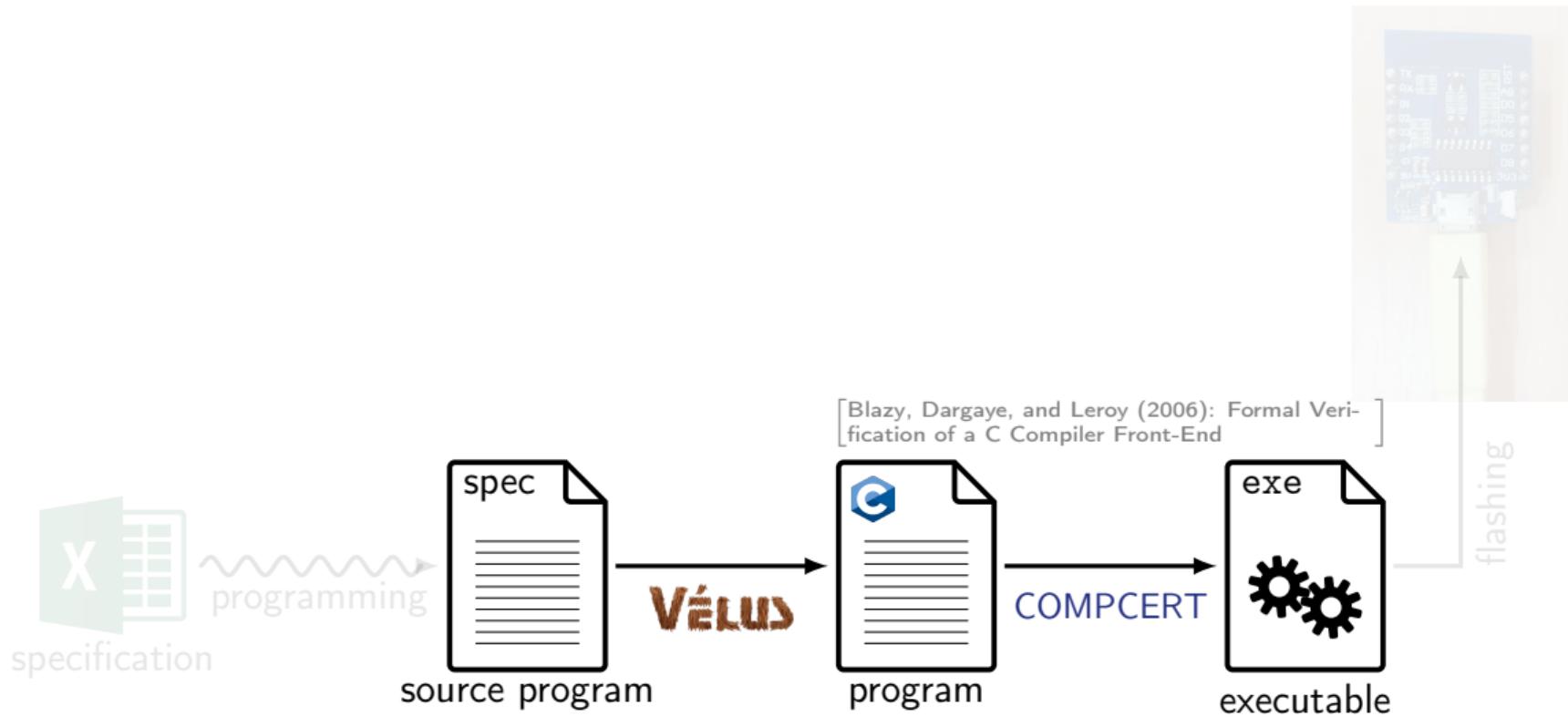
Compiling Lustre to C



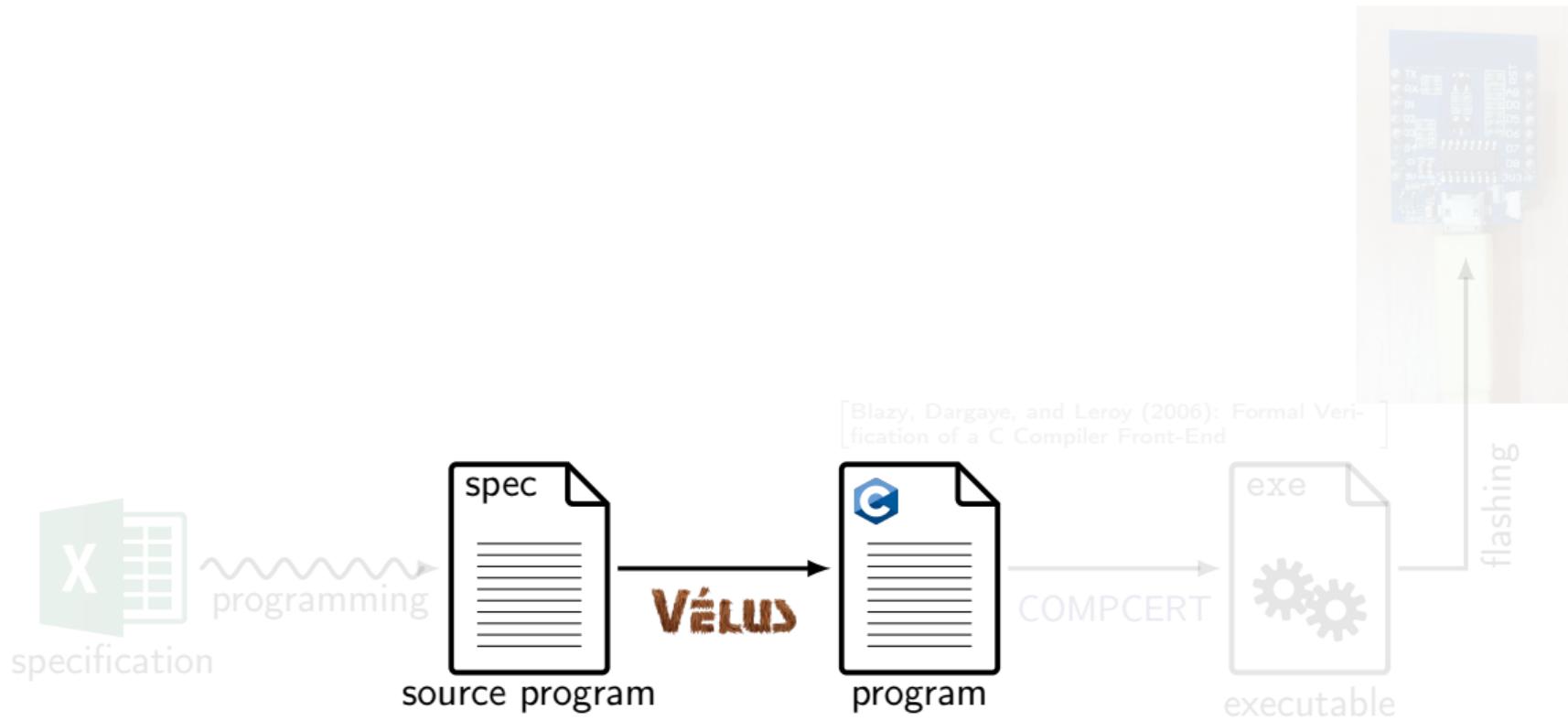
Compiling Lustre to C



Compiling Lustre to C



Compiling Lustre to C



Compiling Lustre to C

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

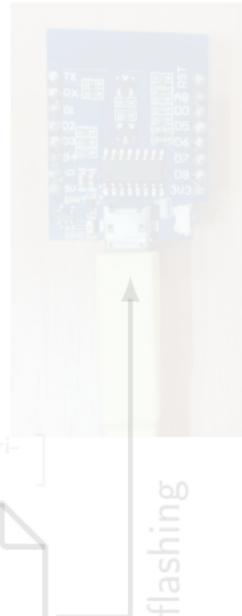


VÉLUS

[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]

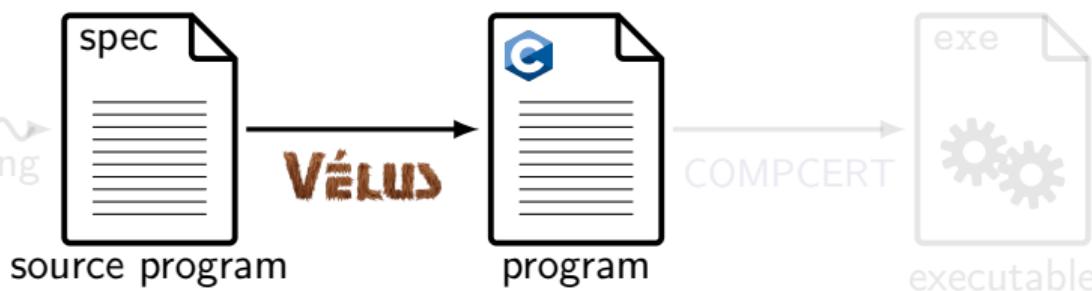


COMPCERT



Compiling Lustre to C

```
node count_up(inc : int)      struct count_up {  
    returns (o : int)          int norm$1;  
    let                      };  
    o = (0 fby o) + inc;  
  tel
```

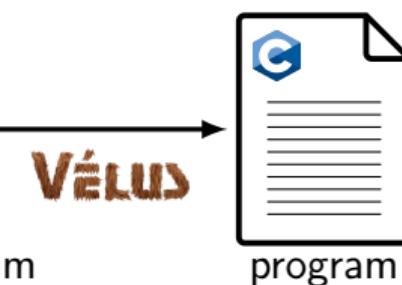


Compiling Lustre to C

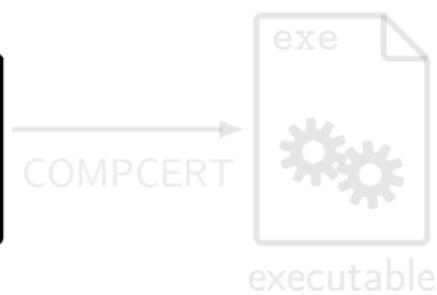
```
node count_up(inc : int)
  returns (o : int)
  let
    o = (0 fby o) + inc;
  tel
```

```
struct count_up {
  int norm$1;
};

void fun$reset$count_up(struct count_up *self) {
  (*self).norm$1 = 0;
}
```



[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]



COMPCERT

flashing
b0

Compiling Lustre to C

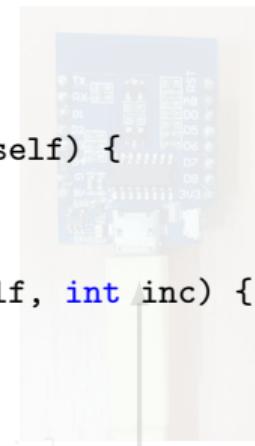
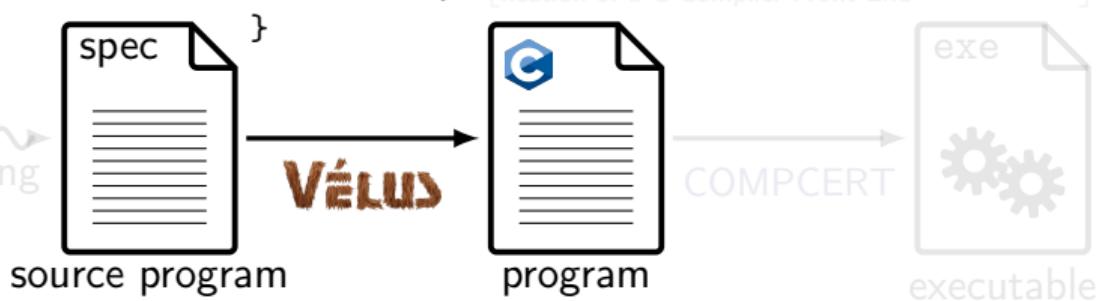
```
node count_up(inc : int)
  returns (o : int)
  let
    o = (0 fby o) + inc;
  tel
```

```
struct count_up {
  int norm$1;
};

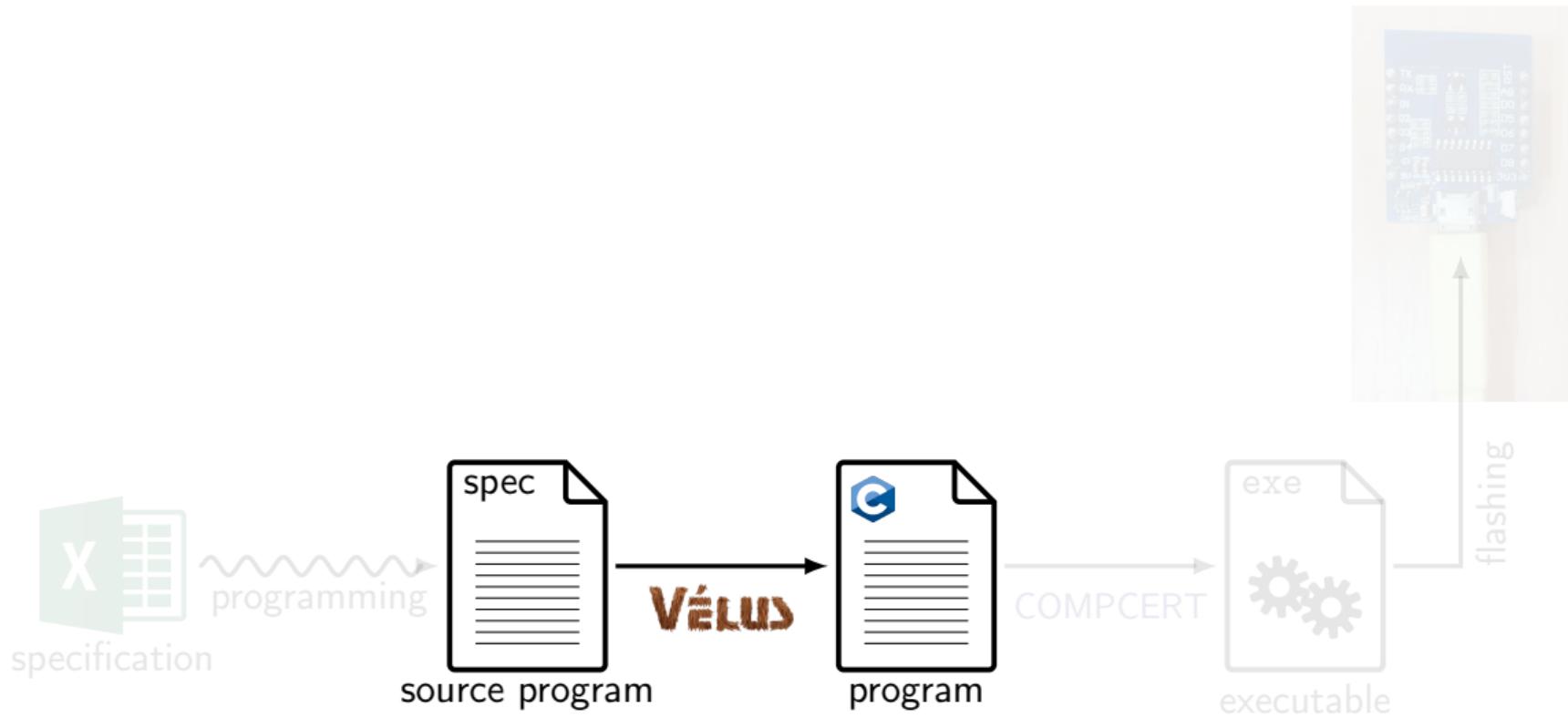
void fun$reset$count_up(struct count_up *self) {
  (*self).norm$1 = 0;
}

int fun$step$count_up(struct count_up *self, int inc) {
  register int o;
  o = (*self).norm$1 + inc;
  (*self).norm$1 = o;
  return o;
}
```

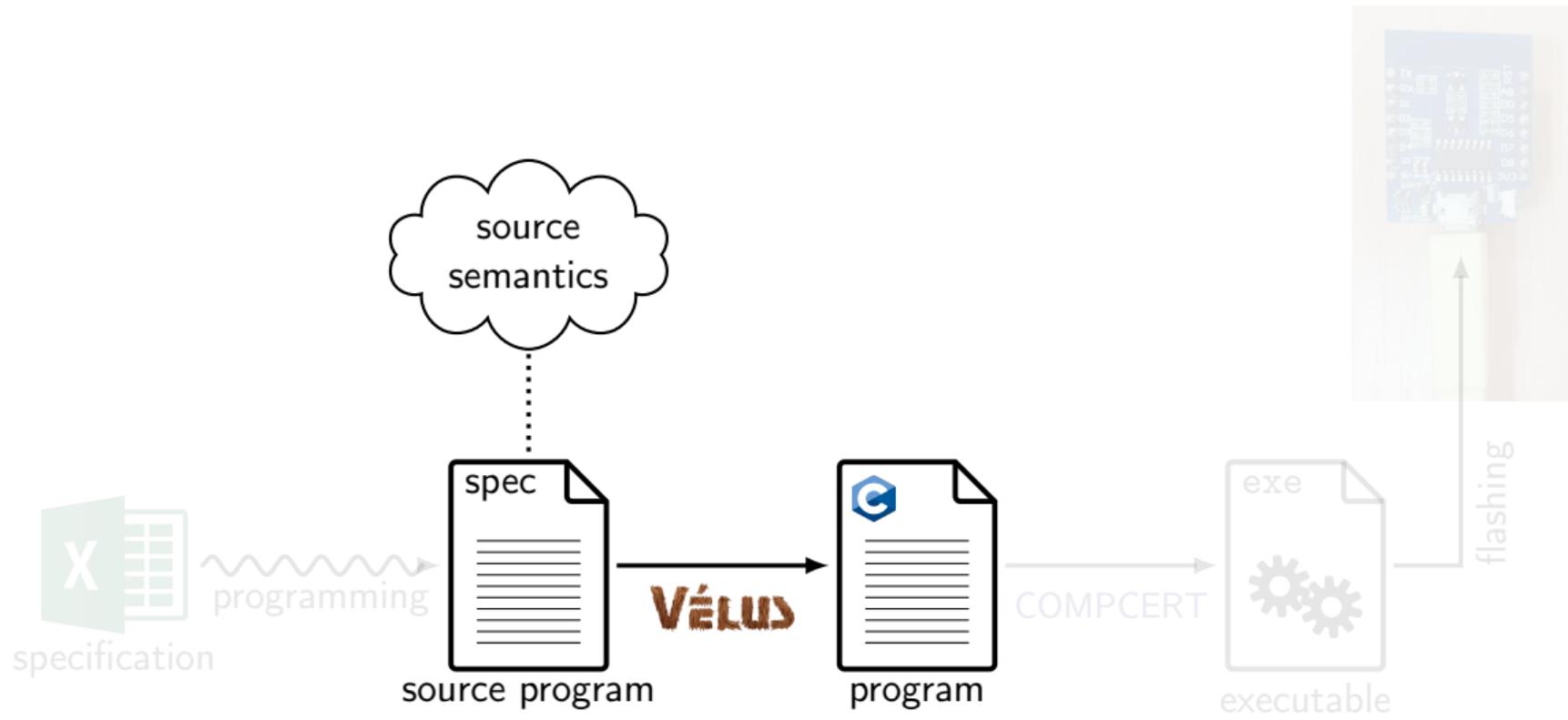
Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End



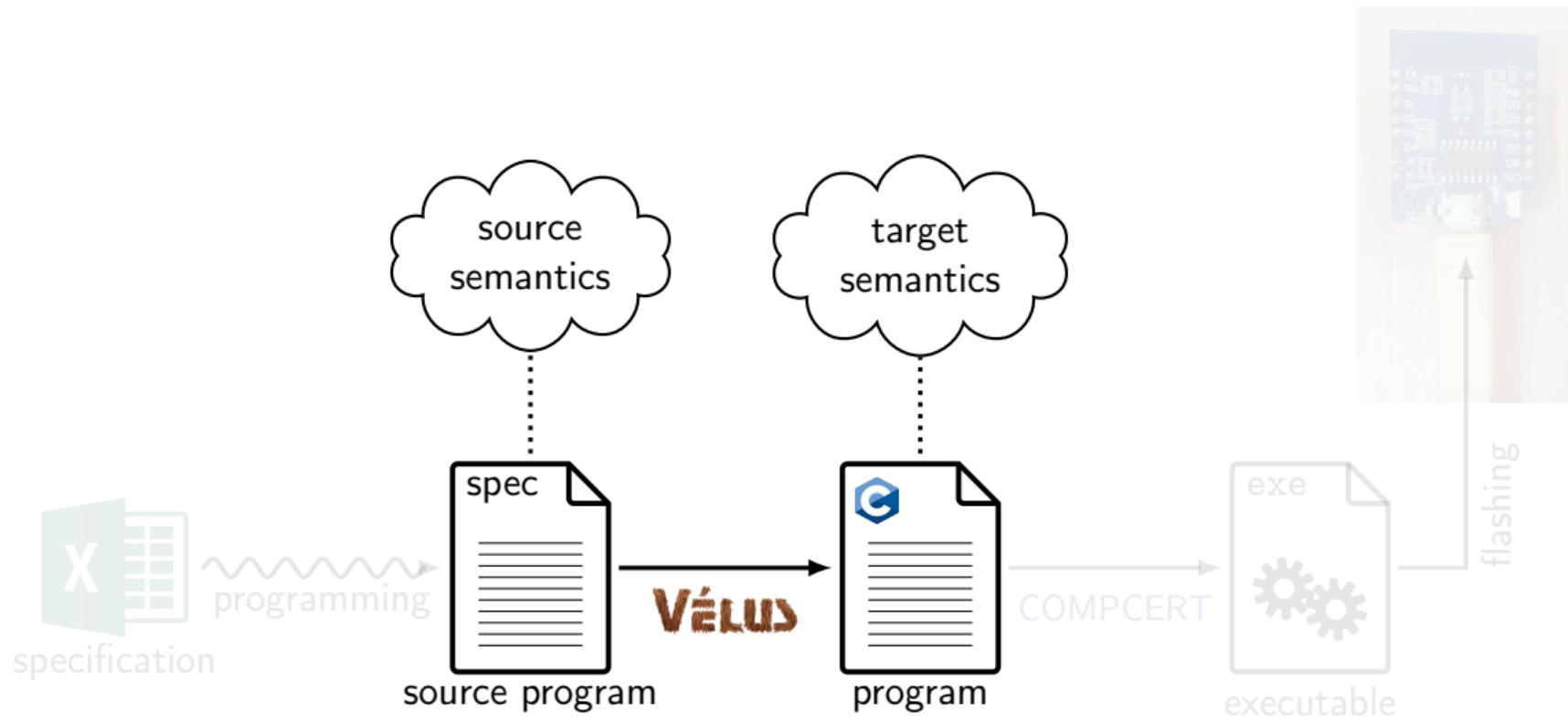
Compiler verification



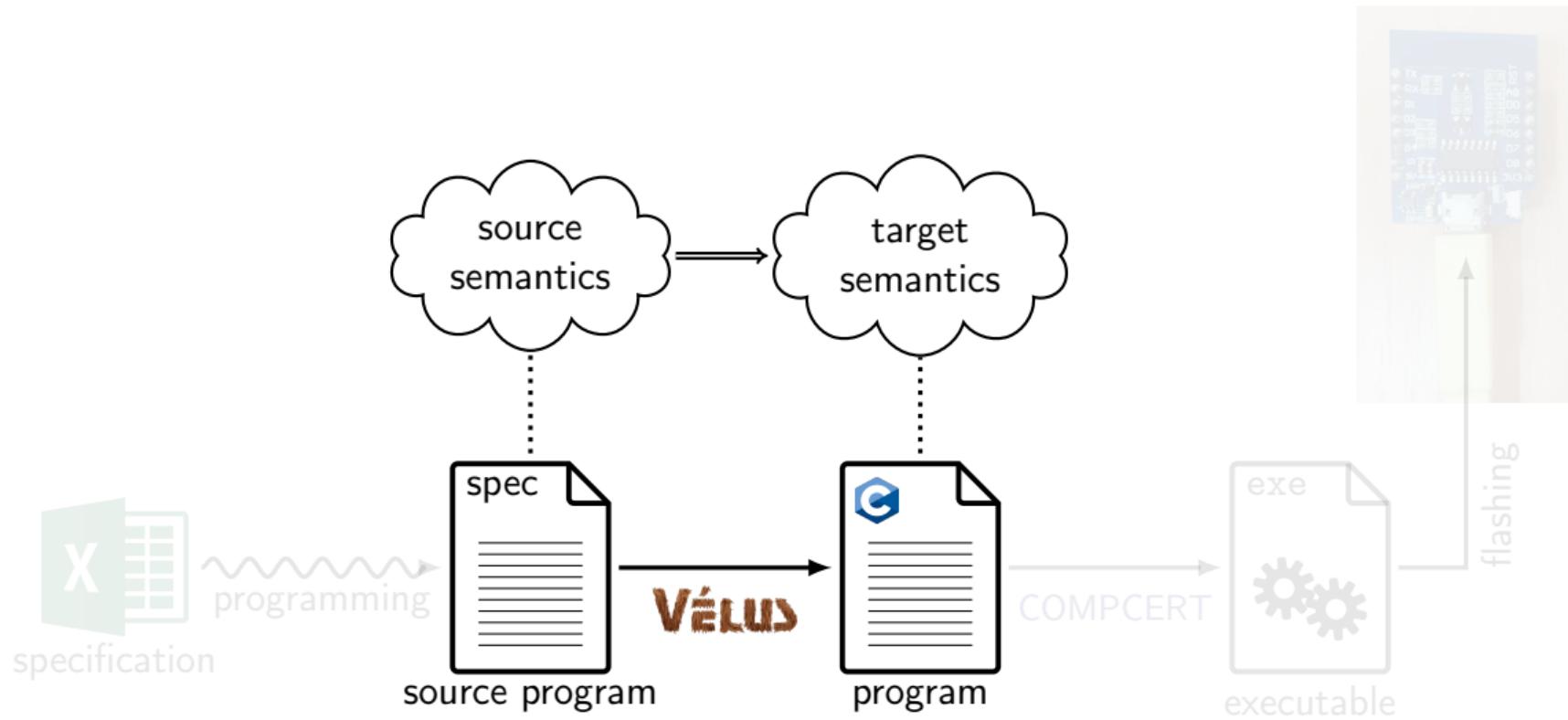
Compiler verification



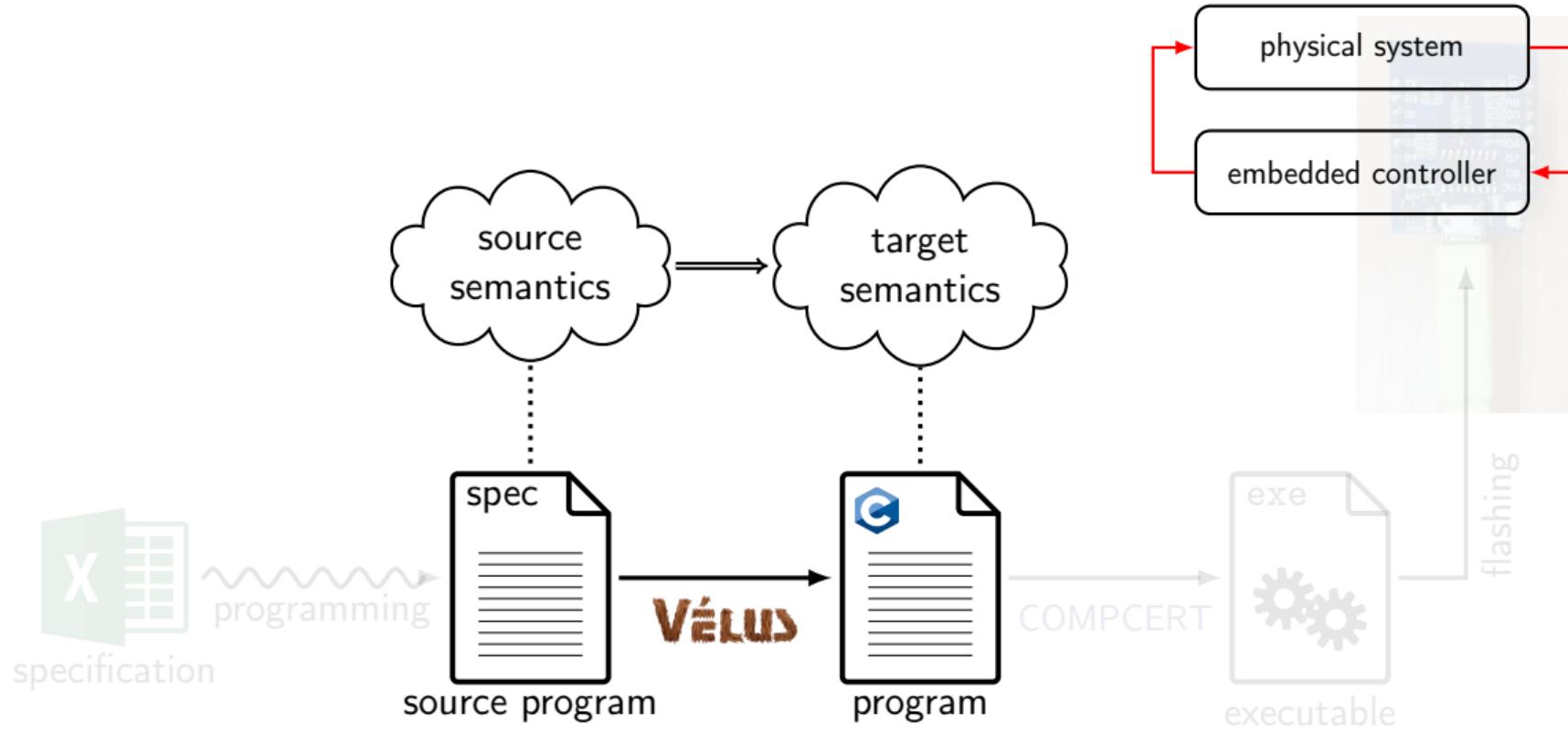
Compiler verification



Compiler verification



Compiler verification

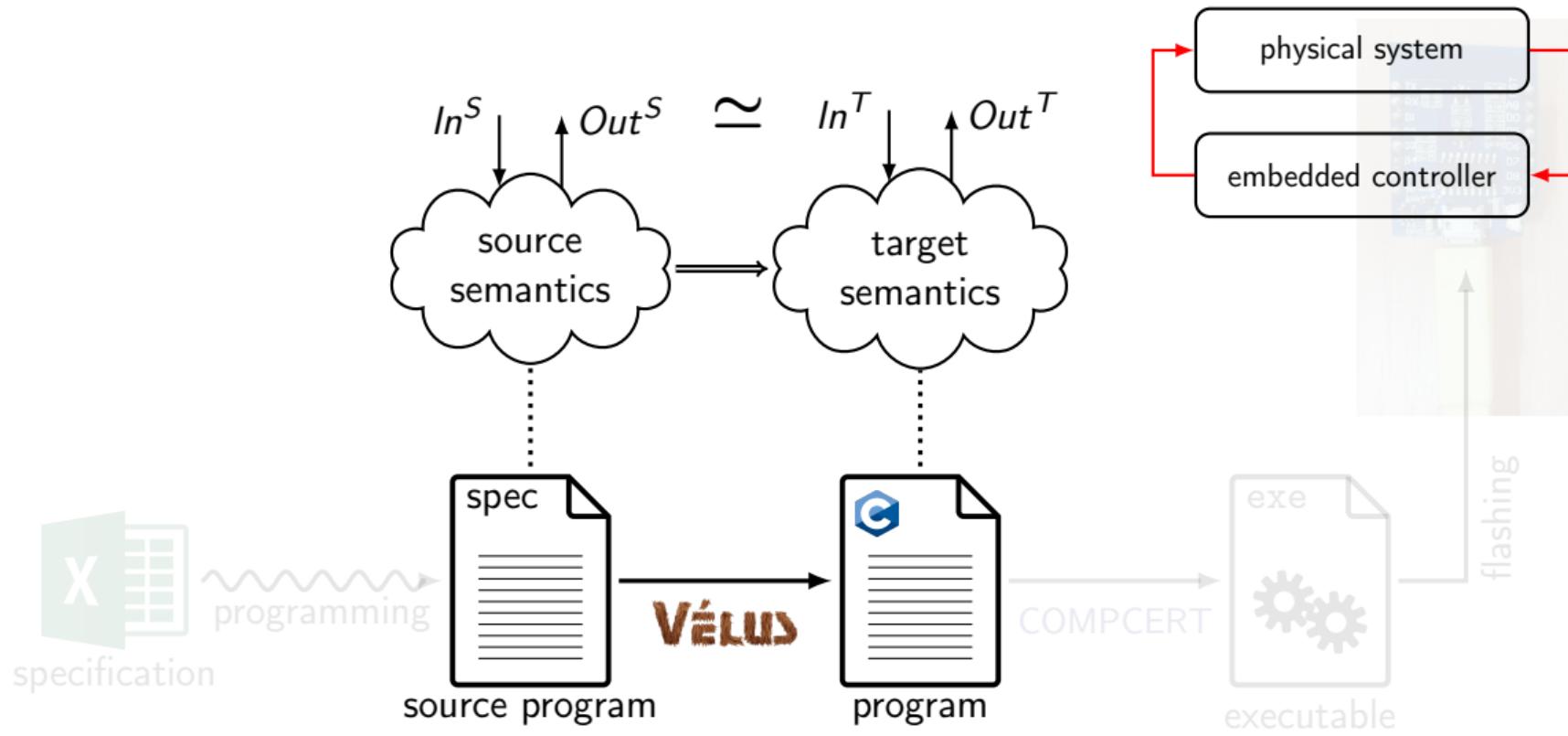


X
specification

wavy line
programming

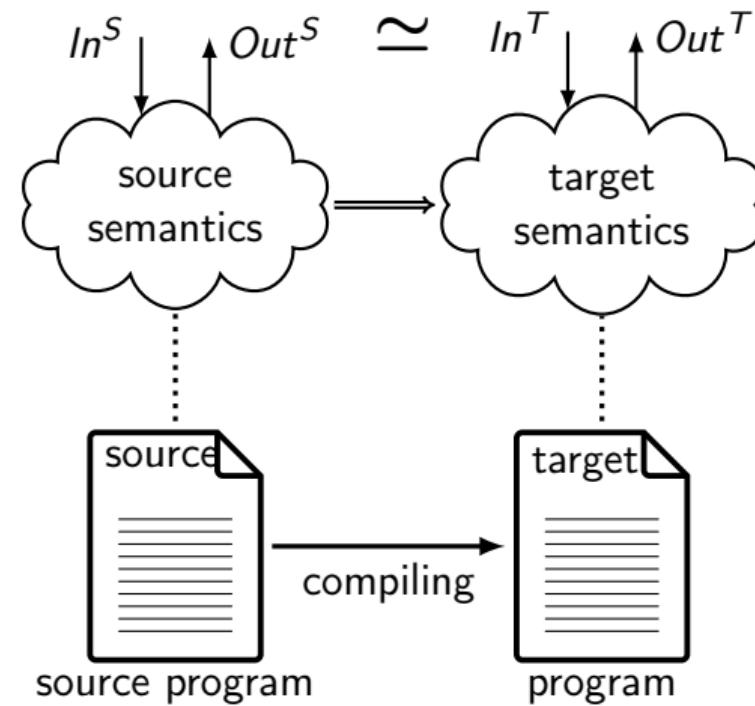
source program

Compiler verification



Compiler verification

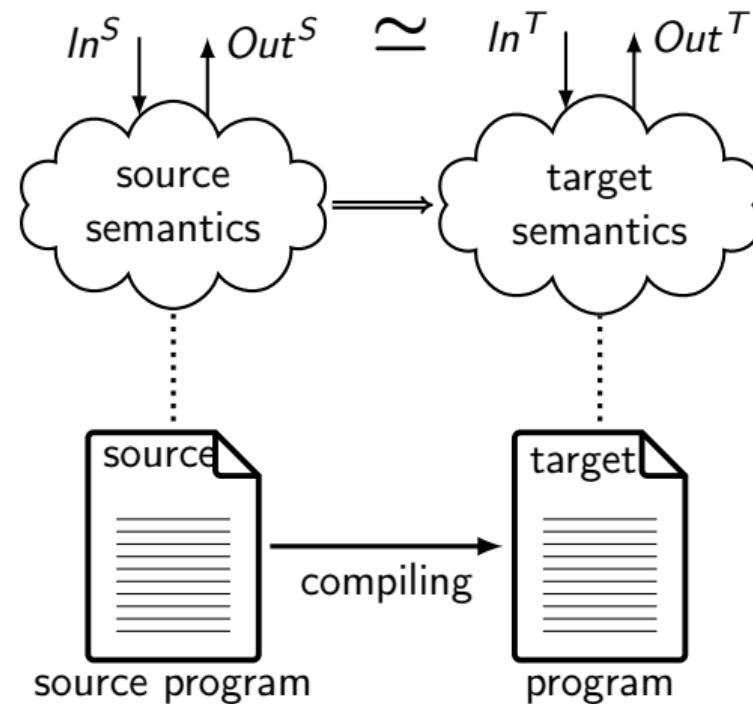
In an Interactive Theorem Prover (recently):



Compiler verification

In an Interactive Theorem Prover (recently):

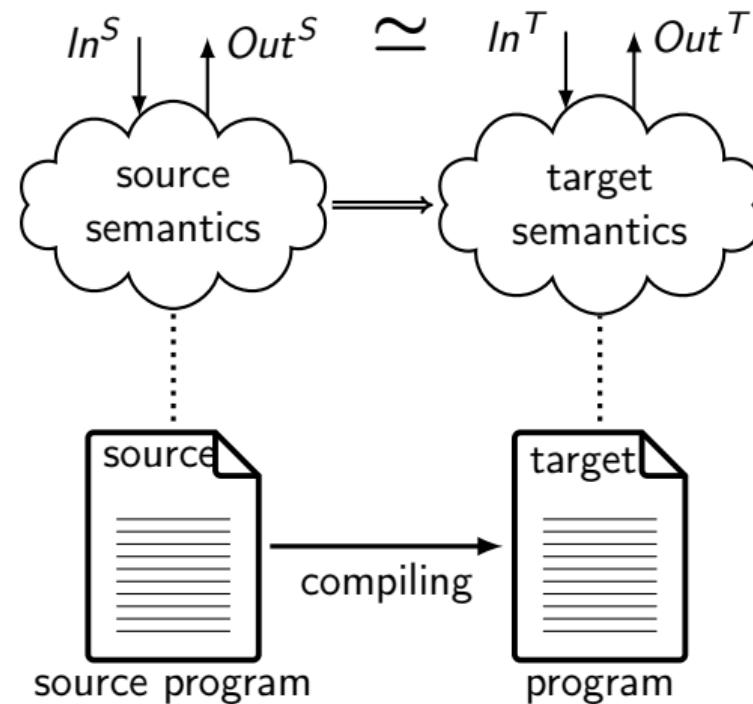
- CompCert: C → machine code
[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]
- CakeML: SML → machine code
[Kumar, Myreen, Norrish, and Owens (2014): CakeML: A Verified Implementation of ML]



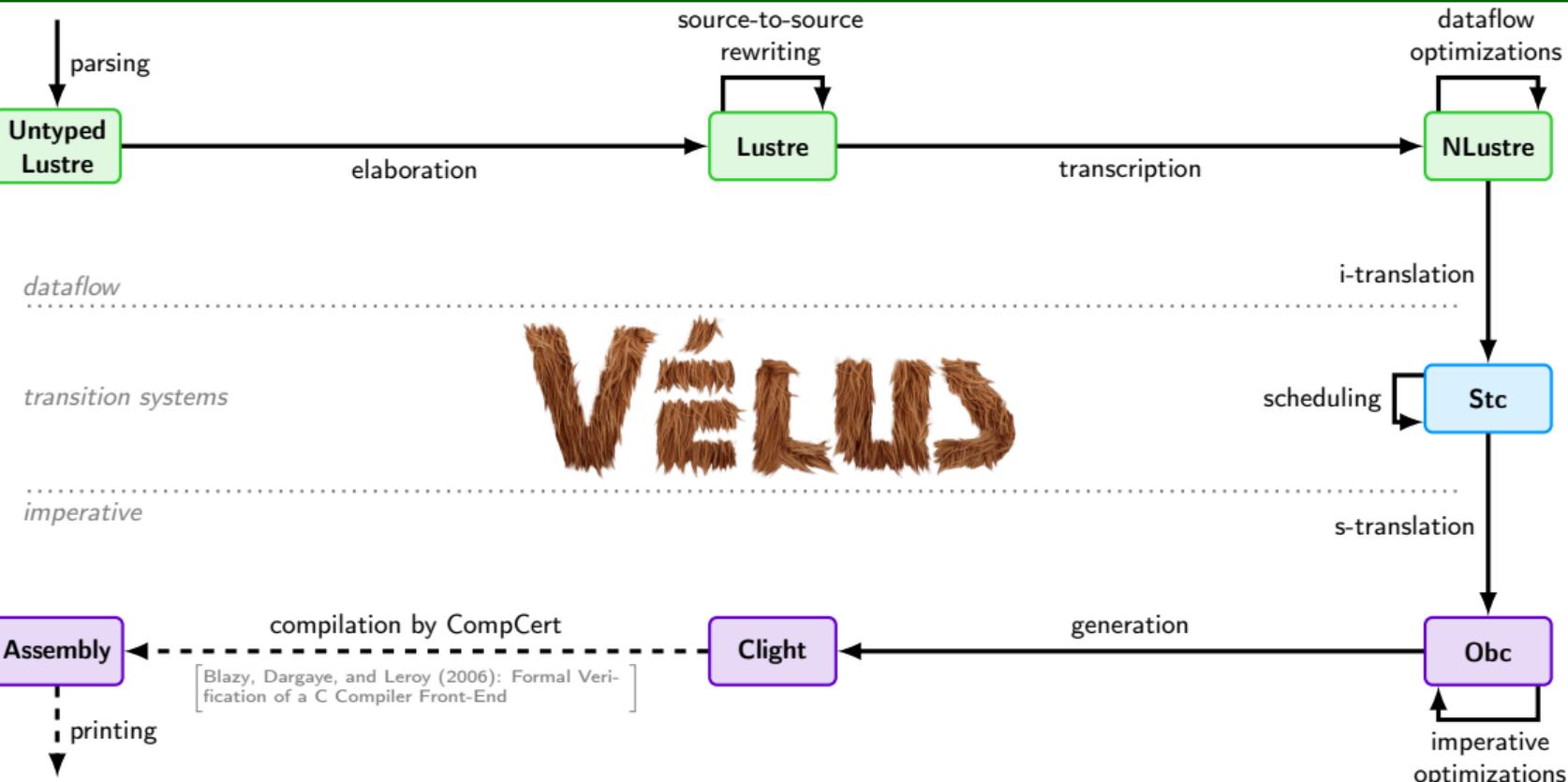
Compiler verification

In an Interactive Theorem Prover (recently):

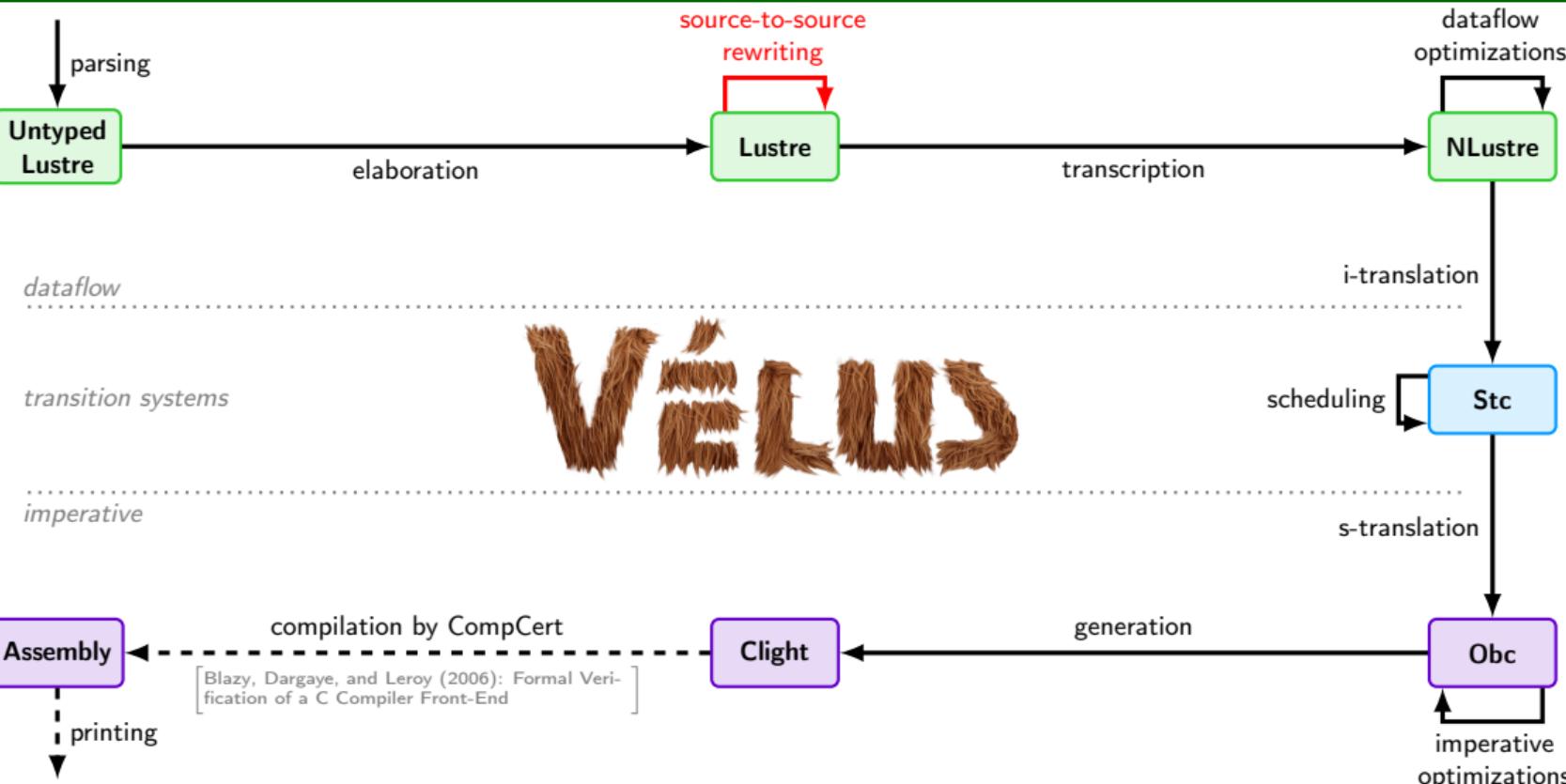
- CompCert: C → machine code
[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]
- CakeML: SML → machine code
[Kumar, Myreen, Norrish, and Owens (2014): CakeML: A Verified Implementation of ML]
- Vélus: Lustre/Scade 6 → C



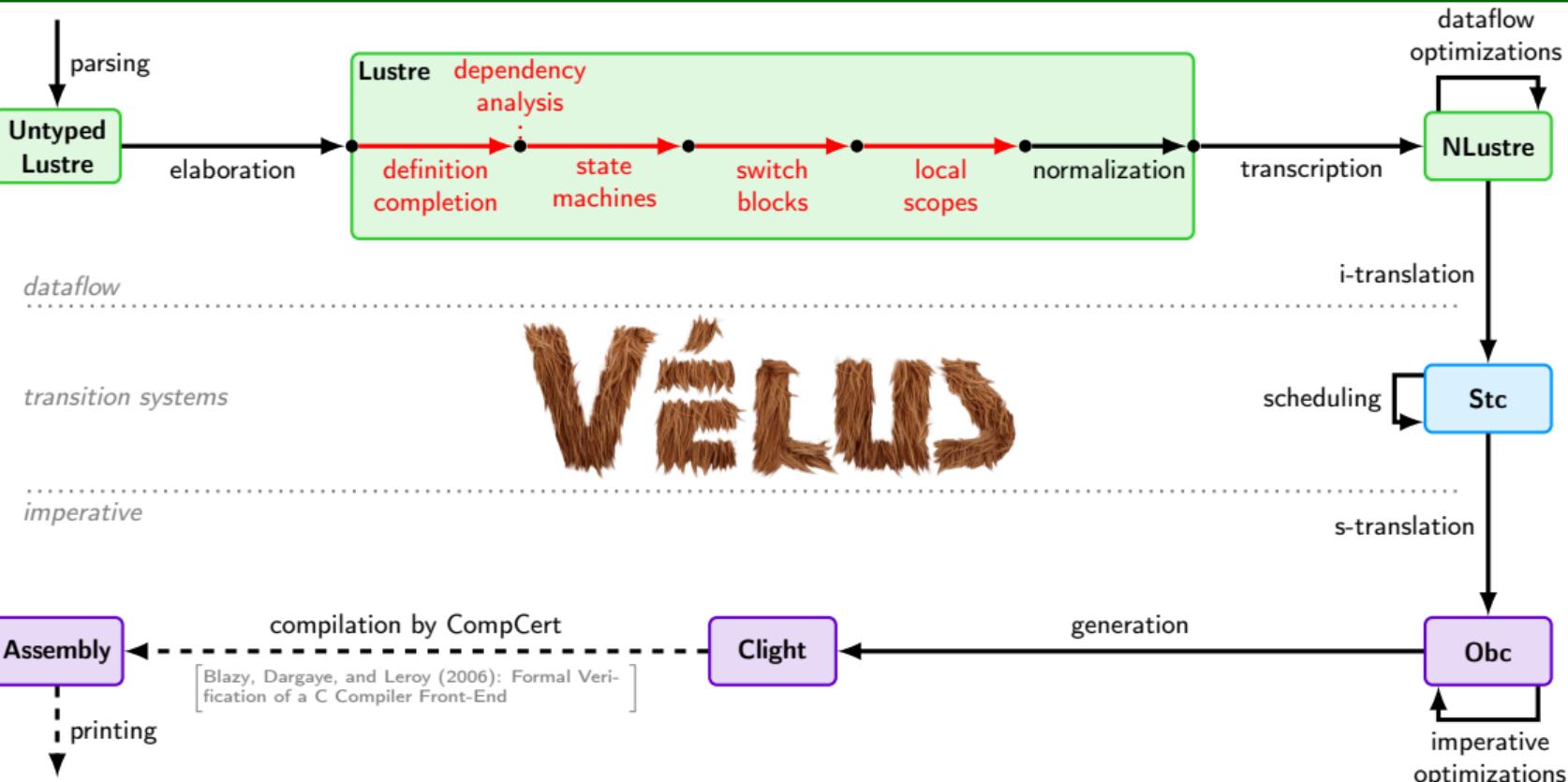
The Vélus Compiler



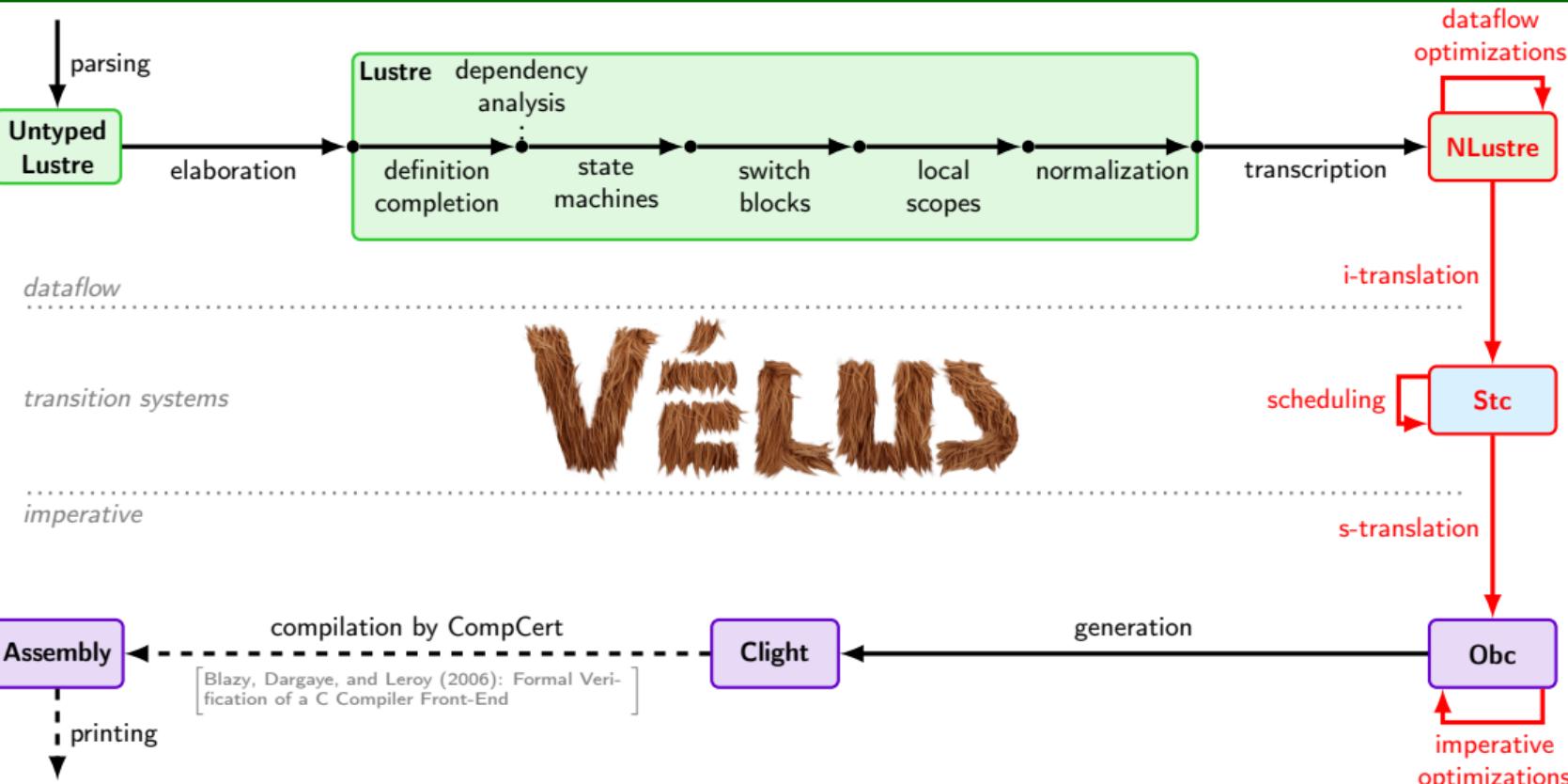
The Vélus Compiler



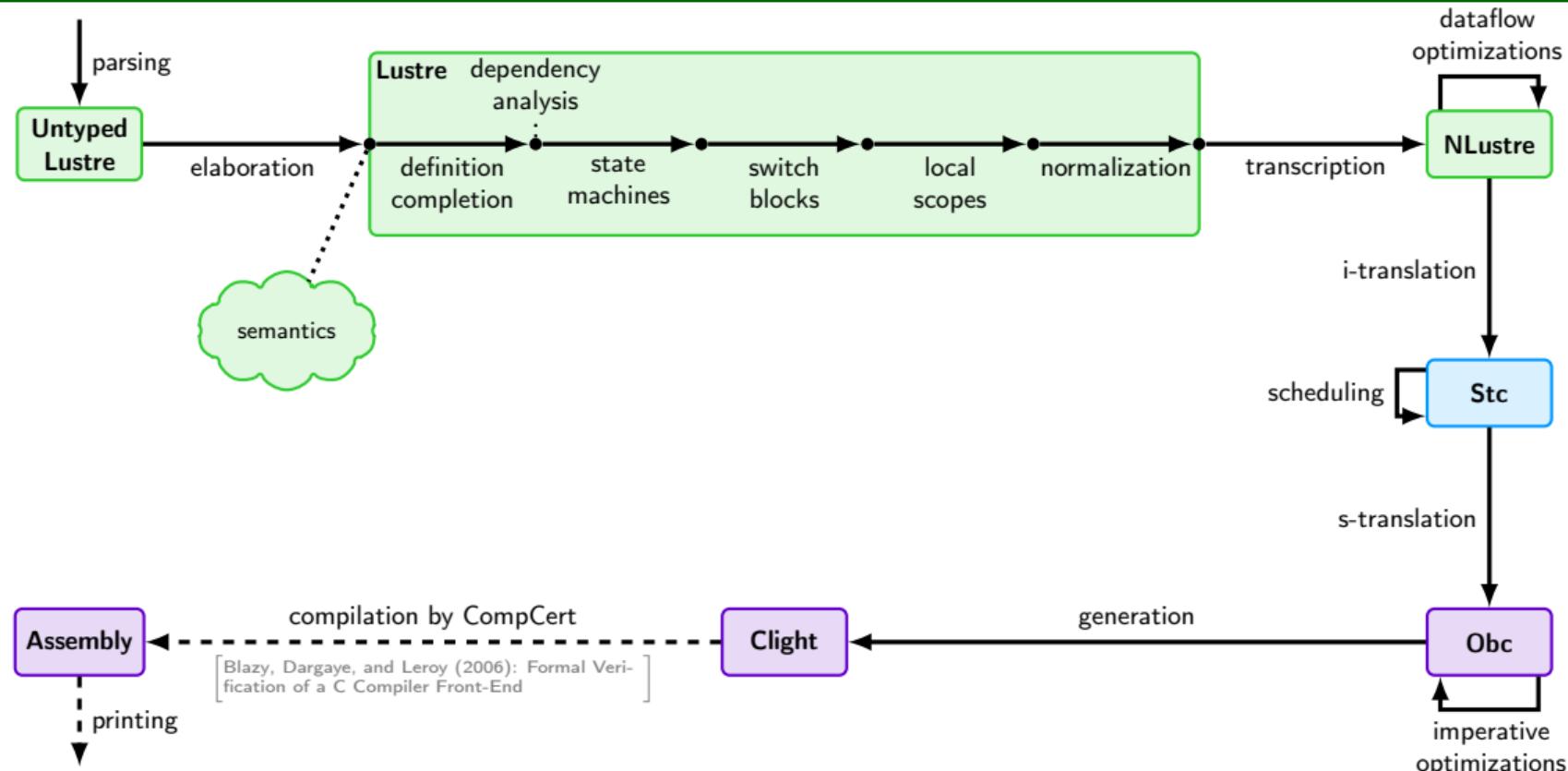
The Vélus Compiler



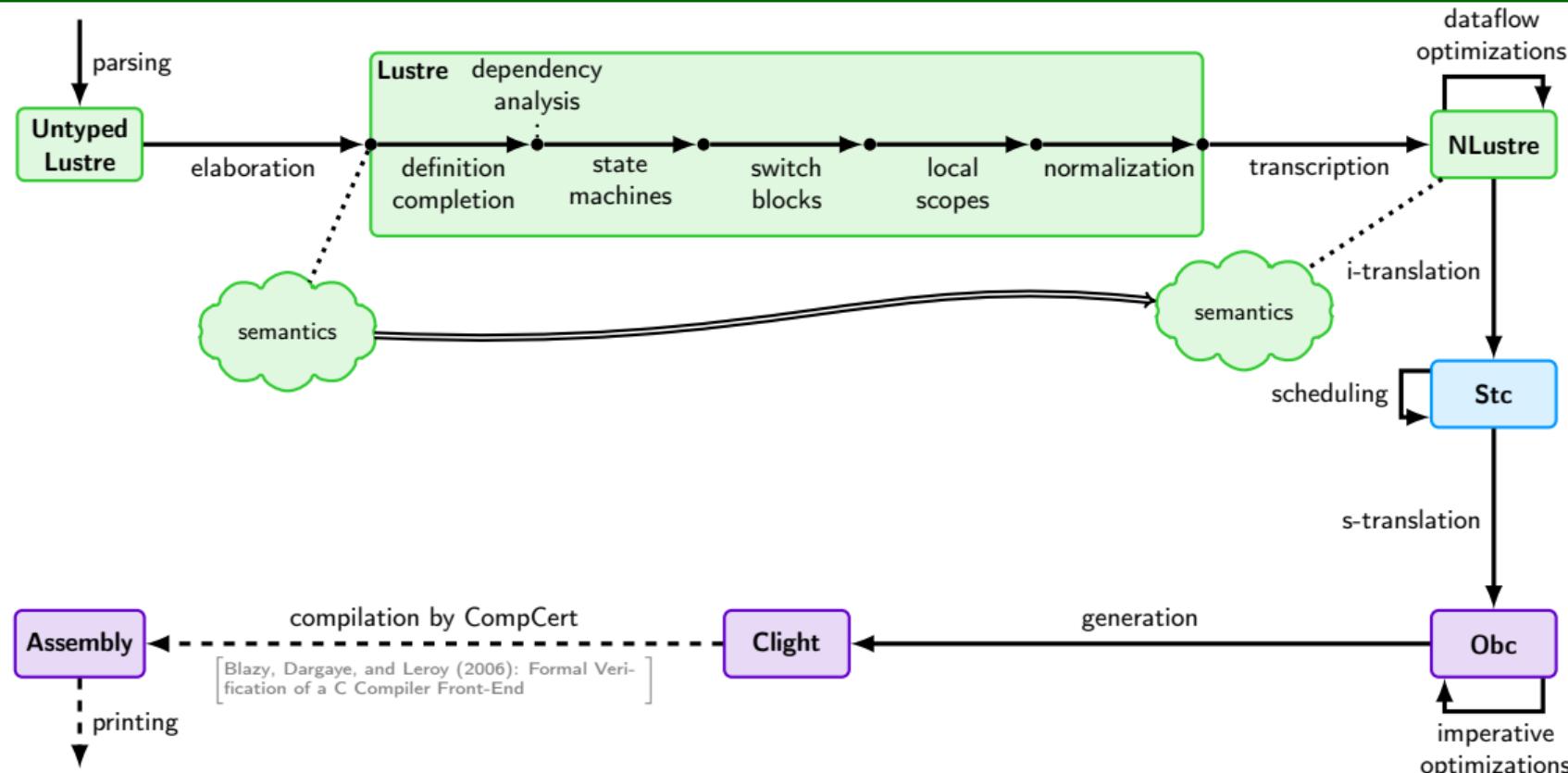
The Vélus Compiler



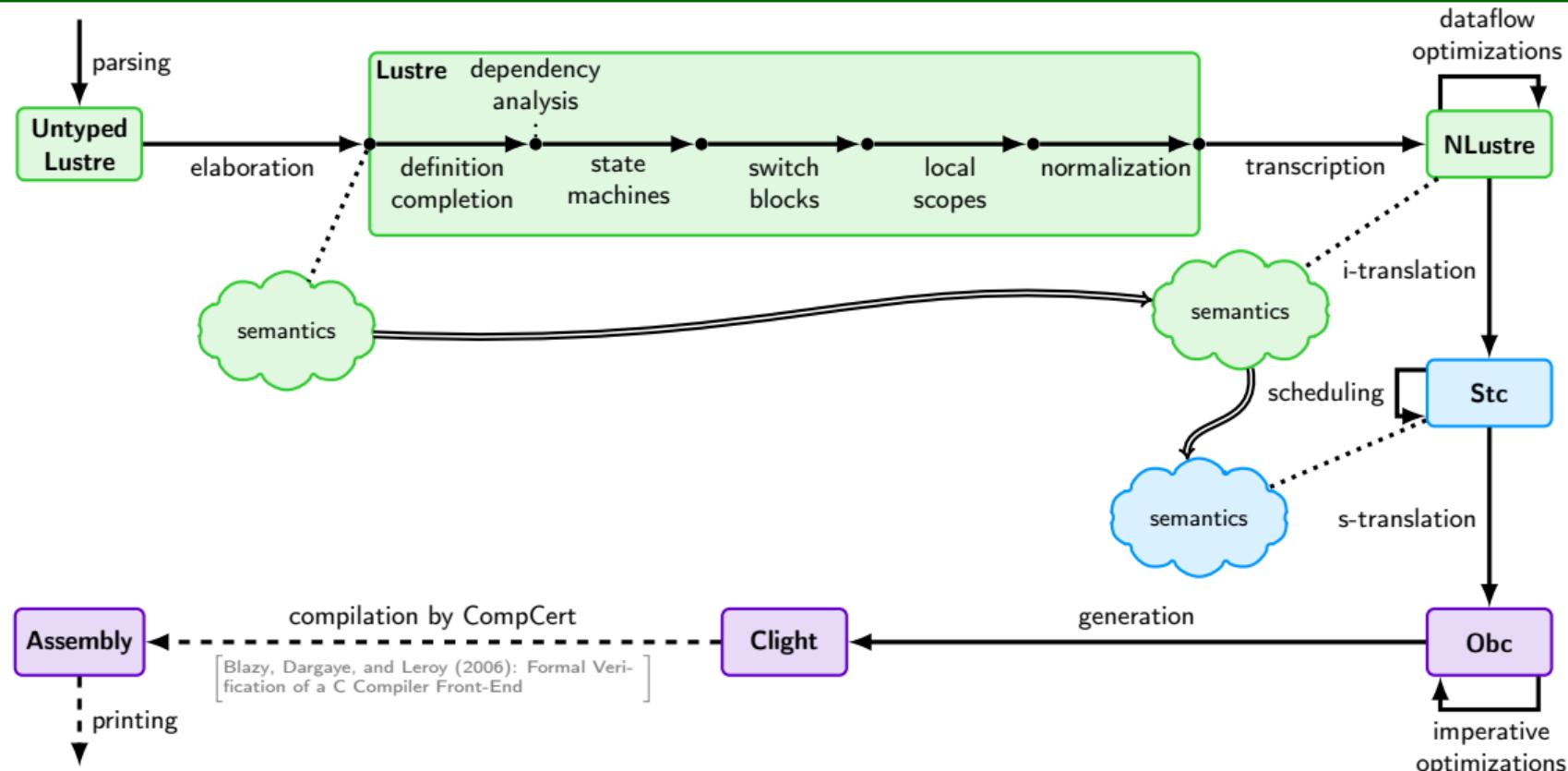
The Vélus Compiler



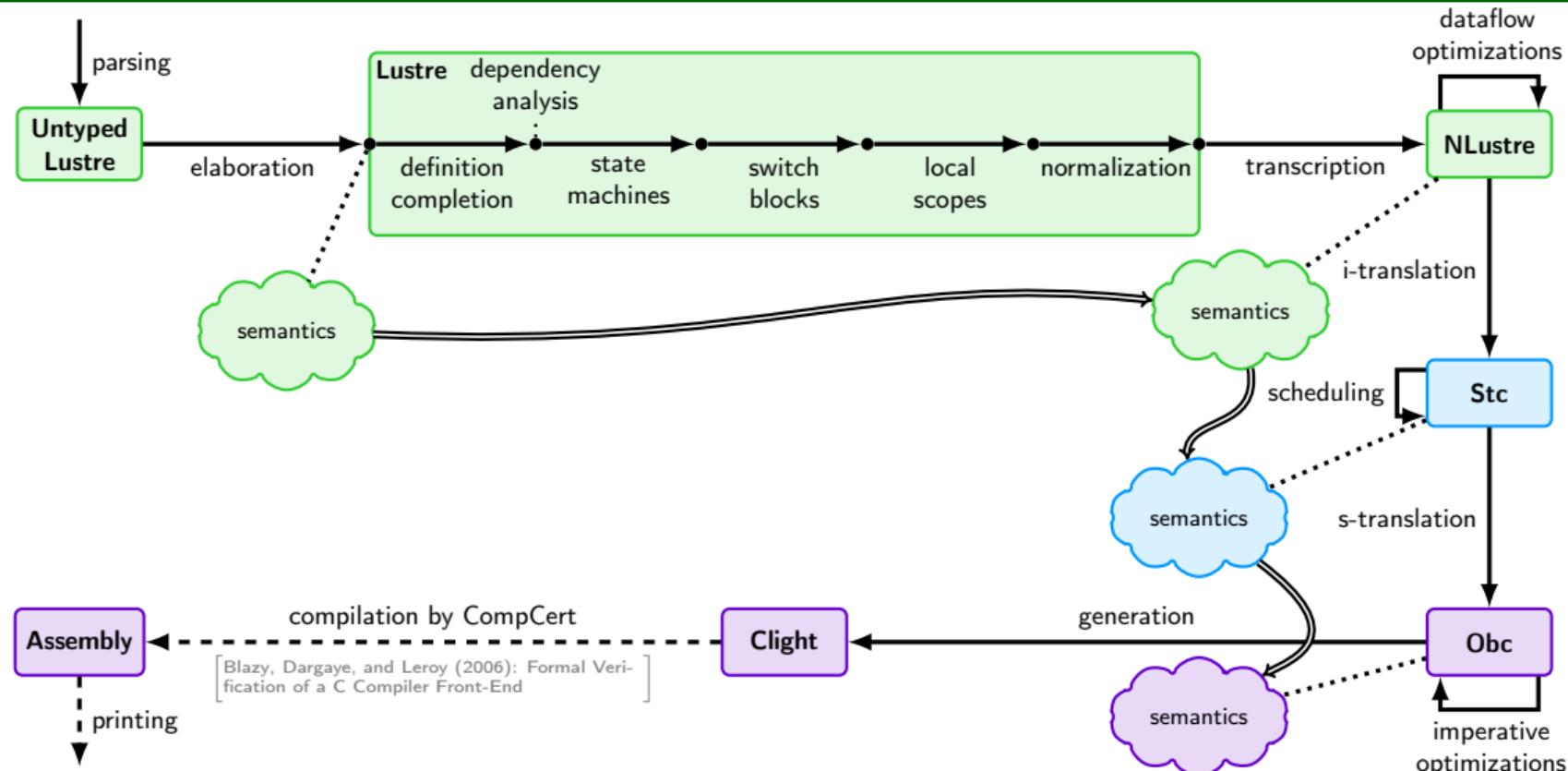
The Vélus Compiler



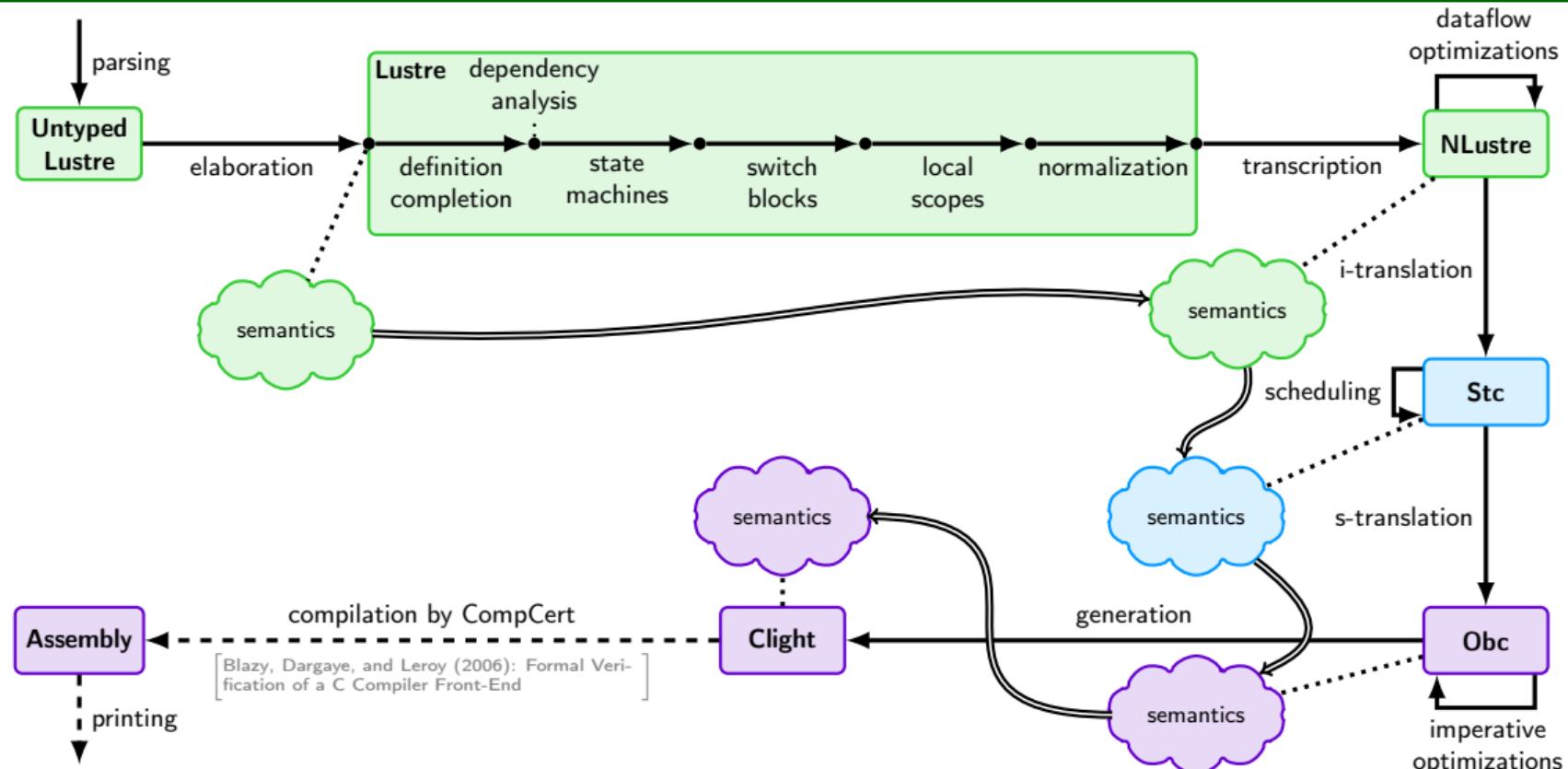
The Vélus Compiler



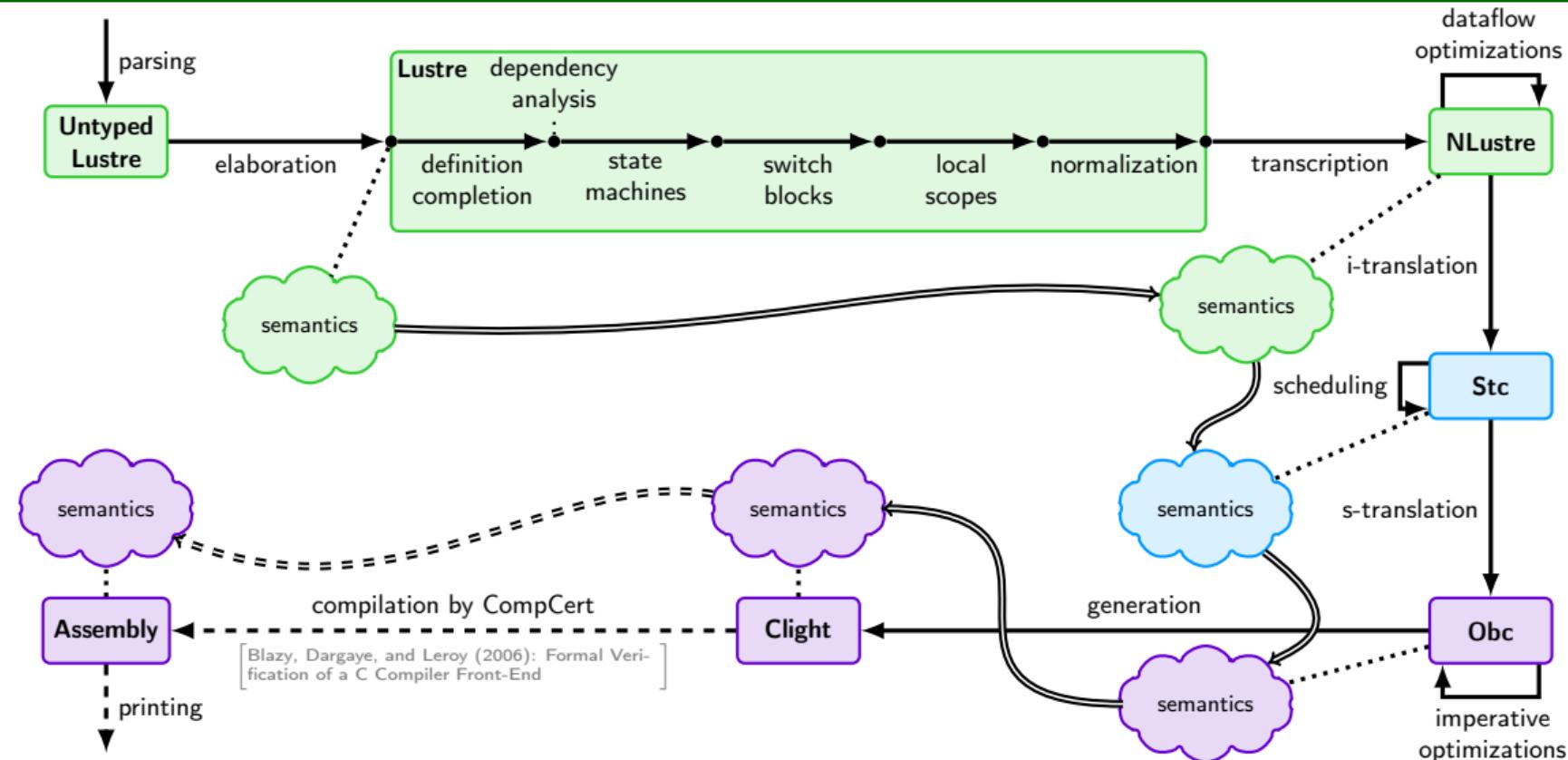
The Vélus Compiler



The Vélus Compiler



The Vélus Compiler



The Coq Interactive Theorem Prover



[Coq Development Team (2020): The Coq proof
assistant reference manual]

- A functional programming language
- ‘Extraction’ to OCaml programs

```
1 Inductive N :=
2 | 0 : N
3 | S : N → N.
4
5 Fixpoint plus n m :=
6   match n with
7   | 0 => m
8   | S n => S (plus n m)
9 end.
10
11 Fact plus_n_0 : ∀ n,
12   plus n 0 = n.
13 Proof.
14   induction n; simpl.
15    reflexivity.
16    now rewrite IHn.
17 Qed.
18
19 Fact plus_n_S : ∀ n m,
20   plus n (S m) = S (plus n m).
21 Proof.
22   induction n; intros; simpl.
23    reflexivity.
24    now rewrite IHn.
25 Qed.
26
27 Lemma plus_comm : ∀ n m,
28   plus n m = plus m n.
29 Proof.
30   induction n; intros.
31    now rewrite plus_n_0.
32    rewrite plus_n_S; simpl.
33    now rewrite IHn.
34 Qed.
```

1 goal (ID 29)

```
- n : N
- IHn : ∀ m : N,
  plus n m = plus m n
- m : N
```

```
plus (S n) m = plus m (S n)
```

● 151 🔒 *goals* 9:0 All

● 550 nat.v 19:3 All Coq ● 0 🔒 *response* 1:0 All

The Coq Interactive Theorem Prover



[Coq Development Team (2020): The Coq proof
assistant reference manual]

- A functional programming language
- ‘Extraction’ to OCaml programs
- A specification language

```
1 Inductive N :=
2 | 0 : N
3 | S : N → N.
4
5 Fixpoint plus n m :=
6   match n with
7   | 0 => m
8   | S n => S (plus n m)
9 end.
10
11 Fact plus_n_0 : ∀ n,
12   plus n 0 = n.
13 Proof.
14   induction n; simpl.
15    reflexivity.
16    now rewrite IHn.
17 Qed.
18
19 Fact plus_n_S : ∀ n m,
20   plus n (S m) = S (plus n m).
21 Proof.
22   induction n; intros; simpl.
23    reflexivity.
24    now rewrite IHn.
25 Qed.
26
27 Lemma plus_comm : ∀ n m,
28   plus n m = plus m n.
29 Proof.
30   induction n; intros.
31    now rewrite plus_n_0.
32    rewrite plus_n_S; simpl.
33    now rewrite IHn.
34 Qed.
```

1 goal (ID 29)

```
- n : N
- IHn : ∀ m : N,
  plus n m = plus m n
- m : N
```

```
plus (S n) m = plus m (S n)
```

● 151 🔒 *goals* 9:0 All

● 550 nat.v 19:3 All Coq ● 0 🔒 *response* 1:0 All

The Coq Interactive Theorem Prover



[Coq Development Team (2020): The Coq proof
assistant reference manual]

- A functional programming language
- ‘Extraction’ to OCaml programs
- A specification language
- Tactic-based interactive proof

The screenshot shows the Coq proof assistant interface. On the left, a tactic script is displayed:

```
1 Inductive N :=  
2 | 0 : N  
3 | S : N → N.  
4  
5 Fixpoint plus n m :=  
6   match n with  
7   | 0 => m  
8   | S n => S (plus n m)  
9 end.  
10  
11 Fact plus_n_0 : ∀ n,  
12   plus n 0 = n.  
13 Proof.  
14   induction n; simpl.  
15    $\vdash$  reflexivity.  
16    $\vdash$  now rewrite IHn.  
17 Qed.  
18  
19 Fact plus_n_S : ∀ n m,  
20   plus n (S m) = S (plus n m).  
21 Proof.  
22   induction n; intros; simpl.  
23    $\vdash$  reflexivity.  
24    $\vdash$  now rewrite IHn.  
25 Qed.  
26  
27 Lemma plus_comm : ∀ n m,  
28   plus n m = plus m n.  
29 Proof.  
30   induction n; intros.  
31    $\vdash$  now rewrite plus_n_0.  
32    $\vdash$  rewrite plus_n_S; simpl.  
33    $\vdash$  now rewrite IHn.  
34 Qed.
```

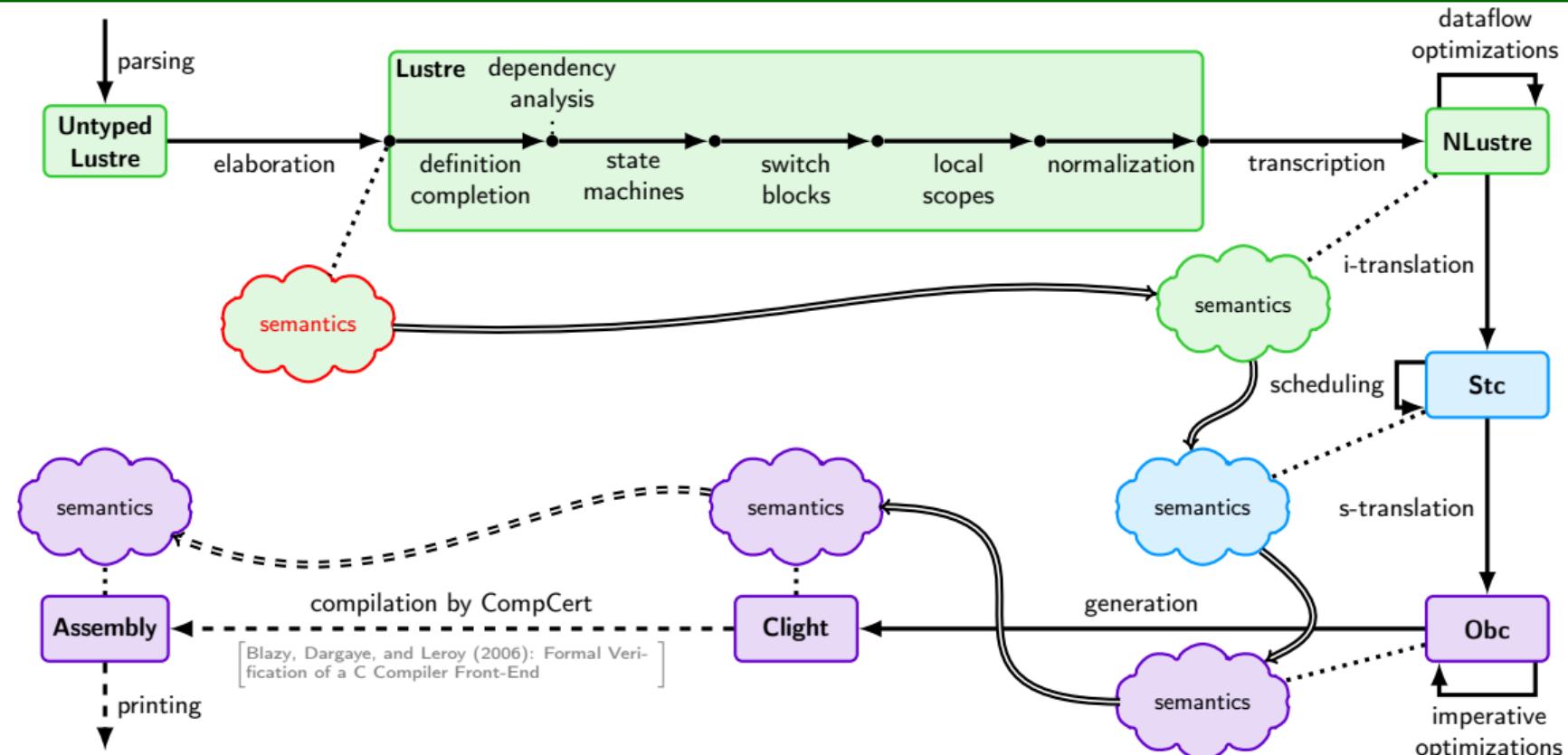
The tactic script uses various tactics like `induction`, `simpl`, `reflexivity`, and `now rewrite`. The last three lines of the script (31-34) are highlighted with a red box. At the bottom of the interface, status bars show the number of goals (151), the current file (nat.v), and the Coq version (550).

On the right, the state of the proof is shown:

```
1 goal (ID 29)  
- n : N  
- IHn :  $\forall m : N$ ,  
  plus n m = plus m n  
- m : N  
_____  
plus (S n) m = plus m (S n)
```

The proof state shows the goal `plus (S n) m = plus m (S n)` with hypotheses `n : N`, `IHn`, and `m : N`.

Relational Semantics of Vélus



Dataflow relational semantics

$$\frac{G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \\ \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H, (\text{base-of } (xss_1, \dots, xss_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

inc	5	4	1	3	2	8	3	...
o	5	9	10	13	15	23	26	...

Dataflow relational semantics

$$\frac{\begin{array}{c} G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \\ \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H, (\text{base-of } (xs_1, \dots, xs_n)) \vdash blk \end{array}}{G \vdash f(xss) \Downarrow yss}$$

$$\frac{\begin{array}{c} \forall i, H(xs_i) \equiv vs_i \quad G, H, bs \vdash es \Downarrow [vs_i]^i \end{array}}{G, H, bs \vdash xs = es}$$

Equations

If the clock is true, the right-hand expression is evaluated and its value is associated with the variable on the left-hand side.

$$\frac{\sigma(ck) = tt, v \vdash \exp \xrightarrow{k} \exp', \sigma(id) = k}{id = (ck) \exp \xrightarrow{k} id = (ck) \exp'}$$

If the clock is not true, the left-hand variable is not evaluated.

$$\frac{\sigma(ck) \neq tt, \sigma(id) = \perp}{id = (ck) \exp \xrightarrow{k} id = (ck) \exp}$$

These rules define σ to be the solution of a fixpoint equation. Moreover, this solution must be unique (otherwise the program contains a deadlock; this problem will be detailed in section 4.1).

inc	5	4	1	3	2	8	3	...
o	5	9	10	13	15	23	26	...

Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems

Dataflow relational semantics – in Coq

Inductive sem_exp:

[...]

with sem_equation:

| Seq:

Forall2 (sem_exp G H bs) es ss →
 Forall2 (sem_var H) xs (concat ss) →
 sem_equation G H bs (xs, es)

[...]

$$\frac{\forall i, H(xs_i) \equiv vs_i \quad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash xs = es}$$

with sem_node:

| Snode:

find_node f G = Some n →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →

let bs := clocks_of ss in

sem_block H bs n.(n_block) →

sem_node f ss os.

$G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk}$

$$\frac{\forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H, (\text{base-of } (xs_1, \dots, xs_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

Dataflow relational semantics – in Coq

Inductive sem_exp:

[...]

with sem_equation:

| Seq:

Forall2 (sem_exp G H bs) es ss →

Forall2 (sem_var H) xs (concat ss) →

sem_equation G H bs (xs, es)

[...]

$$\frac{\forall i, H(xs_i) \equiv vs_i \quad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash xs = es}$$

with sem_node:

| Snode:

find_node f G = Some n →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →

let bs := clocks_of ss in

sem_block H bs n.(n_block) →

sem_node f ss os.

$G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk}$

$$\frac{\forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H, (\text{base-of } (xs_1, \dots, xs_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

Dataflow relational semantics – in Coq

Inductive sem_exp:

[...]

with sem_equation:

| Seq:

Forall2 (sem_exp G H bs) es ss →

Forall2 (sem_var H) xs (concat ss) →

sem_equation G H bs (xs, es)

[...]

$$\frac{\forall i, H(xs_i) \equiv vs_i}{G, H, bs \vdash es \Downarrow [vs_i]^i}$$

$$G, H, bs \vdash xs = es$$

with sem_node:

| Snode:

find_node f G = Some n →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →

let bs := clocks_of ss in

sem_block H bs n.(n_block) →

sem_node f ss os.

$$G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk}$$

$$\frac{\forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H, (\text{base-of } (xs_1, \dots, xs_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

Dataflow relational semantics – in Coq

Inductive sem_exp:

[...]

with sem_equation:

| Seq:

Forall2 (sem_exp G H bs) es ss →
 Forall2 (sem_var H) xs (concat ss) →
 sem_equation G H bs (xs, es)

[...]

$$\frac{\forall i, H(xs_i) \equiv vs_i \quad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash xs = es}$$

with sem_node:

| Snode:

find_node f G = Some n →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →

Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →

let bs := clocks_of ss in

sem_block H bs n.(n_block) →

sem_node f ss os.

$G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk}$

$$\frac{\forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H, (\text{base-of } (xs_1, \dots, xs_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

fby operator semantics

inc	⟨⟩ ⟨⟩ 5 ⟨⟩ ⟨⟩ 4 1 3 2 ⟨⟩ 8 3 ...
0 fby o	⟨⟩ ⟨⟩ 0
o = (0 fby o) + inc	⟨⟩ ⟨⟩ 5

```

node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
  
```

$$\begin{aligned}
 \text{fby } (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) &\equiv \langle \rangle \cdot \text{fby } xs \text{ } ys \\
 \text{fby } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) &\equiv \langle v_1 \rangle \cdot \text{fby } v_2 \text{ } xs \text{ } ys
 \end{aligned}$$

fby operator semantics

inc	$\langle \rangle$	$\langle \rangle$	5	$\langle \rangle$	$\langle \rangle$	4	1	3	2	$\langle \rangle$	8	3	...
0 fby o	$\langle \rangle$	$\langle \rangle$	0	$\langle \rangle$	$\langle \rangle$	5	9	10	13	$\langle \rangle$	15	23	...
$o = (0 \text{ fby } o) + \text{inc}$	$\langle \rangle$	$\langle \rangle$	5	$\langle \rangle$	$\langle \rangle$	9	10	13	15	$\langle \rangle$	23	26	...

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

$$\begin{aligned}
 & \text{fby} (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) \equiv \langle \rangle \cdot \text{fby} xs ys \\
 & \text{fby} (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \equiv \langle v_1 \rangle \cdot \text{fby1 } v_2 xs ys \\
 & \text{fby1 } v_0 (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) \equiv \langle \rangle \cdot \text{fby1 } v_0 xs ys \\
 & \text{fby1 } v_0 (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \equiv \langle v_0 \rangle \cdot \text{fby1 } v_2 xs ys
 \end{aligned}$$

fby operator semantics

inc	⟨⟩ ⟨⟩ 5 ⟨⟩ ⟨⟩ 4 1 3 2 ⟨⟩ 8 3 ...
0 fby o	⟨⟩ ⟨⟩ 0 ⟨⟩ ⟨⟩ 5 9 10 13 ⟨⟩ 15 23 ...
o = (0 fby o) + inc	⟨⟩ ⟨⟩ 5 ⟨⟩ ⟨⟩ 9 10 13 15 ⟨⟩ 23 26 ...

```

node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel

```

$$\begin{aligned}
 & \text{fby } (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) \equiv \langle \rangle \cdot \text{fby } xs \text{ } ys \\
 & \text{fby } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \equiv \langle v_1 \rangle \cdot \text{fby1 } v_2 \text{ } xs \text{ } ys \\
 & \text{fby1 } v_0 \text{ } (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) \equiv \langle \rangle \cdot \text{fby1 } v_0 \text{ } xs \text{ } ys \\
 & \text{fby1 } v_0 \text{ } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \equiv \langle v_0 \rangle \cdot \text{fby1 } v_2 \text{ } xs \text{ } ys
 \end{aligned}$$

$$\frac{\begin{array}{c} G, H, bs \vdash es_0 \Downarrow [xs_i]^i \quad G, H, bs \vdash es_1 \Downarrow [ys_i]^i \\ \forall i, \text{fby } xs_i \text{ } ys_i \equiv vs_i \end{array}}{G, H, bs \vdash es_0 \text{ fby } es_1 \Downarrow [vs_i]^i}$$

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
    switch step
    | true do
        mA = not (last mB);
        mB = last mA;
    | false do (mA, mB) = (last mA, last mB)
    end;
    last mA = true;
    last mB = false;
tel
```

step	...
last mA	...
last mB	...
mA	...
mB	...

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
```

$$\begin{aligned} \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs \\ \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) &\equiv \langle v \rangle \cdot \text{when}^C xs cs \\ \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs \end{aligned}$$

step		...
last mA		...
last mB		...
mA		...
mB		...

Stream semantics of switch blocks

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel

```

$$\begin{aligned}
 \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) &\equiv \langle v \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs
 \end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i} (H, bs) cs \vdash blks_i}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}}$$

step		...
last mA		...
last mB		...
mA		...
mB		...

Stream semantics of switch blocks

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel

```

$$\begin{aligned}
 \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) &\equiv \langle v \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs
 \end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i} (H, bs) cs \vdash blks_i}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}}$$

step	T	T	T	T	T	T	T	...
last mA	T	T	F	F	T	T	F	...
last mB	F	T	T	F	F	T	T	...
mA	T	F	F	T	T	F	F	...
mB	T	T	F	F	T	T	F	...

Stream semantics of switch blocks

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel

```

$$\begin{aligned}
 & \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) \equiv \langle \rangle \cdot \text{when}^C xs cs \\
 & \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) \equiv \langle v \rangle \cdot \text{when}^C xs cs \\
 & \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) \equiv \langle \rangle \cdot \text{when}^C xs cs
 \end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i} (H, bs) cs \vdash blks_i}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}}$$

step	F	F	F	F	F	F	F	F	...
last mA	T	F	F	F	T	T	F	F	...
last mB	F	T	T	F	F	T	T	T	...
mA	T	F	F	F	T	T	F	F	...
mB	F	T	T	F	F	T	T	T	...

Stream semantics of switch blocks

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
  
```

$$\begin{aligned}
 \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) &\equiv \langle v \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs
 \end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i}(H, bs) cs \vdash blks_i}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}}$$

step	F	T	T	F	F	T	F	T	F	T	F	T	F	F	T	...
last mA	T	T	T	F	F	F	F	T	T	T	F	F	F	F	...	
last mB	F	F	T	T	T	T	F	F	F	T	T	T	T	T	...	
mA	T	T	F	F	F	F	T	T	T	F	F	F	F	F	...	
mB	F	T	T	T	T	F	F	F	T	T	T	T	T	F	...	

Stream semantics of reset blocks and state machines

$$\text{mask}_{k'}^k (F \cdot rs) (sv \cdot xs) \equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'}^k rs xs$$

$$\text{mask}_{k'}^k (T \cdot rs) (sv \cdot xs) \equiv (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'+1}^k rs xs$$

[Bourke, Brun, and Pouzet (2020): Mechanized Semantics and Verified
Compilation for a Dataflow Synchronous Language with Reset]

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv rs \\ \forall k, \; G, \text{mask}^k rs (H, bs) \vdash blks \end{array}}{G, H, bs \vdash \text{reset } blks \text{ every } e}$$

- `reset` block \mapsto `mask` operator

Stream semantics of reset blocks and state machines

$\text{mask}_{k'}^k (F \cdot rs) (sv \cdot xs) \equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'}^k rs xs$

$\text{mask}_{k'}^k (T \cdot rs) (sv \cdot xs) \equiv (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'+1}^k rs xs$

[Bourke, Brun, and Pouzet (2020): Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset]

p49

$$\begin{aligned} G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv rs \\ \forall k, G, \text{mask}^k rs (H, bs) \vdash blks \end{aligned}$$

G, H, bs \vdash **reset** blks **every** e

- **reset** block \mapsto **mask** operator
- state machines \mapsto **select** operator

2.9. Semantics of State Machines
$\text{select}_{C,k}^{C,k} (\langle \rangle \cdot sts) (v_1 \cdot xs) \triangleq \langle \rangle \cdot \text{select}_{C,k}^{C,k} sts xs$
$\text{select}_{C,k}^{C,k} ((C, F) \cdot sts) (v_1 \cdot xs) \triangleq (\text{if } k' = k \text{ then } v_1 \text{ else } \langle \rangle) \cdot \text{select}_{C,k}^{C,k} sts xs$
$\text{select}_{C,k}^{C,k} ((C, T) \cdot sts) (v_1 \cdot xs) \triangleq (\text{if } k' + 1 = k \text{ then } v_1 \text{ else } \langle \rangle) \cdot \text{select}_{C,k+1}^{C,k} sts xs$
$\text{select}_{C,k}^{C,k} ((C', b) \cdot sts) (v_1 \cdot xs) \triangleq \langle \rangle \cdot \text{select}_{C,k}^{C,k} sts xs$
(a) select CoinStream v.323
$\forall x, x \in \text{dom}(H') \iff x \in locs$
$G, H + H', bs \vdash blks \quad G, H + H', bs, C_i \vdash trans \Downarrow sts$
$G, H, bs \vdash \text{var } locs \text{ do } blks \text{ until } trans \Downarrow sts$
$H, bs \vdash ck \Downarrow bs' \quad G, H, bs' \vdash \text{autinit} \emptyset sts_0 \quad \text{fly } sts_0 sts_1 \equiv sts$
$\forall i, \forall k, G, (\text{select}_{C,k}^{C,k} sts (H, bs)), C_i \vdash \text{autscope}_i \Downarrow (\text{select}_{C,i}^{C,k} sts sts_0)$
$G, H, bs \vdash \text{automation initially autinit}^{ck} [\text{state } C_i \text{ autscope}_i]^i \text{ end}$
(b) AutoWeak Lustre/LSemantics v.306
$H, bs \vdash ck \Downarrow bs' \quad \text{fly } (\text{const } bs' (C, F)) sts_1 \equiv sts$
$\forall i, \forall k, G, (\text{select}_{C,i}^{C,k} sts (H, bs)), C_i \vdash \text{trans}_i \Downarrow (\text{select}_{C,i}^{C,k} sts sts_1)$
$\forall i, \forall k, G, (\text{select}_{C,i}^{C,k} sts_1 (H, bs)) \vdash blks_i$
$G, H, bs \vdash \text{automation initially autinit}^{ck} [\text{state } C_i \text{ do } blks_i \text{ unless trans}_i]^i \text{ end}$
(c) AutoStrong Lustre/LSemantics v.328
$G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs'$
$G, H, bs \vdash G, H, bs, C_i \vdash trans \Downarrow sts$
$G, H, bs \vdash \text{first-of}_i^{C,k} bs' \Downarrow sts$
$G, H, bs \vdash C \text{ if } e; \text{autinit } \emptyset sts' \Downarrow$
$sts = \text{const } bs (C, F)$
$G, H, bs \vdash \text{otherwise } C \Downarrow sts$
(d) Initial state

$G, H, bs, C_i \vdash \text{if } e \text{ then } C \text{ trans } \Downarrow sts'$
$G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs'$
$G, H, bs, C_i \vdash trans \Downarrow sts$
$G, H, bs \vdash \text{first-of}_i^{C,k} bs' \Downarrow sts$
$G, H, bs \vdash \text{if } e \text{ then } C \text{ trans } \Downarrow sts'$
$G, H, bs \vdash \text{sts} = \text{first-of}_i^{C,k} bs' \Downarrow$
$G, H, bs, C_i \vdash \text{if } e \text{ then } C \text{ trans } \Downarrow sts'$
$G, H, bs \vdash \text{sts} = \text{const } bs (C, F) \Downarrow$
$G, H, bs, C_i \vdash trans \Downarrow sts$
$G, H, bs \vdash \text{sts} = \text{first-of}_i^{C,k} bs' \Downarrow$
$G, H, bs \vdash \text{sts} = \text{const } bs (C, F) \Downarrow$
(e) sem_transitions Lustre/LSemantics v.261

Stream semantics of reset blocks and state machines

$\text{mask}_{k'}^k (F \cdot rs) (sv \cdot xs) \equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'}^k rs xs$

$\text{mask}_{k'}^k (T \cdot rs) (sv \cdot xs) \equiv (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'+1}^k rs xs$

[Bourke, Brun, and Pouzet (2020): Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset]

$$\begin{aligned} G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv rs \\ \forall k, G, \text{mask}^k rs (H, bs) \vdash blks \end{aligned}$$

G, H, bs $\vdash \text{reset blks every } e$

- **reset** block \mapsto **mask** operator
- state machines \mapsto **select** operator

p49



Proving semantic meta-properties

Prove properties of the semantic model:

- Determinism of the semantics:

if $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$ **then** $ys_1 \equiv ys_2$

Proving semantic meta-properties

Prove properties of the semantic model:

- Determinism of the semantics:

if $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$ **then** $ys_1 \equiv ys_2$

- Clock-system correctness:

if $\Gamma \vdash e : ck$ **and** $G, H, bs \vdash e \Downarrow vs$ **then** $H, bs \vdash ck \Downarrow (\text{clock-of } vs)$

Proving semantic meta-properties

Prove properties of the semantic model:

- Determinism of the semantics:

if $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$ **then** $ys_1 \equiv ys_2$

- Clock-system correctness:

if $\Gamma \vdash e : ck$ **and** $G, H, bs \vdash e \Downarrow vs$ **then** $H, bs \vdash ck \Downarrow (\text{clock-of } vs)$

Proof by induction on the syntax, inversion of the semantics:

- ...
- variable: inverting $G, H, bs \vdash x \Downarrow [vs]$ tells us $H(x) \equiv vs$. What now ?
- ...

Dependency Analysis

Consider a program with the following definitions:

- $x = x$; admits all value
- $x = x + 1$; admits no value

Dependency Analysis

Consider a program with the following definitions:

- $x = x$; admits all value
- $x = x + 1$; admits no value

Not possible to prove any property of the stream of x .

We can only reason on program without dependency cycle.

Dependency Analysis

Consider a program with the following definitions:

- $x = x$; admits all value
- $x = x + 1$; admits no value

Not possible to prove any property of the stream of x .

We can only reason on program without dependency cycle.

Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The]
synchronous dataflow programming language LUSTRE

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a])
Synchronous Stream Language

Dependency Analysis

Consider a program with the following definitions:

- $x = x$; admits all value
- $x = x + 1$; admits no value

Not possible to prove any property of the stream of x .

We can only reason on program without dependency cycle.

Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])
- extended to handle control blocks (using labels)

Dependency Analysis

Consider a program with the following definitions:

- $x = x$; admits all value
- $x = x + 1$; admits no value

Not possible to prove any property of the stream of x .

We can only reason on program without dependency cycle.

Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])
- extended to handle control blocks (using labels)
- verified graph analysis algorithm: produces a witness of acyclicity

Dependency Analysis

Consider a program with the following definitions:

- $x = x$; admits all value
- $x = x + 1$; admits no value

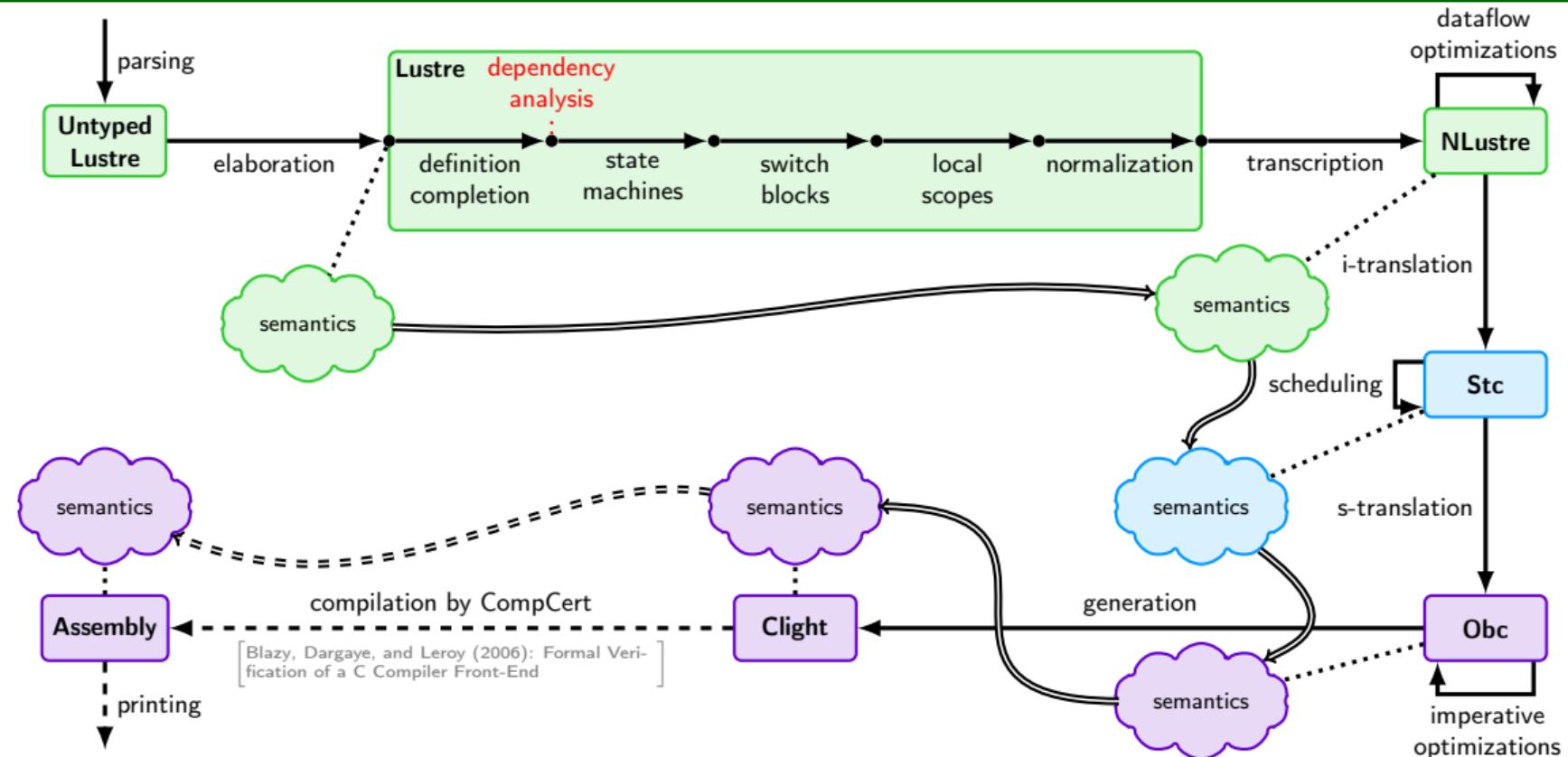
Not possible to prove any property of the stream of x .

We can only reason on program without dependency cycle.

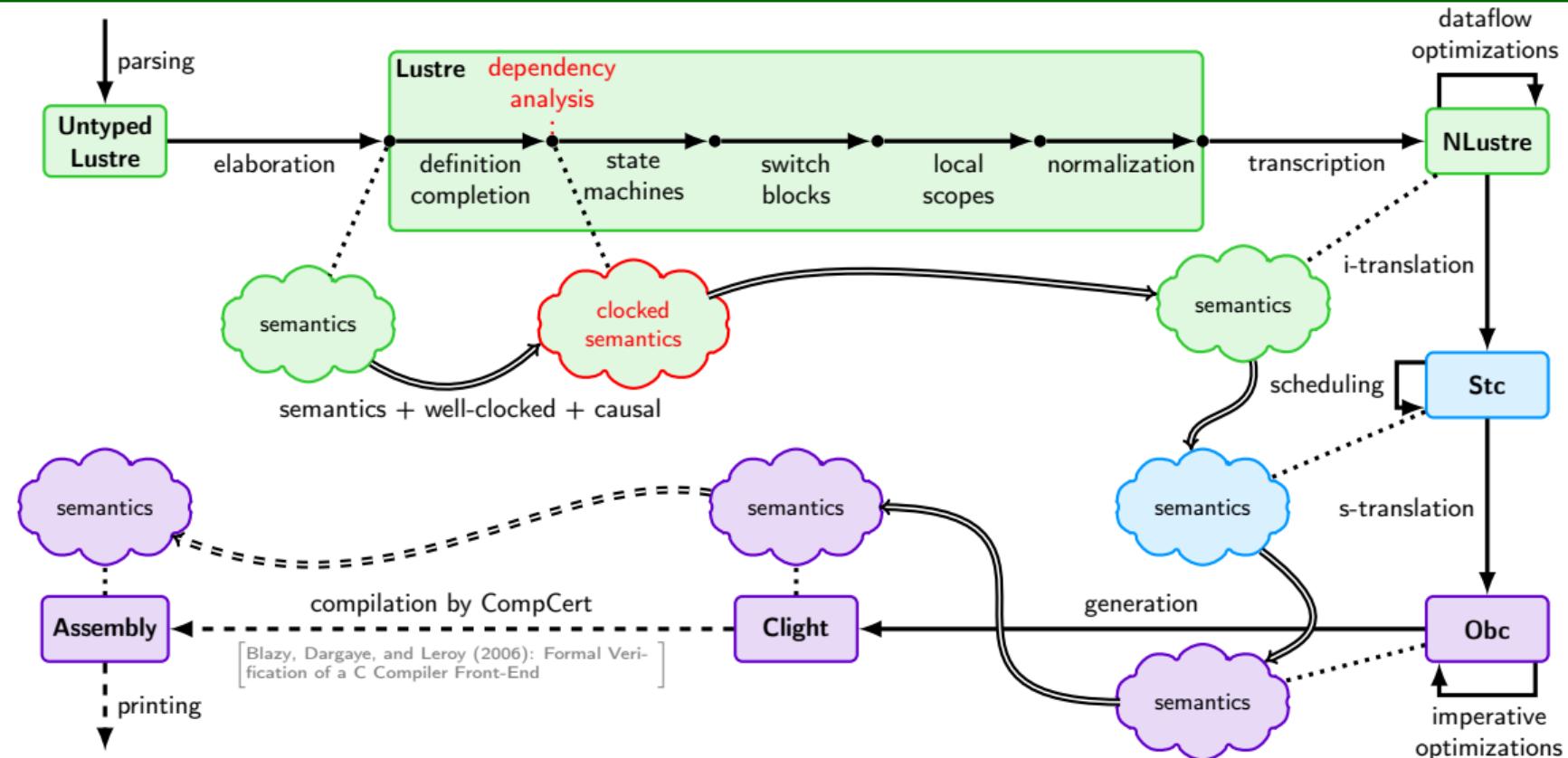
Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])
- extended to handle control blocks (using labels)
- verified graph analysis algorithm: produces a witness of acyclicity
- Used to prove properties of the semantics (clock-system correctness, determinism)

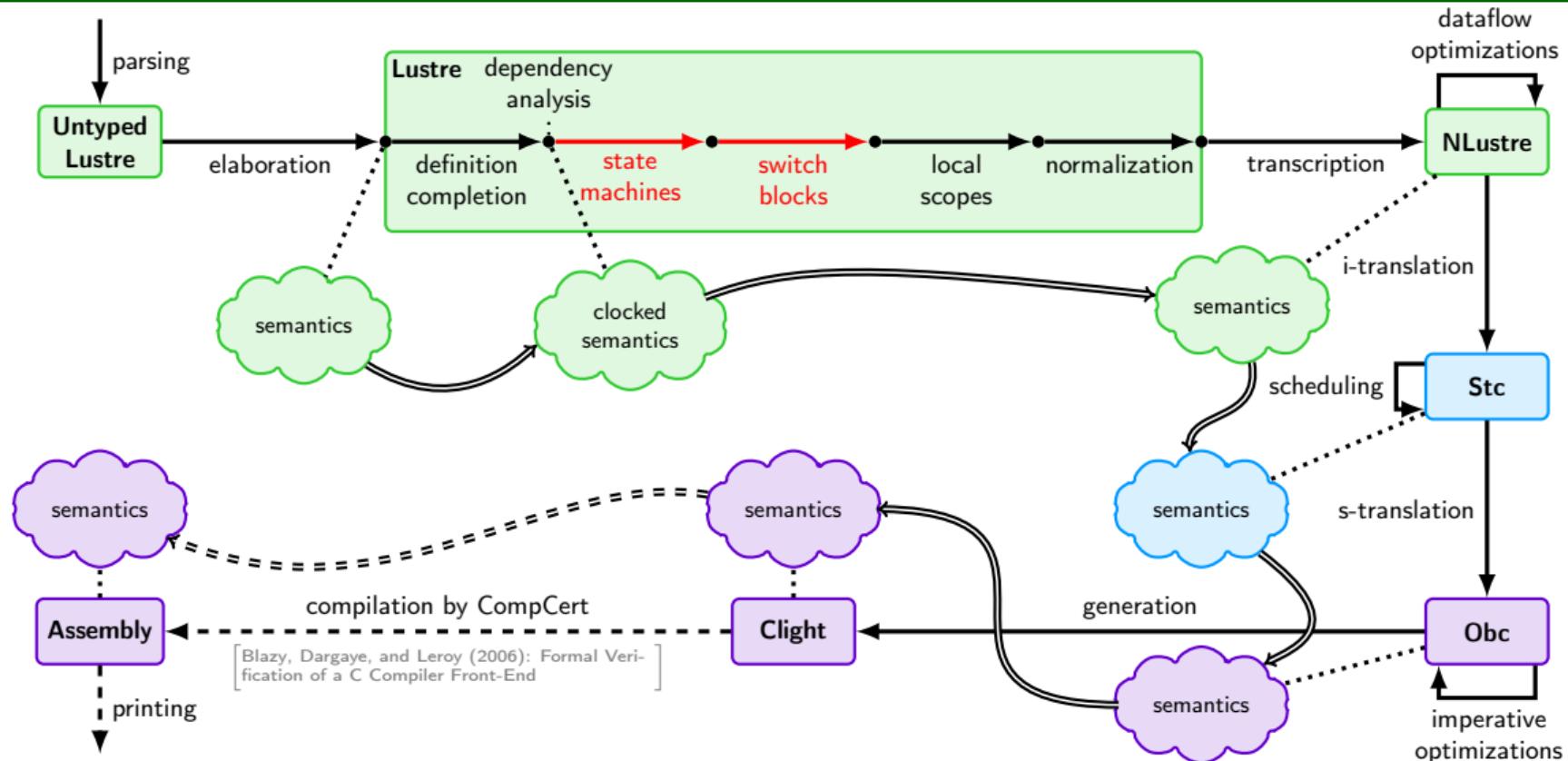
Instrumented Semantic Model



Instrumented Semantic Model



Compilation of State Machines and Switch Blocks



Compilation of State Machines

```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
```

automaton initially Feeding

state Feeding do

ena = true;

automaton initially Starting

state Starting do

step = true fby false

unless time >= 750 then Moving

state Moving do

step = true fby false

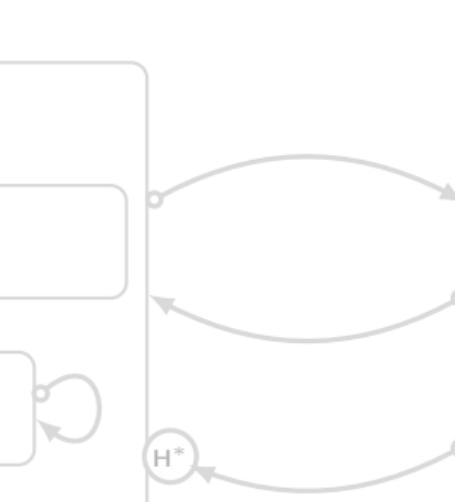
unless time >= 500 then Moving

end;

unless pause then Holding

end

tel



state Holding do

step = false;

automaton initially Waiting

state Waiting do

ena = true

unless time >= 500 then Modulating

state Modulating do

ena = pwm(true)

end;

unless

| not pause and time >= 750 then Feeding

| not pause continue Feeding

Compilation of State Machines

```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
  time = count_up(50)
every (false fby step);
```

automaton initially Feeding

```
state Feeding do
  ena = true;
automaton initially Starting
```

```
state Starting do
  step = true fby false
unless time >= 750 then Moving
```

```
state Moving do
  step = true fby false
unless time >= 500 then Moving
```

```
end;
unless pause then Holding
end
tel
```

```
state Holding do
  step = false;
automaton initially Waiting
```

```
state Waiting do
  ena = true
unless time >= 500 then Modulating
```

```
state Modulating do
  ena = pwm(true)
end;
unless
| not pause and time >= 750 then Feeding
| not pause continue Feeding
```

Compilation of State Machines

automaton initially Starting

```
state Starting do
  step = true fby false
unless time >= 750 then Moving
```

```
state Moving do
  step = true fby false
unless time >= 500 then Moving
```

end

Compilation of State Machines

automaton initially Starting

```
state Starting do
  step = true fby false
unless time >= 750 then Moving
```

```
state Moving do
  step = true fby false
unless time >= 500 then Moving
```

end

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

GMatch (σ) ($C_1 \rightarrow (D_1, N_1)$, ..., $C_n \rightarrow (D_n, N_n)$) =

$\bigcup_{\text{clock } x \in \text{and}}$

$y = \text{merge } x$

$y = \text{prod}_{i=1}^n \sigma(C_i)$

\dots

$y = \text{prod}_{i=1}^n \sigma(C_i)$

and

$y = \text{prod}_{i=1}^n \sigma(C_i)$

\dots

$y = \text{prod}_{i=1}^n \sigma(C_i)$

and

$y = \text{prod}_{i=1}^n \sigma(C_i)$

\dots

$y = \text{prod}_{i=1}^n \sigma(C_i)$

where $N_i = \text{prod}_{j=1}^{n-i} N_j \cup \dots \cup N_1$

and $y = \text{prod}_{i=1}^n \sigma(C_i)$

and $y = \text{prod}_{i=1}^n \sigma(C_i)$

Figure 5: The translation of `match`

when This code is translated into

$\text{clock } x = \text{prod}_{i=1}^n \text{clock } x_i \text{ when } \text{length}(x_i) = 1 \Rightarrow x_i$

and $x = \text{prod}_{i=1}^n \text{clock } x_i \text{ when } \text{length}(x_i) > 1$

$\text{and } y = \text{prod}_{i=1}^n \text{prod}_{j=1}^{n-i} \text{clock } y_{ij} \text{ when } \text{length}(y_{ij}) = 1 \Rightarrow y_{ij}$

$\text{and } y = \text{prod}_{i=1}^n \text{prod}_{j=1}^{n-i} \text{clock } y_{ij} \text{ when } \text{length}(y_{ij}) > 1$

This translation highlights the fact that `prod` is in the

language of synchronous dataflow

GMatch (σ) ($S_1 \rightarrow (D_1, N_1)$, ..., $S_n \rightarrow (D_n, N_n)$) =

$\bigcup_{\text{clock } x \in \text{and}}$

$y = \text{prod}_{i=1}^n \sigma(S_i)$

\dots

$y = \text{prod}_{i=1}^n \sigma(S_i)$

and

$y = \text{prod}_{i=1}^n \sigma(S_i)$

\dots

$y = \text{prod}_{i=1}^n \sigma(S_i)$

and

$y = \text{prod}_{i=1}^n \sigma(S_i)$

where $N_i = \text{prod}_{j=1}^{n-i} N_j \cup \dots \cup N_1$

and $y = \text{prod}_{i=1}^n \sigma(S_i)$

and $y = \text{prod}_{i=1}^n \sigma(S_i)$

Figure 6: The translation of `match`

possible reasoning and have strongly typed automata, that is, to whom operate upon them. Thus, `match` is more difficult than `prod` to implement in VélusCompiler, and largely limits the use of programs undergoing analysis and synthesis.

2.3.2 The Type System

We shall now introduce the typing rules for the new program constructs. The typing rules are similar to those types of transitions introduced in Section 2.1. The type states in particular are the same as in the original language. Their role is to provide a type environment for the compilation of programs, as well as for their execution (as described in Section 2.2). The type system is based on a type system for the original language, which is a very simplified construction (with full abstraction), as well as on soundness and completeness proofs for the

Compilation of State Machines

automaton initially Starting

```
state Starting do
  step = true fby false
  unless time >= 750 then Moving

state Moving do
  step = true fby false
  unless time >= 500 then Moving

end
```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of `automaton`

Figure 5 shows the translation of the automaton code into synchronous dataflow. The code defines two states: `Starting` and `Moving`. In the `Starting` state, the step is set to `true fby false`, and if the time is not greater than or equal to 750, it transitions to the `Moving` state. In the `Moving` state, the step is set to `true fby false`, and if the time is not greater than or equal to 500, it loops back to the `Moving` state. The `end` keyword indicates the end of the automaton definition.

Figure 6: The translation of `automaton`.

Figure 6 shows the detailed translation of the automaton code into synchronous dataflow. It includes annotations explaining the translation rules and the resulting dataflow graph. The annotations describe how parallel regions and label merging are handled, and how the original automaton's behavior is preserved in the translated code.

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
    every pres
    | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
    every res
    | Moving do ...
  end
tel
```

Compilation of State Machines

automaton initially Starting

```

state Starting do
  step = true fby false
unless time >= 750 then Moving

state Moving do
  step = true fby false
unless time >= 500 then Moving

end

```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of `switch`

Code: $\{G_1 \rightarrow (D_1, N_1)\}, \{G_2 \rightarrow (D_2, N_2)\} =$
 $\{D_1 \rightarrow \text{merge } x \text{ and}$
 $y = \text{merge } z \text{ and}$
 $z = \text{merge } a$
 $a = \text{merge } \{G_1 \rightarrow \text{merge } x^{\text{out}}(G_1)\}$
 \dots
 $\{G_n \rightarrow \text{merge } x^{\text{out}}(G_n)\}$
 $\text{and } \text{merge } x^{\text{out}}$
 $x^{\text{out}} = \text{merge } \{G_1 \rightarrow \text{merge } x^{\text{out}}(G_1)\}$
 \dots
 $\{G_n \rightarrow \text{merge } x^{\text{out}}(G_n)\}$
 $\text{where } x = \text{merge } \{x^{\text{in}}, x^{\text{out}}\}$
 $\text{and } x^{\text{in}} = N_1 \cup \dots \cup N_n$
 $\text{and } x^{\text{out}} = \text{merge } \{N_1 \cup \dots \cup N_n\}$

Figure 6: The translation of `switch`

Code: $\{G_1 \rightarrow (D_1, \text{move } x)\}, \{G_2 \rightarrow (D_2, \text{move } y)\} =$
 $\{D_1 \rightarrow \text{move } x \text{ and}$
 $\text{move } y = \text{move } z \text{ and}$
 $z = \text{move } a$
 $a = \text{move } \{G_1 \rightarrow \text{move } x^{\text{out}}(G_1)\}$
 \dots
 $\{G_n \rightarrow \text{move } x^{\text{out}}(G_n)\}$
 $\text{and } \text{move } x^{\text{out}}$
 $x^{\text{out}} = \text{move } \{G_1 \rightarrow \text{move } x^{\text{out}}(G_1)\}$
 \dots
 $\{G_n \rightarrow \text{move } x^{\text{out}}(G_n)\}$
 $\text{where } x = \text{move } \{x^{\text{in}}, x^{\text{out}}\}$
 $\text{and } x^{\text{in}} = N_1 \cup \dots \cup N_n$
 $\text{and } x^{\text{out}} = \text{move } \{N_1 \cup \dots \cup N_n\}$

Notes: This code is translated into:
 $\text{clock } x = 1 \text{ and } \text{clock } y = 1 \text{ when } \text{clock}(x) = 1 \Rightarrow x$
 $\text{and } x = \text{move } z \text{ when } \text{move}(z) = 1 \Rightarrow z$
 $\text{and } z = \text{move } a \text{ when } \text{move}(a) = 1 \Rightarrow a$
 $\text{clock } x = 1 \text{ and } \text{clock } y = 1 \text{ when } \text{clock}(y) = 1 \Rightarrow y$
 $\text{and } y = \text{move } z \text{ when } \text{move}(z) = 1 \Rightarrow z$
 $\text{and } z = \text{move } a \text{ when } \text{move}(a) = 1 \Rightarrow a$

This translation highlights the fact that `move` is in the scope of `clock`, while `move` is not in the scope of `clock`.

```

var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res)
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
  every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
  every res
  | Moving do ...
  end
tel

```

Compilation of State Machines

automaton initially Starting

```
state Starting do
    step = true fby false
unless time >= 750 then Moving
```

```
state Moving do
    step = true fby false
unless time >= 500 then Moving
```

end

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of `start`

Figure 5 shows the translation of the `start` automaton. It includes two sets of code snippets: one for the `Starting` state and one for the `Moving` state. The `Starting` state code uses `fby` and `unless` constructs. The `Moving` state code also uses `fby` and `unless`. Below the code is a note about the translation of `start`.

Figure 6: The translation of `start`.

Figure 6 shows the detailed translation of the `start` automaton. It includes several sets of code snippets, each enclosed in a yellow box. These snippets represent different parts of the state machine's logic, such as initial conditions and transitions between states. The code uses various operators like `merge`, `and`, `or`, and `reset`.

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
  every pres
  | Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
  | Moving do ...
end
tel
```

Compilation of State Machines

automaton initially Starting

```
state Starting do
  step = true fby false
  unless time >= 750 then Moving

state Moving do
  step = true fby false
  unless time >= 500 then Moving

end
```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of `start`

Figure 5 shows the translation of the `start` automaton. It consists of two parts: a textual representation of the automaton and its corresponding synchronous data-flow graph.



Figure 6: The translation of automata. This diagram illustrates the conservative extension of SDF with state machines. It shows how a simple state machine (an automaton with states and transitions) is translated into a synchronous data-flow graph (a directed graph with nodes and edges). The nodes represent states, and the edges represent transitions, often involving operations like merging or joining multiple data flows.

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
  every pres
  | Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
  | Moving do ...
end
tel
```

Compilation of State Machines

automaton initially Starting

```
state Starting do
  step = true fby false
unless time >= 750 then Moving
```

```
state Moving do
  step = true fby false
unless time >= 500 then Moving
```

end

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of `start`

Code: $(x) \{ C_1 \rightarrow (D_1, N_1); \dots; C_n \rightarrow (D_n, N_n) \} =$
 $\{ \text{clock } x \text{ and}$
 $y = \text{merge } x$
 \dots
 $C_1 \rightarrow \text{pres}_x^{C_1}(y|G_1)$
 \dots
 $C_n \rightarrow \text{pres}_x^{C_n}(y|G_n)$
 $\text{and } \dots$
 $y = \text{merge } x$
 \dots
 $G_1 \rightarrow \text{pres}_x^{G_1}(y|G_1)$
 \dots
 $G_n \rightarrow \text{pres}_x^{G_n}(y|G_n)$

where $y = \text{merge } x$ if $x \in N_1 \cup \dots \cup N_n$
 $\text{and } y|G_i = \text{skip}_{N_i \cup \dots \cup N_n}(G_i)$
 $\text{and } \text{skip}_{N_i \cup \dots \cup N_n}(G_i) = \text{skip}_{N_i}(G_i) \cup \dots \cup \text{skip}_{N_n}(G_n)$

Figure 6: The translation of `start`

Code: $(x) \{ D_1 \rightarrow (D_1, N_1); \dots; D_n \rightarrow (D_n, N_n) \} =$
 $\{ \text{clock } x \text{ and}$
 $\text{and } y = \text{merge } x$
 \dots
 $D_1 \rightarrow \text{pres}_x^{D_1}(y|G_1)$
 \dots
 $D_n \rightarrow \text{pres}_x^{D_n}(y|G_n)$
 $\text{and } \dots$
 $y = \text{merge } x$
 \dots
 $G_1 \rightarrow \text{pres}_x^{G_1}(y|G_1)$
 \dots
 $G_n \rightarrow \text{pres}_x^{G_n}(y|G_n)$

where $y = \text{merge } x$ if $x \in N_1 \cup \dots \cup N_n$
 $\text{and } y|G_i = \text{skip}_{D_i \cup \dots \cup D_n}(G_i)$
 $\text{and } \text{skip}_{D_i \cup \dots \cup D_n}(G_i) = \text{skip}_{D_i}(G_i) \cup \dots \cup \text{skip}_{D_n}(G_n)$

Notes: This code is translated into:
 $\text{clock } x \text{ and } \text{skip}_{N_1 \cup \dots \cup N_n}(\text{skip}_{N_1}(x) \wedge \dots \wedge \text{skip}_{N_n}(x))$
 $\text{and } y = \text{merge } x \Rightarrow x = \text{skip}_{N_1 \cup \dots \cup N_n}(y)$
 $\text{and } y|G_i = \text{skip}_{N_i \cup \dots \cup N_n}(G_i) \Rightarrow G_i = \text{skip}_{N_i}(G_i) \wedge \dots \wedge \text{skip}_{N_n}(G_n)$

This translation highlights the fact that `pres` is in the scope of `clock`, which is a common mistake in the implementation of state machines.

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
  every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
  every res
  | Moving do ...
  end
tel
```

Compilation of State Machines

automaton initially Starting

```
state Starting do
  step = true fby false
unless time >= 750 then Moving
```

```
state Moving do
  step = true fby false
unless time >= 500 then Moving
```

end

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of `automaton`

This code is translated into:

```
clock x := true
and x = merge a
y = merge b
and y = merge c
z = merge d
and z = merge e
where a = merge((x))
and b = merge((y))
and c = merge((z))
and d = merge((x))
and e = merge((y))
```

Figure 6: The translation of `automaton`

This translation highlights the fact that `merge` is in the global scope. This code is translated into:

```
clock x := true
and x = merge((x))
and y = merge((y))
and z = merge((z))
and d = merge((d))
and e = merge((e))
clock x := true
and y = merge((y))
and z = merge((z))
and d = merge((d))
and e = merge((e))
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res)
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
        then (Moving, true)
        else (Starting, false)
  every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
  every res
  | Moving do ...
  end
tel
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
    every pres
  | Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
| Moving do ...
end
tel
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
    every res
  | Moving do ...
  end
tel
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

But Coq is a pure functional language!

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
    every res
  | Moving do ...
  end
tel
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

But Coq is a pure functional language!

- Monadic approach: Fresh

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
    every res
  | Moving do ...
  end
tel
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

But Coq is a pure functional language!

- Monadic approach: Fresh
- Access OCaml code: gensym

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
    step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Coq Implementation

```

Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states =>
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
    Beq ([pst; pres],
         [Efy [Eenum oth; Eenum false]
          [Evar st; Evar res]]) in
  let branches := map (fun' ((e, _), (unl, _)) =>
    let transeq := Beq ([st; res], trans_exp unl e) in
    (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun' ((e, _), (_, (blks, _))) =>
    do blks' ← mmap auto_block blks;
    ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])

```

```

var pst, pres, st, res; let
(pst, pres) = (Starting, false) fby (st, res);
switch pst
| Starting do
  reset
  (st, res) =
    if time >= 750
    then (Moving, true)
    else (Starting, false)
  every pres
| Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
| Moving do ...
end
tel

```

Compilation of State Machines – Coq Implementation

```

Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states =>
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
    Beq ([pst; pres],
         [Efy [Eenum oth; Eenum false]
          [Evar st; Evar res]]) in
  let branches := map (fun' ((e, _), (unl, _)) =>
    let transeq := Beq ([st; res], trans_exp unl e) in
    (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun' ((e, _), (_, (blks, _))) =>
    do blks' ← mmap auto_block blks;
    ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])

```

```

var pst, pres, st, res; let
(pst, pres) = (Starting, false) fby (st, res);
switch pst
| Starting do
  reset
  (st, res) =
    if time >= 750
    then (Moving, true)
    else (Starting, false)
every pres
| Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
every res
| Moving do ...
end
tel

```

Compilation of State Machines – Coq Implementation

```

Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states =>
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
    Beq ([pst; pres],
         [Efy [Eenum oth; Eenum false]
          [Evar st; Evar res]]) in
  let branches := map (fun' ((e, _), (unl, _)) =>
    let transeq := Beq ([st; res], trans_exp unl e) in
    (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun' ((e, _), (_, (blks, _))) =>
    do blks' ← mmap auto_block blks;
    ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])

```

```

var pst, pres, st, res: let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
    (st, res) =
      if time >= 750
      then (Moving, true)
      else (Starting, false)
  every pres
  | Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
| Moving do ...
end
tel

```

Common monadic notation:

do x ← e1; e2 ~ let x := e1 in e2

Compilation of State Machines – Coq Implementation

```

Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states =>
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
    Beq ([pst; pres],
         [Efby [Eenum oth; Eenum false]
          [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) =>
    let transeq := Beq ([st; res], trans_exp unl e) in
    (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) =>
    do blks' ← mmap auto_block blks;
    ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])

```

```

var pst, pres, st, res; let
(pst, pres) = (Starting, false) fby (st, res);
switch pst
| Starting do
  reset
  (st, res) =
    if time >= 750
    then (Moving, true)
    else (Starting, false)
  every pres
| Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
| Moving do ...
end
tel

```

Compilation of State Machines – Coq Implementation

```

Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states =>
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
    Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
            [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) =>
    let transeq := Beq ([st; res], trans_exp unl e) in
    (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_ , (blk, _))) =>
    do blks' ← mmap auto_block blk;
    ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])

```

```

var pst, pres, st, res; let
(pst, pres) = (Starting, false) fby (st, res);
switch pst
| Starting do
  reset
  (st, res) =
    if time >= 750
    then (Moving, true)
    else (Starting, false)
  every pres
| Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
  every res
| Moving do ...
end
tel

```

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=  
match blk with  
| ...  
| Bauto Strong ck (_, oth) states =>  
  do pst ← fresh_ident; do pres ← fresh_ident;  
  do st ← fresh_ident; do res ← fresh_ident;  
  let stateq :=  
    Beq ([pst; pres],  
         [Efby [Eenum oth; Eenum false]  
          [Evar st; Evar res]]) in  
  let branches := map (fun '((e, _), (unl, _)) =>  
    let transeq := Beq ([st; res], trans_exp unl e) in  
    (e, [Breset [transeq] (Evar pres)])) states in  
  let sw1 := Bswitch (Evar pst) branches in  
  do branches ← mmap (fun '((e, _), (_, (blk, _))) =>  
    do blks' ← mmap auto_block blk;  
    ret (e, ([Breset blks' (Evar res)]))) states;  
  let sw2 := Bswitch (Evar st) branches in  
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let  
(pst, pres) = (Starting, false) fby (st, res);  
switch pst  
| Starting do  
  reset  
  (st, res) =  
    if time >= 750  
    then (Moving, true)  
    else (Starting, false)  
every pres  
| Moving do ...  
end;  
switch st  
| Starting do  
  reset  
  step = true fby false  
every res  
| Moving do ...  
end  
tel
```

Compilation of State Machines – Coq Implementation

```

Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states =>
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
    Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
            [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) =>
    let transeq := Beq ([st; res], trans_exp unl e) in
    (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) =>
    do blks' ← mmap auto_block blks;
    ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocl [pst; pres; st; res] [stateq; sw1; sw2])

```

```

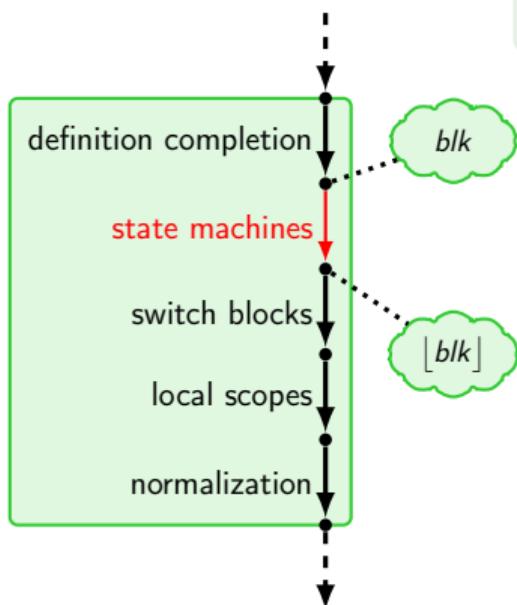
var pst, pres, st, res; let
(pst, pres) = (Starting, false) fby (st, res);
switch pst
| Starting do
  reset
  (st, res) =
    if time >= 750
    then (Moving, true)
    else (Starting, false)
every pres
| Moving do ...
end;
switch st
| Starting do
  reset
  step = true fby false
every res
| Moving do ...
end
tel

```

Compilation of State Machines – Proof Intuition

Lemma (State machines correctness)

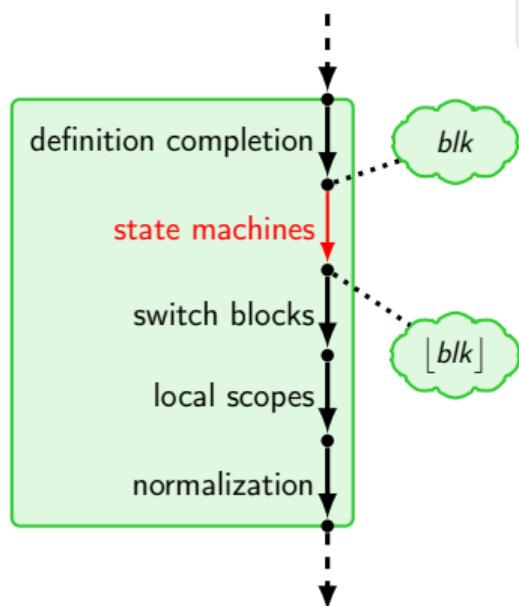
$$\text{if } G, H \vdash blk \text{ then } G, H \vdash [blk]$$



Compilation of State Machines – Proof Intuition

Lemma (State machines correctness)

$$\text{if } G, H \vdash blk \text{ then } G, H \vdash [blk]$$



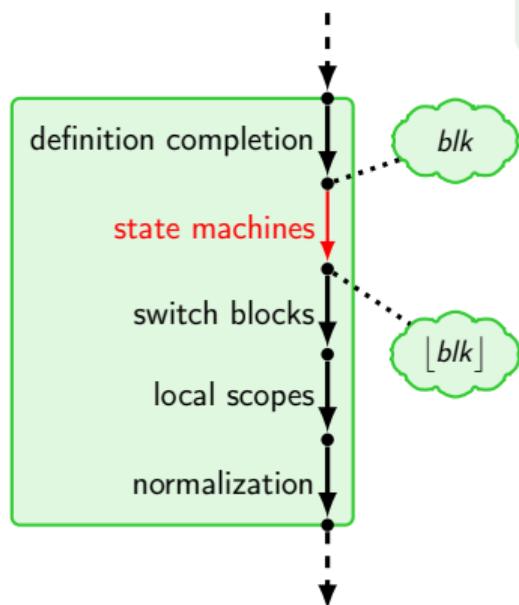
Works well:

- local transformation and reasoning
- correspondence between select, mask and when

Compilation of State Machines – Proof Intuition

Lemma (State machines correctness)

$$\text{if } G, H \vdash blk \text{ then } G, H \vdash [blk]$$



Works well:

- local transformation and reasoning
- correspondence between select, mask and when

Works less well:

- static invariants (typing, clock-typing, ...)
- fresh identifiers

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γc Hi bs blk' tys st st',
  (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
  (∀ x, IsVar Γck x → IsVar Γty x) →
  (∀ x, Islast Γck x → Islast Γty x) →
  NoDupLocals (List.map fst Γty) blk →
  GoodLocals elab_prefs blk →
  wt_block G1 Γty blk →
  wc_block G1 Γck blk →
  dom_ub Hi Γty →
  sc_vars Γck Hi bs →
  sem_block_ck G1 Hi bs blk →
  auto_block blk st = (blk', tys, st') →
  sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γck Hi bs blk' tys st st',
  (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
  (∀ x, IsVar Γck x → IsVar Γty x) →
  (∀ x, Islast Γck x → Islast Γty x) →
  NoDupLocals (List.map fst Γty) blk →
  GoodLocals elab_prefs blk →
  wt_block G1 Γty blk →
  wc_block G1 Γck blk →
  dom_ub Hi Γty →
  sc_vars Γck Hi bs →
  sem_block_ck G1 Hi bs blk →
  auto_block blk st = (blk', tys, st') →
  sem_block_ck G2 Hi bs blk'.
```

Proof.

```
induction blk using block_ind;
```

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Tty Γck Hi bs blk' tys st st',
  (∀ x, IsVar Tty x → AtomOrGensym elab_prefs x) →
  (∀ x, IsVar Γck x → IsVar Tty x) →
  (∀ x, Islast Γck x → Islast Tty x) →
  NoDupLocals (List.map fst Tty) blk →
  GoodLocals elab_prefs blk →
  wt_block G1 Tty blk →
  wc_block G1 Γck blk →
  dom_ub Hi Tty →
  sc_vars Γck Hi bs →
  sem_block_ck G1 Hi bs blk →
  auto_block blk st = (blk', tys, st') →
  sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γc Hi bs blk' tys st st',
  (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
  (∀ x, IsVar Γck x → IsVar Γty x) →
  (∀ x, Islast Γck x → Islast Γty x) →
  NoDupLocals (List.map fst Γty) blk →
  GoodLocals elab_prefs blk →
  wt_block G1 Γty blk →
  wc_block G1 Γck blk →
  dom_ub Hi Γty →
  sc_vars Γck Hi bs →
  sem_block_ck G1 Hi bs blk →
  auto_block blk st = (blk', tys, st') →
  sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$

Compilation of State Machines – Coq Proof

```

Lemma auto_block_sem : ∀ blk Fty Fck Hi bs blk' tys st st',
  (∀ x, IsVar Fty x → AtomOrGensym elab_prefs x) →
  (∀ x, IsVar Fck x → IsVar Fty x) →
  (∀ x, IsLast Fck x → IsLast Fty x) →
  NoDupLocals (List.map fst Fty) blk →
  GoodLocals elab_prefs blk →
  wt_block G1 Fty blk →
  wc_block G1 Fck blk →
  dom_ub Hi Fty →
  sc_vars Fck Hi bs →
  sem_block_ck G1 Hi bs blk →
  auto_block blk st = (blk', tys, st') →
  sem_block_ck G2 Hi bs blk'.

```

Proof.

```
induction blk using block_ind2;
```

```

(* auto_block_sem *)
(* Case 1: auto_block_sem (wt_block G1 Fty blk) *)
(* Case 2: auto_block_sem (wc_block G1 Fck blk) *)
(* Case 3: auto_block_sem (dom_ub Hi Fty) *)
(* Case 4: auto_block_sem (sc_vars Fck Hi bs) *)
(* Case 5: auto_block_sem (sem_block_ck G1 Hi bs blk) *)
(* Case 6: auto_block_sem (auto_block blk st = (blk', tys, st')) *)
(* Case 7: auto_block_sem (sem_block_ck G2 Hi bs blk') *)

```

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$

```

(* Lemma (State machines correctness) *)
(* if G, H ⊢ blk then G, H ⊢ [blk] *)
(* Proof by induction on blk *)
(* Case 1: auto_block_sem (wt_block G1 Fty blk) *)
(* Case 2: auto_block_sem (wc_block G1 Fck blk) *)
(* Case 3: auto_block_sem (dom_ub Hi Fty) *)
(* Case 4: auto_block_sem (sc_vars Fck Hi bs) *)
(* Case 5: auto_block_sem (sem_block_ck G1 Hi bs blk) *)
(* Case 6: auto_block_sem (auto_block blk st = (blk', tys, st')) *)
(* Case 7: auto_block_sem (sem_block_ck G2 Hi bs blk') *)

```

Compilation of Switch Blocks

```
switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end
```

```
resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;
```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]



Figure 5: The translation of switch

switch x
| Starting do
 reset
 step = true fby false
 every res
| Holding do ...
end

where $x \in \{x_1, \dots, x_n\}$ and $x_i \in \{x_1, \dots, x_n\}$
and $x_1, \dots, x_n \in \{x_1, \dots, x_n\}$
and $x_1, \dots, x_n \in \{x_1, \dots, x_n\}$

Figure 5: The translation of switch

switch x
| Starting do
 reset
 step = true fby false
 every res
| Holding do ...
end

This translation highlights the fact that `every` is in the same scope as `reset`, which is a common mistake.

Figure 6: The translation of switch

switch x
| Starting do
 reset
 step = true fby false
 every res
| Holding do ...
end

possible rewrites and later strongly depend on mutation, that is, to whom applies this transformation. The difference is, in some sense, like this transformation, and largely related with the notion of `STATEFULNESS`, and largely related with the notion of `PROGRAM UNDERSTANDING` and `APPLICATION`.

2.3.2 The Type System

We shall now introduce the typical rule for the new program under construction. The typical rule is the one that translates synchronous data-flow (the same types as the original language) into the target language. These rules state in particular that the type of a variable is the same as the type of the variable under construction. This means that a variable can have different types at different times (as in the case of a variable that is used in the same scope as `reset`, which is a common mistake).

Figure 6: The translation of switch

Compilation of Switch Blocks

```
switch st
| Starting do
  reset
  step = true fby false
  every res
| Holding do ...
end
```

```
resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every res;
```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]

Figure 5: The translation of switch

This code is translated into:

```
switch x
  case 0: do
    step := true fby false
    every res
  case 1: do
    step := true fby false
    every res
  end
end
```

Figure 6: The translation of switch

This code is translated into:

```
switch x
  case 0: do
    step := true fby false
    every res
  case 1: do
    step := true fby false
    every res
  end
end
```

This translation highlights the fact that `every` is in the scope of `do`, which is a common mistake in the original code.

- sampling explicated by `when`

possible reasoning and later strongly depend on mutation, that is, to some extent. This translation is conservative, and hopefully will not break existing programs, but it is not yet complete, and needs more work on program understanding and synthesis.

2.3.2 The Type System

We shall now discuss the typical rules for the new program semantics. The typical rule for the type system is the translation of a type rule from the old type system to the new one. There are two main types of translation: semantic and syntactic. These rules state in particular how to translate the new types into the old ones (which is a task that nicely interpenetrates compilation and verification). In our setting we need to consider full substitutions, i.e., a mapping from old types to new ones for every component of the

Compilation of Switch Blocks

```

switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end

```

```

resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;

```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]



Figure 5: The translation of switch.
This code is translated into:
switch x < 10 do if x < 10 then f(x) = 1 > x
and x = 10 then g(x) = 2 > x
and x > 10 then h(x) = 3 > x
This translation highlights the fact that **switch** is in the scope of **if**, **then**, and **else**.

Figure 6: The translation of **switch**.

This code is translated into:
switch x < 10 do if x < 10 then f(x) = 1 > x
and x = 10 then g(x) = 2 > x
and x > 10 then h(x) = 3 > x
This translation highlights the fact that **switch** is in the scope of **if**, **then**, and **else**.

- sampling explicated by **when**
- choice explicated by **merge**

Compilation of Switch Blocks

```

switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end

```

```

resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;

```

[Colaço, Pagano, and Pouzet (2005): A Conservative Extension
of Synchronous Data-flow with State Machines]



Figure 6: The translation of switch

switch x {
 case 0: y := 0;
 case 1: y := 1;
 case 2: y := 2;
}

Figure 6: The translation of switch

This code is translated into:

```

switch x {
  case 0: y := 0;
  case 1: y := 1;
  case 2: y := 2;
}

```

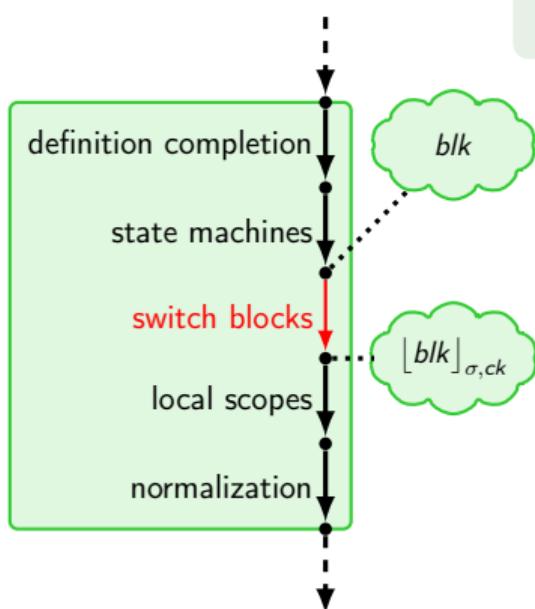
Figure 6: The translation of switch

- sampling explicated by **when**
- choice explicated by **merge**
- constants are also sampled

Compilation of Switch Blocks – Proof Intuition

Lemma (Switch correctness)

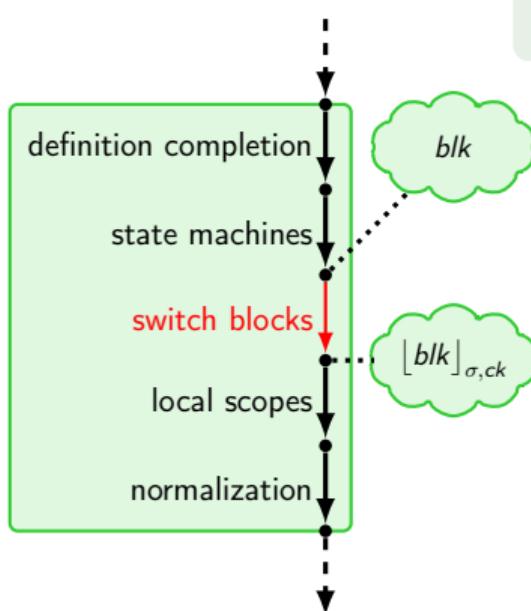
if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_{\sigma} H_2$ then $G, H_2 \vdash [blk]_{\sigma, ck}$



Compilation of Switch Blocks – Proof Intuition

Lemma (Switch correctness)

$$\text{if } G, H_1 \vdash blk \text{ and } H_1 \sqsubseteq_{\sigma} H_2 \text{ then } G, H_2 \vdash [blk]_{\sigma, ck}$$



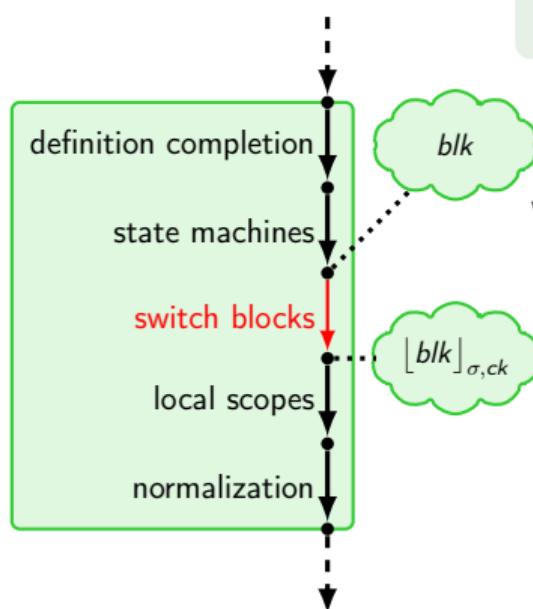
Works less well:

- reasoning is not local:
renaming propagates to
sub-blocks
- static invariants, in
particular clock-typing

Compilation of Switch Blocks – Proof Intuition

Lemma (Switch correctness)

if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_\sigma H_2$ then $G, H_2 \vdash [blk]_{\sigma, ck}$



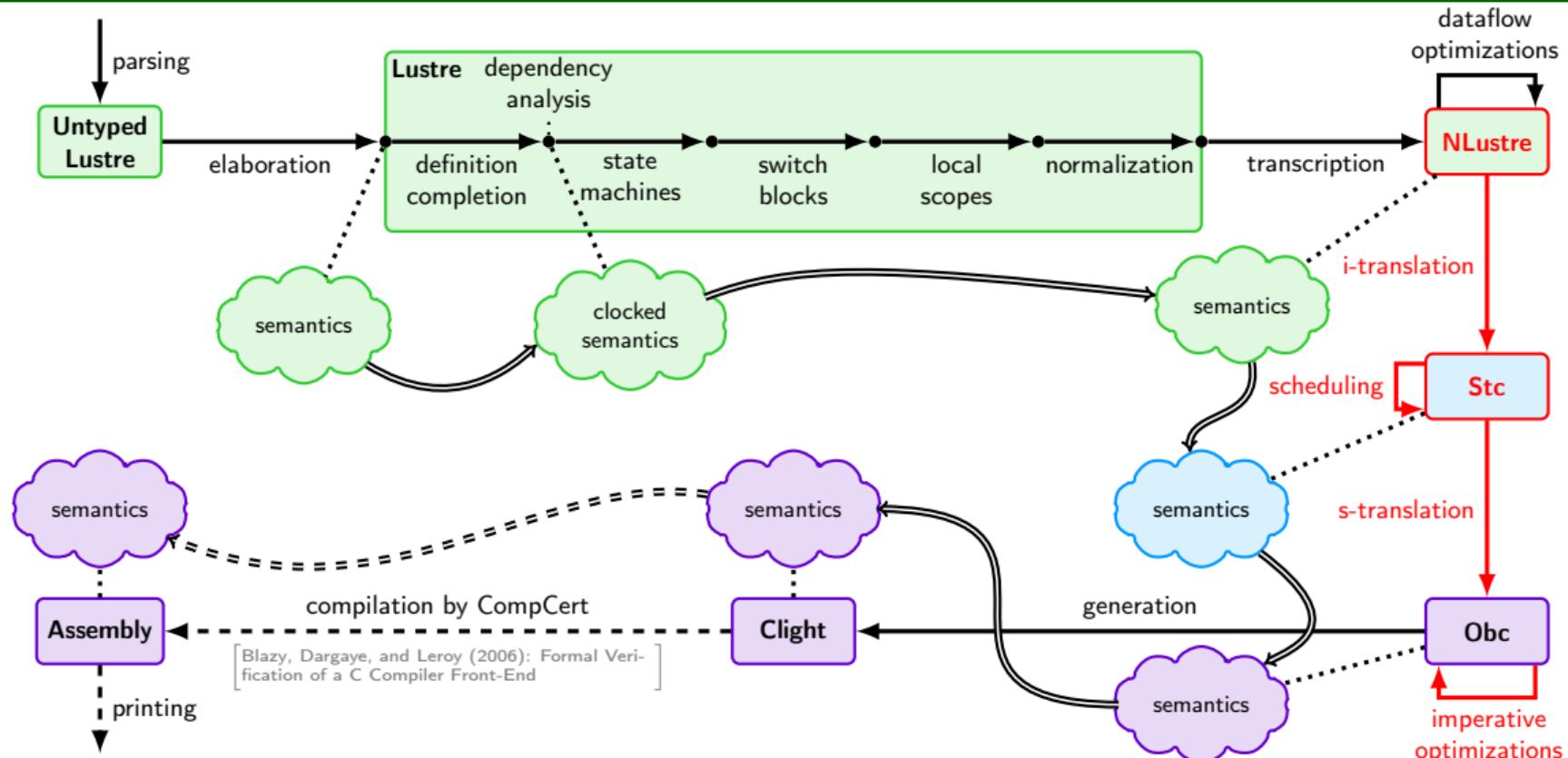
Works well:

- correspondence between **switch** and **when/merge**: implicit to explicit sampling
- less cases to handle

Works less well:

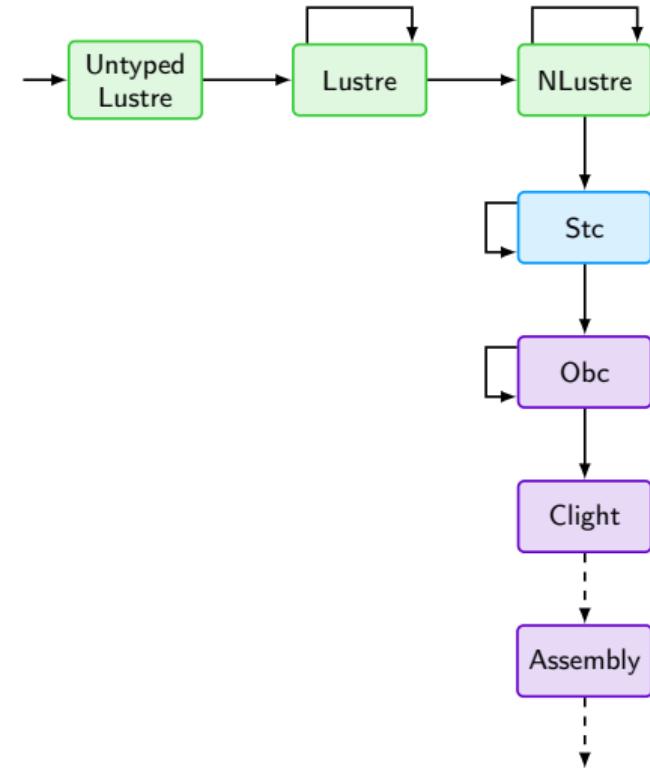
- reasoning is not local: renaming propagates to sub-blocks
- static invariants, in particular clock-typing

Compilation to Imperative Code



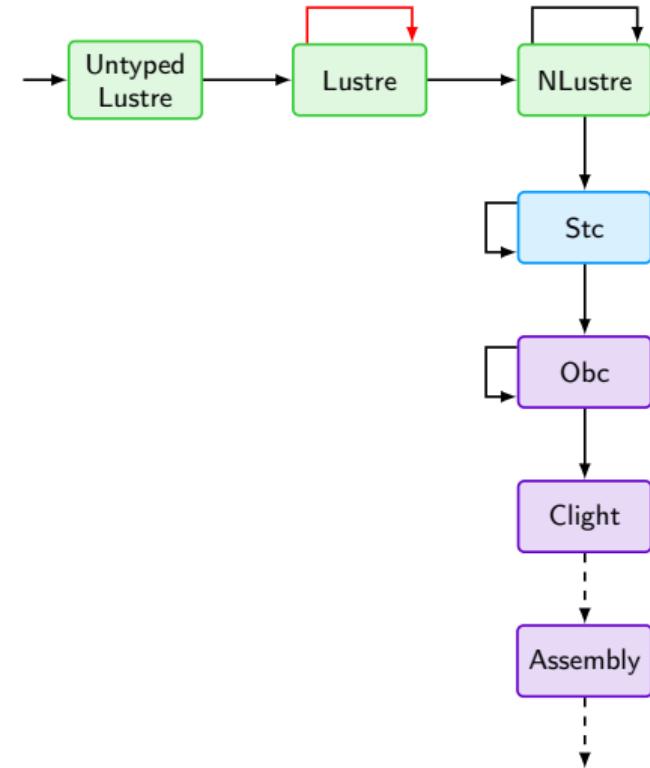
Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```



Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
↓
switch step
| true do
  mA = not last$mB;
  mB = last$mA;
| false do (mA, mB) = (last$mA, last$mB)
end;
last$mA = true fby mA;
last$mB = false fby mB;
```

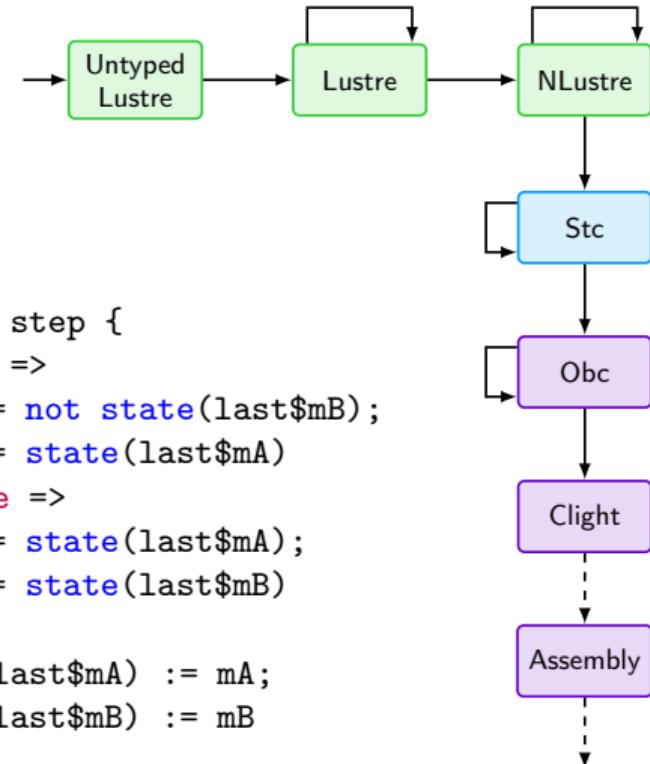


Compiling Last Variables

```

switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
 $\downarrow$ 
switch step
| true do
  mA = not last$mB; ----->
  mB = last$mA;
| false do (mA, mB) = (last$mA, last$mB)
end;
last$mA = true fby mA;
last$mB = false fby mB;

```



```

switch step {
| true =>
  mA := not state(last$mB);
  mB := state(last$mA)
| false =>
  mA := state(last$mA);
  mB := state(last$mB)
};
state(last$mA) := mA;
state(last$mB) := mB

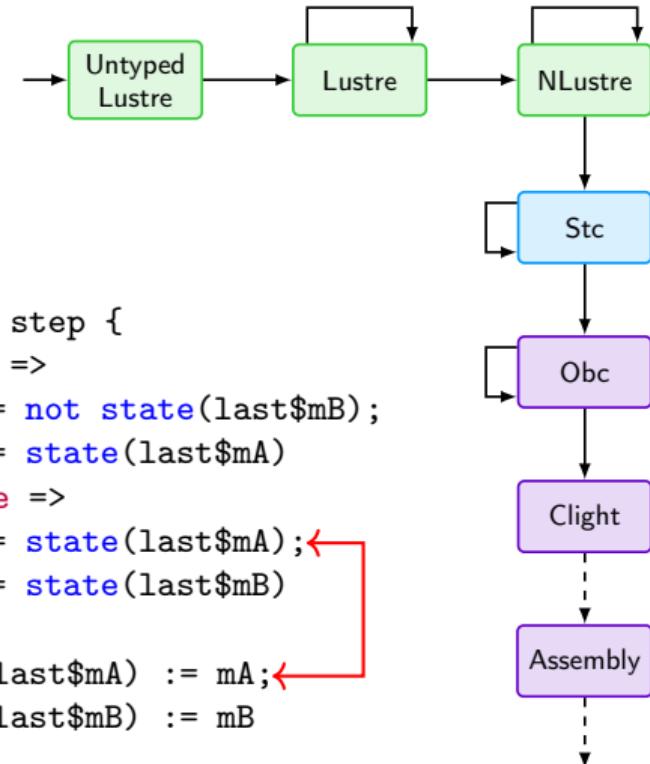
```

Compiling Last Variables

```

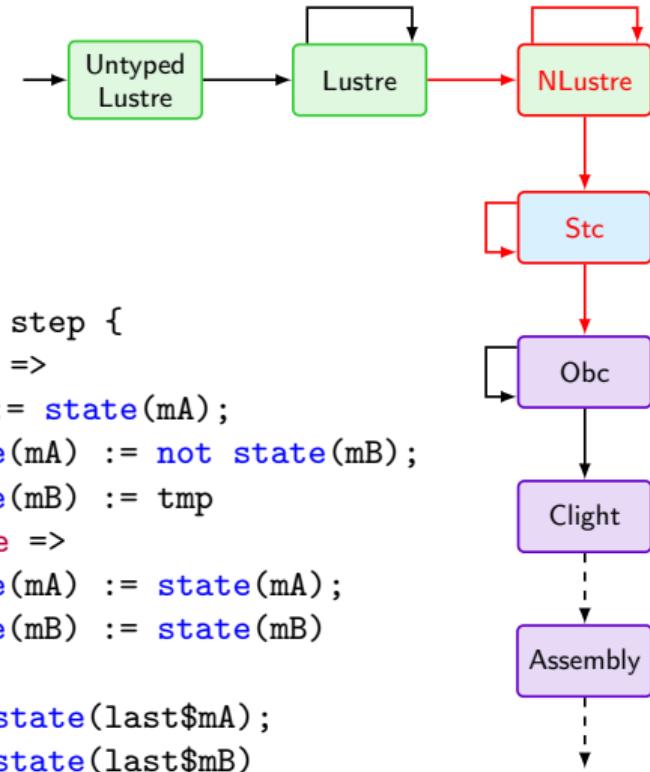
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
 $\downarrow$ 
switch step
| true do
  mA = not last$mB; ----->
  mB = last$mA;
| false do (mA, mB) = (last$mA, last$mB)
end;
last$mA = true fby mA;
last$mB = false fby mB;

```



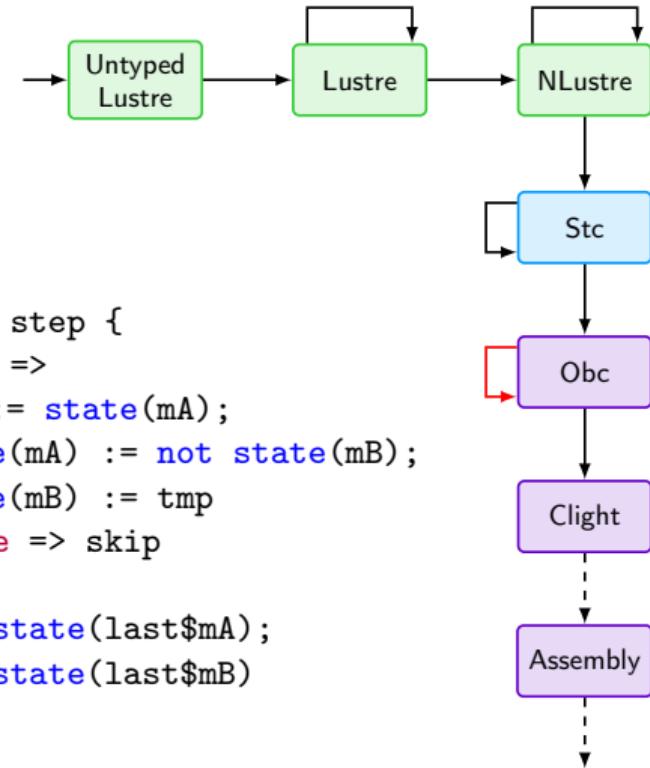
Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

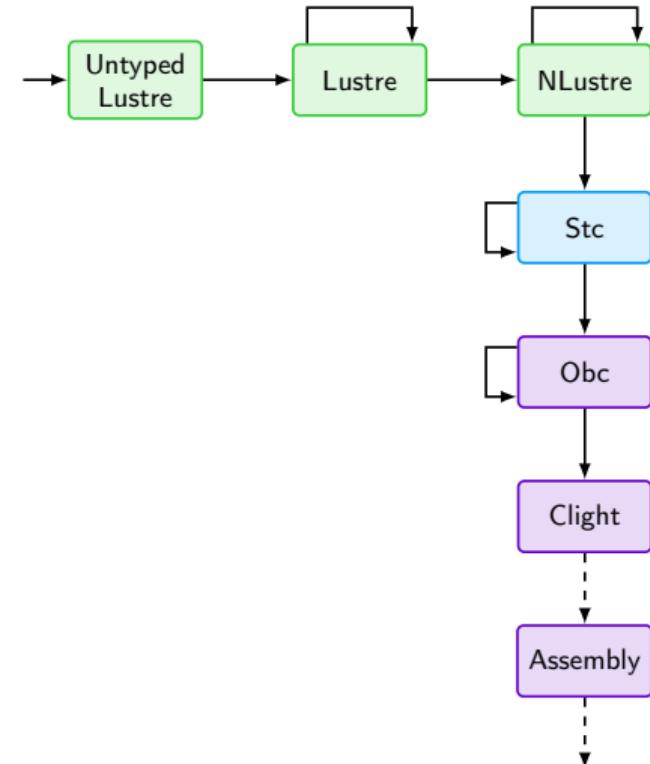


Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```



Main Correctness Theorem



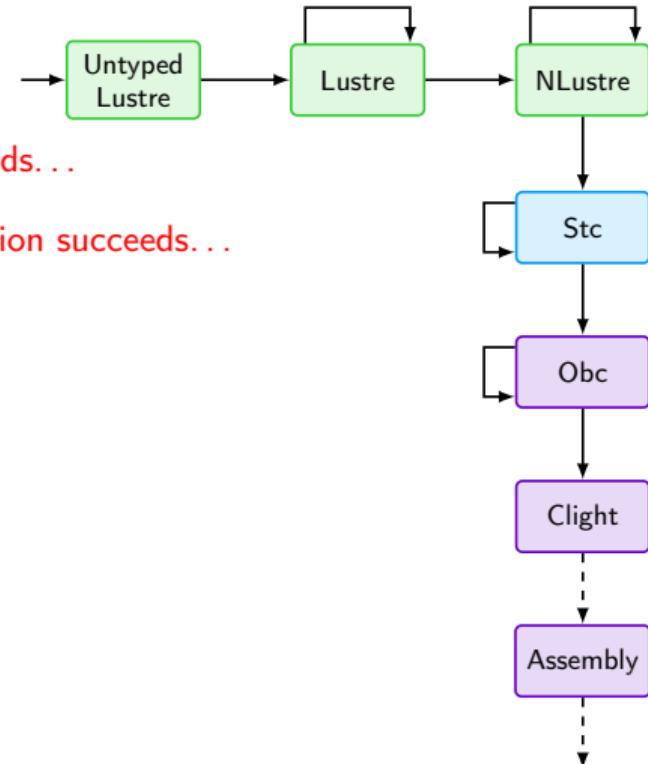
Theorem behavior_asm:

```
forall D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  compile D main = OK P →
  sem_node G main (pStr ins) (pStr outs) →
  wt_ins G main ins →
  wc_ins G main ins →
  exists T, program_behaves (Asm.semantics P) (Reacts T)
    ∧ bisim_IO G main ins outs T.
```

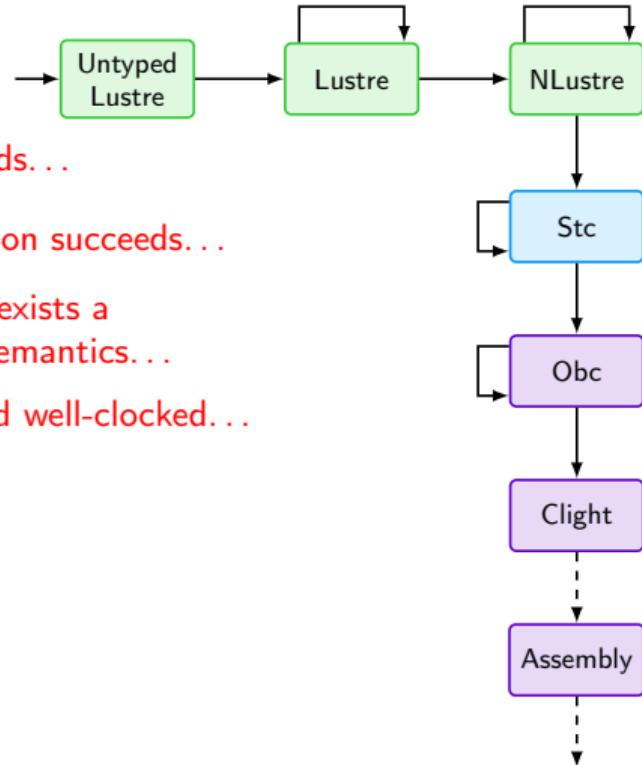
Main Correctness Theorem

Theorem behavior_asm:

$\forall D G Gp P \text{ main ins outs},$
 $\text{elab_declarations } D = \text{OK} (\text{exist } _- G Gp) \rightarrow$
 $\text{compile } D \text{ main } = \text{OK } P \rightarrow$ if typing/elaboration succeeds...
 $\text{sem_node } G \text{ main } (\text{pStr ins}) (\text{pStr outs}) \rightarrow$
 $\text{wt_ins } G \text{ main ins } \rightarrow$
 $\text{wc_ins } G \text{ main ins } \rightarrow$
 $\exists T, \text{program_behaves } (\text{Asm.semantics } P) (\text{Reacts } T)$
 $\wedge \text{bisim_IO } G \text{ main ins outs } T.$
and compilation succeeds...



Main Correctness Theorem

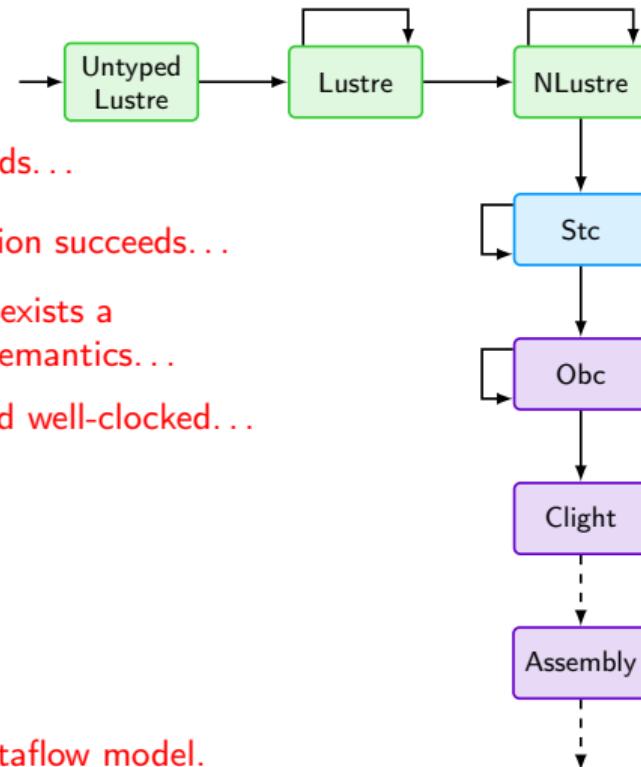


Theorem `behavior_asm`:

$$\begin{aligned}
 & \forall D G Gp P \text{ main ins outs}, \\
 & \text{elab_declarations } D = \text{OK} \text{ (exist } _G Gp) \rightarrow \\
 & \text{compile } D \text{ main } = \text{OK } P \rightarrow \\
 & \text{sem_node } G \text{ main } (pStr \text{ ins}) (pStr \text{ outs}) \rightarrow \\
 & \text{wt_ins } G \text{ main ins } \rightarrow \\
 & \text{wc_ins } G \text{ main ins } \rightarrow \\
 & \quad \left. \right\} \text{ and input streams are well-typed and well-coded...} \\
 & \exists T, \text{program_behaves (Asm.semantics } P) (\text{Reacts } T) \\
 & \quad \wedge \text{bisim_IO } G \text{ main ins outs } T.
 \end{aligned}$$

if typing/elaboration succeeds...
 and compilation succeeds...
 and there exists a
 dataflow semantics...

Main Correctness Theorem



Theorem behavior_asm:

$\forall D G Gp P \text{ main ins outs},$
elab_declarations $D = \text{OK} (\exists G Gp) \rightarrow$
compile $D \text{ main } = \text{OK } P \rightarrow$ and compilation succeeds...
sem_node $G \text{ main } (pStr \text{ ins}) (pStr \text{ outs}) \rightarrow$ and there exists a dataflow semantics...
 $wt_ins G \text{ main ins} \rightarrow$ and input streams are well-typed and well-coded...
 $wc_ins G \text{ main ins} \rightarrow$
 $\exists T, \text{program_behaves} (\text{Asm.semantics } P) (\text{Reacts } T)$
 $\wedge \text{bisim_IO } G \text{ main ins outs } T.$

if typing/elaboration succeeds...

and there exists a dataflow semantics...

then the generated assembly produces an infinite trace

and the trace corresponds to the dataflow model.

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
 - `switch` blocks
 - `reset` blocks
 - state machines
 - `last` variables
- a verified dependency analysis used to prove meta-properties of the model
- a verified implementation of an efficient compilation scheme for these blocks

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
 - `switch` blocks
 - `reset` blocks
 - state machines
 - `last` variables
- a verified dependency analysis used to prove meta-properties of the model
- a verified implementation of an efficient compilation scheme for these blocks

Future work:

- proof automation?
- missing Scade 6 features:
 - inlining and modular dependency analysis
 - `pre` operator and initialization analysis
 - arrays

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
 - `switch` blocks
 - `reset` blocks
 - state machines
 - `last` variables
- a verified dependency analysis used to prove meta-properties of the model
- a verified implementation of an efficient compilation scheme for these blocks

Future work:

- proof automation?
- missing Scade 6 features:
 - inlining and modular dependency analysis
 - `pre` operator and initialization analysis
 - arrays

<https://velus.inria.fr/phd-pesin>

Introduction
○○○

Synchronous Dataflow
○○○○○

The Vélus Compiler
○○○○

Relational Semantics
○○○○○○

Dependency Analysis
○○○

Verified Compilation
○○○○○○○○○○

Conclusion
○○●

Semantics – switch blocks

$$\begin{aligned}\text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs \\ \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) &\equiv \langle v \rangle \cdot \text{when}^C xs cs \\ \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) &\equiv \langle \rangle \cdot \text{when}^C xs cs\end{aligned}$$

$$(\text{when}^C H cs)(x) \equiv \text{when}^C (H(x)) cs$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i} (H, bs) cs \vdash blks_i}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}}$$

Semantics – reset blocks

$$\text{mask}_{k'}^k (F \cdot rs) (sv \cdot xs) \equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'}^k rs xs$$

$$\text{mask}_{k'}^k (T \cdot rs) (sv \cdot xs) \equiv (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'+1}^k rs xs$$

$$\frac{\begin{array}{c} G, H, bs \vdash es \Downarrow xss \\ G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv rs \\ \forall k, G \vdash f(\text{mask}^k rs xss) \Downarrow (\text{mask}^k rs yss) \end{array}}{G, H, bs \vdash (\text{reset } f \text{ every } e)(es) \Downarrow yss}$$

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \\ \text{bools-of } ys \equiv rs \\ \forall k, G, \text{mask}^k rs (H, bs) \vdash blks \end{array}}{G, H, bs \vdash \text{reset } blks \text{ every } e}$$

Semantics – Hierarchical State Machines

$$\frac{H, bs \vdash ck \Downarrow bs' \quad G, H, bs' \vdash_{\text{I}} \text{autinit} \Downarrow sts_0 \quad \text{fby } sts_0 \ sts_1 \equiv sts}{\forall i, \forall k, G, (\text{select}_0^{C_i, k} \ sts (H, bs)), C_i \vdash_{\text{w}} \text{autscope}_i \Downarrow (\text{select}_0^{C_i, k} \ sts \ sts_1)} \\ G, H, bs \vdash \text{automaton initially autinit}^{ck} [\text{state } C_i \text{ autscope}_i]^i \text{ end}$$

$$\frac{\begin{array}{c} \forall x, x \in \text{dom}(H') \iff x \in \text{locs} \\ \forall x e, (\text{last } x = e) \in \text{locs} \implies G, H + H', bs \vdash_{\text{L}} \text{last } x = e \\ G, H + H', bs \vdash \text{blk} \quad G, H + H', bs, C_i \vdash_{\text{TR}} \text{trans} \Downarrow sts \end{array}}{G, H, bs, C_i \vdash_{\text{w}} \text{var locs do blk until trans} \Downarrow sts}$$

$$\frac{\begin{array}{c} H, bs \vdash ck \Downarrow bs' \quad \text{fby } (\text{const } bs' (C, F)) \ sts_1 \equiv sts \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} \ sts (H, bs)), C_i \vdash_{\text{TR}} \text{trans}_i \Downarrow (\text{select}_0^{C_i, k} \ sts \ sts_1) \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} \ sts_1 (H, bs)) \vdash \text{blk}_i; \end{array}}{G, H, bs \vdash \text{automaton initially } C^{ck} [\text{state } C_i \text{ do blk}_i \text{ unless trans}_i]^i \text{ end}}$$

Semantics – Transitions

$$G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs'$$

$$G, H, bs \vdash_{\perp} \text{autunits} \Downarrow sts$$

$$sts' \equiv \text{first-of}_{\text{F}}^C bs' sts$$

$$G, H, bs \vdash_{\perp} C \text{ if } e; \text{autunits} \Downarrow sts'$$

$$sts \equiv \text{const } bs(C, F)$$

$$G, H, bs \vdash_{\perp} \text{otherwise } C \Downarrow sts$$

$$\text{first-of}_r^C (T \cdot bs) (st \cdot sts) \equiv \langle C, r \rangle \cdot \text{first-of}_r^C bs sts$$

$$\text{first-of}_r^C (F \cdot bs) (st \cdot sts) \equiv st \cdot \text{first-of}_r^C bs sts$$

$$sts \equiv \text{const } bs(C_i, F)$$

$$G, H, bs, C_i \vdash_{\text{TR}} \epsilon \Downarrow sts$$

$$G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs'$$

$$G, H, bs, C_i \vdash_{\text{TR}} \text{trans} \Downarrow sts$$

$$sts' \equiv \text{first-of}_{\text{F}}^C bs' sts$$

$$G, H, bs, C_i \vdash_{\text{TR}} \text{if } e \text{ continue } C \text{ trans} \Downarrow sts'$$

$$G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs'$$

$$G, H, bs, C_i \vdash_{\text{TR}} \text{trans} \Downarrow sts$$

$$sts' \equiv \text{first-of}_{\text{T}}^C bs' sts$$

$$G, H, bs, C_i \vdash_{\text{TR}} \text{if } e \text{ then } C \text{ trans} \Downarrow sts'$$

Semantics – local blocks and last variables

$$\frac{H(\text{last } x) \equiv vs}{G, H, bs \vdash \text{last } x \Downarrow [vs]}$$

$$\begin{array}{c} \forall x, x \in \text{dom}(H') \iff x \in \text{locs} \\ \forall x e, (\text{last } x = e) \in \text{locs} \implies G, H + H', bs \vdash_{\text{L}} \text{last } x = e \\ \hline G, H + H', bs \vdash \text{blk} \\ \hline G, H, bs \vdash \text{var } \text{locs let blk tel} \end{array}$$

$$\frac{G, H, bs \vdash e \Downarrow [vs_0] \quad H(x) \equiv vs_1 \quad H(\text{last } x) \equiv \text{fby } vs_0 vs_1}{G, H, bs \vdash_{\text{L}} \text{last } x = e}$$

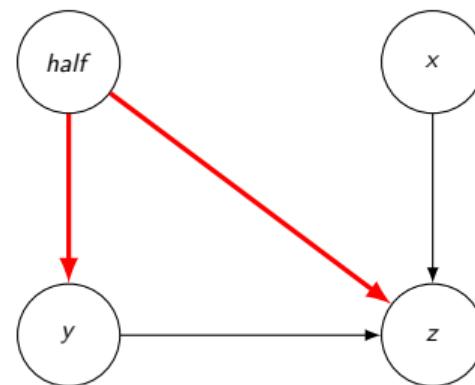
$$(H_1 + H_2)(x) = \begin{cases} H_2(x) & \text{if } x \in H_2 \\ H_1(x) & \text{otherwise.} \end{cases}$$

Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```

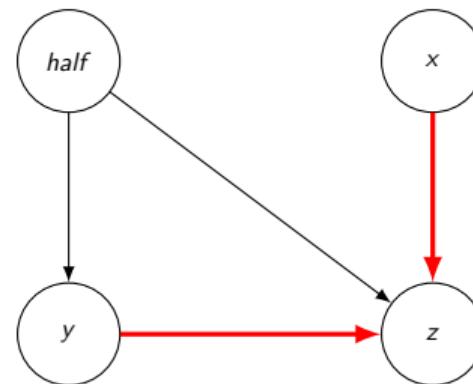
Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



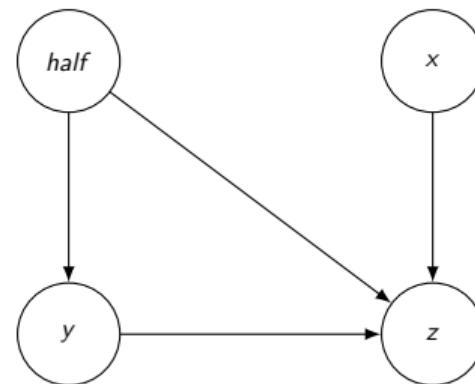
Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



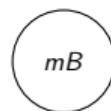
Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : boolx
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



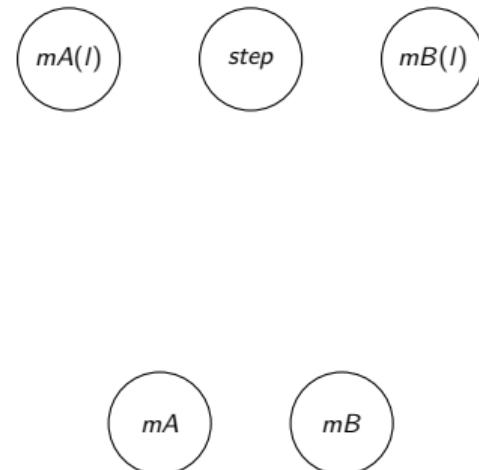
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
tel
```



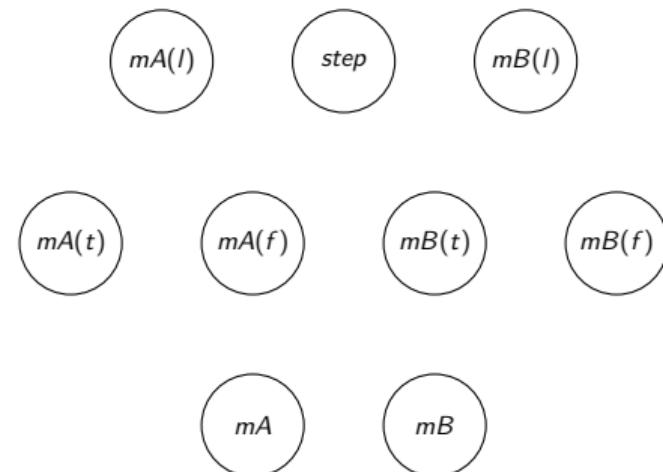
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
end;
last mAmA(I) = true;
last mBmB(I) = false;
tel
```



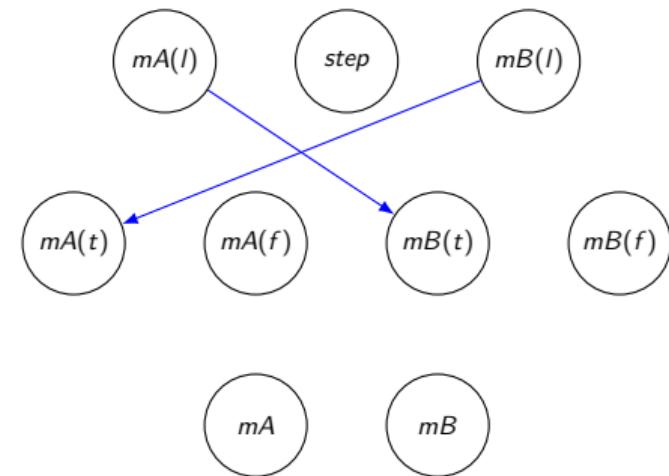
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
```



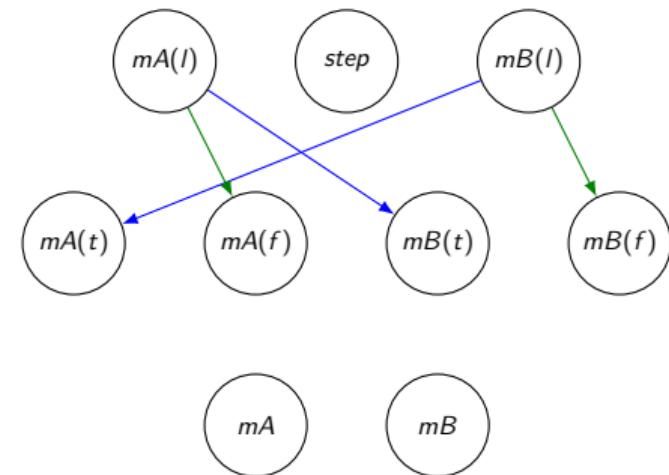
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
```



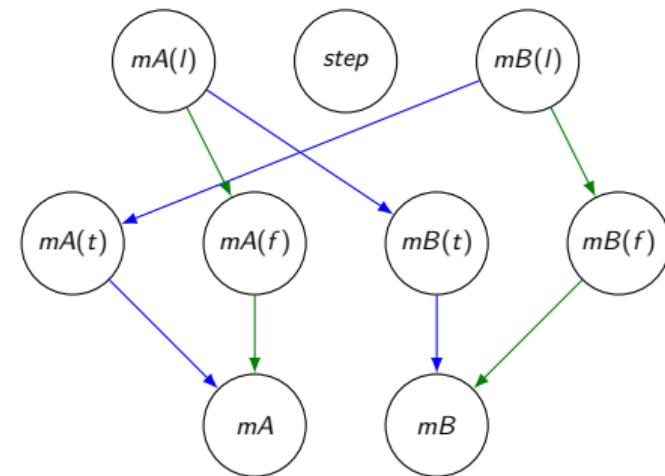
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
```



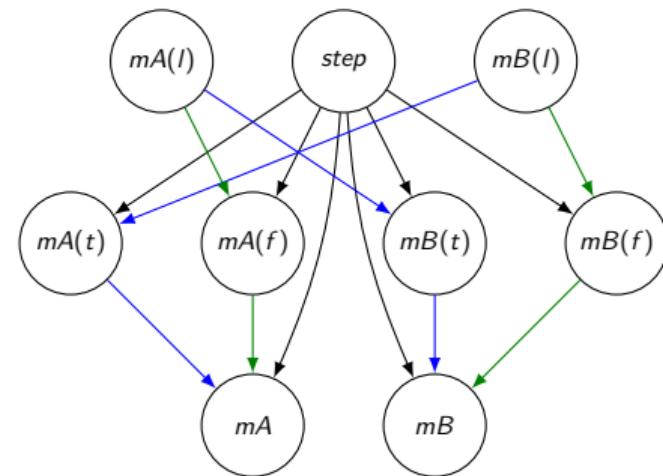
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
```



Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
```



Dependency graph analysis

$$\frac{\text{AcyGraph } \emptyset \emptyset}{\text{AcyGraph } (V \cup \{x\}) E} \qquad \frac{\text{AcyGraph } V E \qquad x, y \in V \qquad y \xrightarrow{E}^* x}{\text{AcyGraph } V(E \cup \{x \rightarrow y\})}$$

- Simple graph analysis, based on DFS
- Produces a witness that the graph is acyclic (AcyGraph) that we will reason on
- More difficult to show termination in Coq

Dependency graph analysis

$$\frac{\text{AcyGraph } \emptyset \emptyset}{\text{AcyGraph } (V \cup \{x\}) E} \qquad \frac{\text{AcyGraph } V E \quad x, y \in V \quad y \xrightarrow{E} x}{\text{AcyGraph } V(E \cup \{x \rightarrow y\})}$$

Definition `visited` (`p : set`) (`v : set`) : `Prop` :=
 $(\forall x, x \in p \rightarrow \neg(x \in v))$
 $\wedge \exists a, \text{AcyGraph } v a$
 $\wedge (\forall x, x \in v \rightarrow \exists zs, \text{graph}(x) = \text{Some } zs$
 $\wedge (\forall y, y \in zs \rightarrow \text{has_arc } a y x)).$

Program `Fixpoint dfs'`
`(s : { p | $\forall x, x \in p \rightarrow x \in \text{graph}$ }) (x : ident)`
`(v : { v | visited s v }) {measure (|graph| - |s|)}`
`: option { v' | visited s v' & $x \in v' \wedge v \subseteq v'$ } := ...`

Dependency graph analysis

$$\frac{}{\text{AcyGraph } \emptyset \emptyset} \quad \frac{\text{AcyGraph } V E}{\text{AcyGraph } (V \cup \{x\}) E} \quad \frac{\text{AcyGraph } V E \quad x, y \in V \quad y \xrightarrow{E}^* x}{\text{AcyGraph } V(E \cup \{x \rightarrow y\})}$$

Definition visited ($p : \text{set}$) ($v : \text{set}$) : Prop :=
 $(\forall x, x \in p \rightarrow \neg(x \in v))$
 $\wedge \exists a, \text{AcyGraph } v a$
 $\wedge (\forall x, x \in v \rightarrow \exists zs, \text{graph}(x) = \text{Some } zs$
 $\wedge (\forall y, y \in zs \rightarrow \text{has_arc } a y x)).$

Program Fixpoint dfs'
($s : \{ p \mid \forall x, x \in p \rightarrow x \in \text{graph} \}$) ($x : \text{ident}$)
($v : \{ v \mid \text{visited } s v \}$) [measure $(|\text{graph}| - |s|)$]
: option { $v' \mid \text{visited } s v' \& x \in v' \wedge v \subseteq v' \}$:= ...

Dependency graph analysis

$$\frac{}{\text{AcyGraph } \emptyset \emptyset} \quad \frac{\text{AcyGraph } V E}{\text{AcyGraph } (V \cup \{x\}) E} \quad \frac{\text{AcyGraph } V E \quad x, y \in V \quad y \xrightarrow{E}^* x}{\text{AcyGraph } V(E \cup \{x \rightarrow y\})}$$

Definition `visited` (`p : set`) (`v : set`) : `Prop` :=
 $(\forall x, x \in p \rightarrow \neg(x \in v))$
 $\wedge \exists a, \text{AcyGraph } v a$
 $\wedge (\forall x, x \in v \rightarrow \exists zs, \text{graph}(x) = \text{Some } zs$
 $\wedge (\forall y, y \in zs \rightarrow \text{has_arc } a y x)).$

Program Fixpoint `dfs'`
 $(s : \{ p \mid \forall x, x \in p \rightarrow x \in \text{graph} \}) (x : \text{ident})$
 $(v : \{ v \mid \text{visited } s v \})$ `[measure (|graph| - |s|)]`
 $: \text{option } \{ v' \mid \text{visited } s v' \& x \in v' \wedge v \subset v' \} := \dots$

Dependency graph analysis

$$\frac{}{\text{AcyGraph } \emptyset \emptyset} \quad \frac{\text{AcyGraph } V E}{\text{AcyGraph } (V \cup \{x\}) E} \quad \frac{\text{AcyGraph } V E \quad x, y \in V \quad y \xrightarrow{E} x^*}{\text{AcyGraph } V(E \cup \{x \rightarrow y\})}$$

Definition `visited` (`p : set`) (`v : set`) : `Prop` :=
 $(\forall x, x \in p \rightarrow \neg(x \in v))$
 $\wedge \exists a, \text{AcyGraph } v a$
 $\wedge (\forall x, x \in v \rightarrow \exists zs, \text{graph}(x) = \text{Some } zs$
 $\wedge (\forall y, y \in zs \rightarrow \text{has_arc } a y x)).$

Program `Fixpoint dfs'`
 $(s : \{ p \mid \forall x, x \in p \rightarrow x \in \text{graph} \}) (x : \text{ident})$
 $(v : \{ v \mid \text{visited } s v \})$ `[measure (|graph| - |s|)]`
 $: \text{option } \{ v' \mid \text{visited } s v' \& x \in v' \wedge v \subset v' \} := \dots$

Proving with dependencies

TopoOrder (AcyGraph $V E$) []

$$\frac{\text{TopoOrder (AcyGraph } V E) / \\ x \in V \quad \neg \text{In } x I \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y I)}{\text{TopoOrder (AcyGraph } V E) (x :: I)}$$

Proving with dependencies

$$\frac{\text{TopoOrder (AcyGraph } V E) / \\ x \in V \quad \neg \text{In } x / \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y /)}{\text{TopoOrder (AcyGraph } V E) (x :: /)}$$

TopoOrder (AcyGraph $V E$) []

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
```

Proving with dependencies

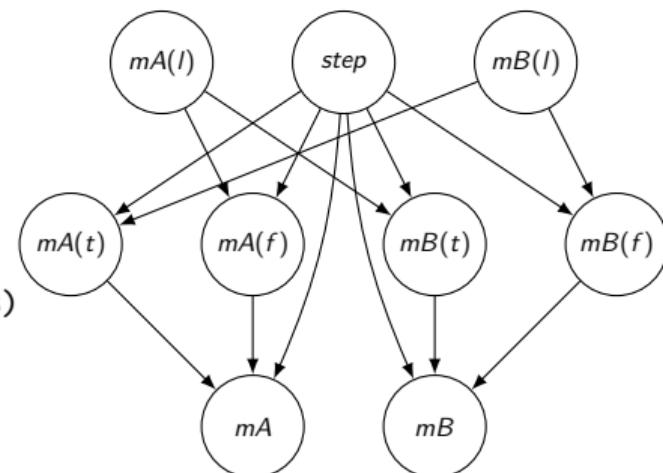
TopoOrder (AcyGraph V E) []

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(I) = true;
last mBmB(I) = false;
tel

```

$$\frac{\text{TopoOrder (AcyGraph } V E \text{) /} \\ x \in V \quad \neg \text{In } x \text{ /} \quad (\forall y, y \xrightarrow{*} E x \implies \text{In } y \text{ /})}{\text{TopoOrder (AcyGraph } V E \text{) } (x :: I)}$$



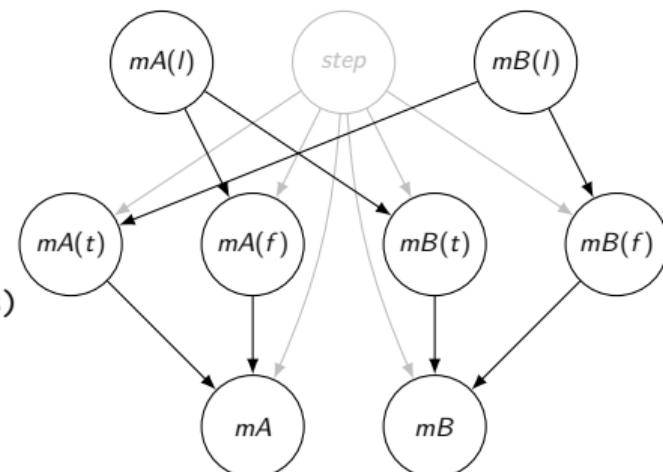
Proving with dependencies

TopoOrder (AcyGraph V E) []

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(I) = true;
last mBmB(I) = false;
tel
  
```

$$\frac{x \in V \quad \neg \text{In } x \text{ I} \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y \text{ I})}{\text{TopoOrder (AcyGraph } V E \text{)} (x :: I)}$$



Proving with dependencies

TopoOrder (AcyGraph V E) []

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
  
```

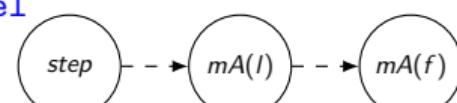
$$\frac{x \in V \quad \neg \text{In } x \text{ I} \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y \text{ I})}{\text{TopoOrder (AcyGraph } V E \text{)} (x :: I)}$$

Proving with dependencies

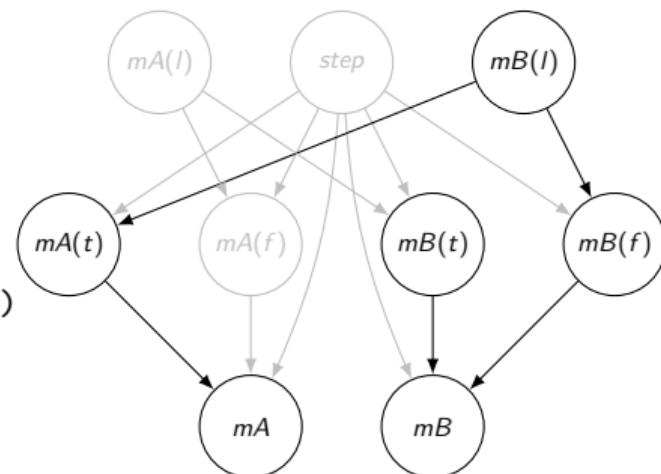
TopoOrder (AcyGraph V E) []

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(l) = true;
last mBmB(l) = false;
tel
  
```



$$\frac{x \in V \quad \neg \text{In } x \text{ I} \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y \text{ I})}{\text{TopoOrder (AcyGraph } V E \text{)} (x :: I)}$$



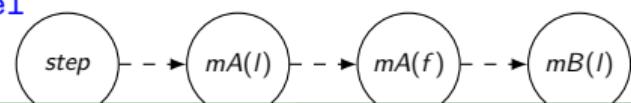
Proving with dependencies

TopoOrder (AcyGraph V E) []

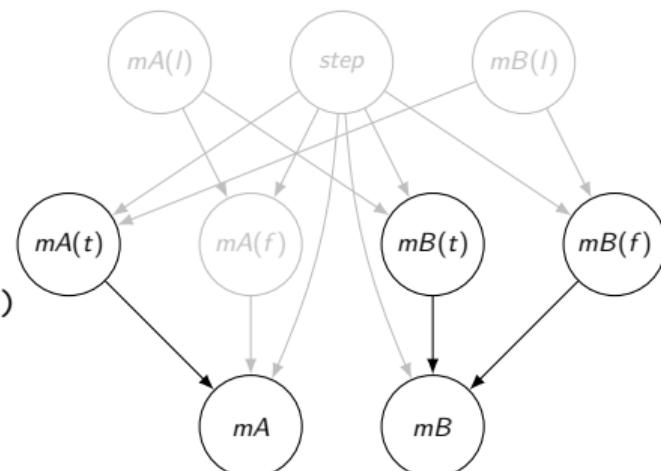
```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
last mAmA(I) = true;
last mBmB(I) = false;
tel

```



$$\frac{x \in V \quad \neg \text{In } x \text{ I} \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y \text{ I})}{\text{TopoOrder (AcyGraph } V E \text{)} (x :: I)}$$



Proving with dependencies

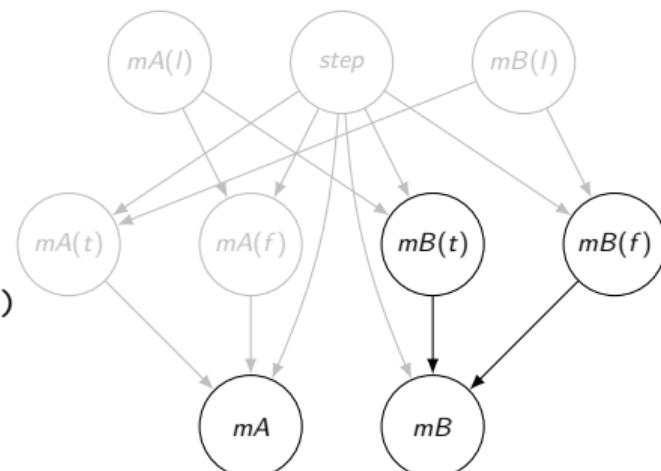
$$\frac{\text{TopoOrder (AcyGraph } V E \text{) []} \quad x \in V \quad \neg \text{In } x \text{ I} \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y \text{ I})}{\text{TopoOrder (AcyGraph } V E \text{) } (x :: I)}$$

TopoOrder (AcyGraph $V E$) []

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
```

```
last mAmA(I) = true;
last mBmB(I) = false;
```

tel



Proving with dependencies

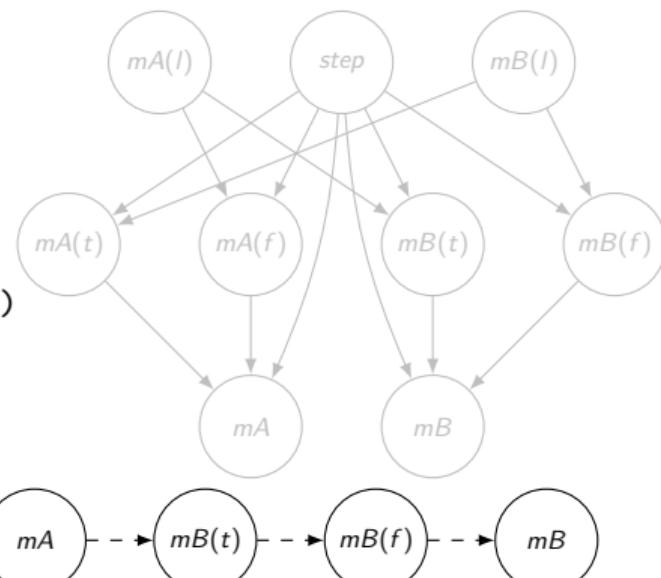
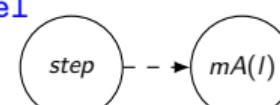
$$\frac{\text{TopoOrder (AcyGraph } V E \text{) []} \quad x \in V \quad \neg \text{In } x \text{ I} \quad (\forall y, y \rightarrow_E^* x \implies \text{In } y \text{ I})}{\text{TopoOrder (AcyGraph } V E \text{) } (x :: I)}$$

TopoOrder (AcyGraph $V E$) []

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mAmA(t) = not (last mB);
    mBmB(t) = last mA;
  | false do (mAmA(f), mBmB(f)) = (last mA, last mB)
end;
```

```
last mAmA(I) = true;
last mBmB(I) = false;
```

tel



Performances

	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcci</i>
stepper_motor	930	1185 (+27 %)	610 (-34 %)	535 (-42 %)
chrono	505	970 (+92 %)	570 (+12 %)	570 (+12 %)
cruisecontrol	1405	1745 (+24 %)	960 (-31 %)	895 (-36 %)
heater	2415	3125 (+29 %)	730 (-69 %)	515 (-78 %)
buttons	1015	1430 (+40 %)	625 (-38 %)	625 (-38 %)
stopwatch	1305	1970 (+50 %)	1290 (-1 %)	1290 (-1 %)

WCET estimated by OTAWA 2 [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

- Vélus generally better than Heptagon, but worse than Heptagon+GCC

Performances

	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcci</i>
stepper_motor	930	1185 (+27 %)	610 (-34 %)	535 (-42 %)
chrono	505	970 (+92 %)	570 (+12 %)	570 (+12 %)
cruisecontrol	1405	1745 (+24 %)	960 (-31 %)	895 (-36 %)
heater	2415	3125 (+29 %)	730 (-69 %)	515 (-78 %)
buttons	1015	1430 (+40 %)	625 (-38 %)	625 (-38 %)
stopwatch	1305	1970 (+50 %)	1290 (-1 %)	1290 (-1 %)

WCET estimated by OTAWA 2 [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

- Vélus generally better than Heptagon, but worse than Heptagon+GCC
- Inlining of CompCert not fine tuned to small functions generated by Vélus

Performances

	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcci</i>
stepper_motor	930	1185 (+27 %)	610 (-34 %)	535 (-42 %)
chrono	505	970 (+92 %)	570 (+12 %)	570 (+12 %)
cruisecontrol	1405	1745 (+24 %)	960 (-31 %)	895 (-36 %)
heater	2415	3125 (+29 %)	730 (-69 %)	515 (-78 %)
buttons	1015	1430 (+40 %)	625 (-38 %)	625 (-38 %)
stopwatch	1305	1970 (+50 %)	1290 (-1 %)	1290 (-1 %)

WCET estimated by OTAWA 2 [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

- Vélus generally better than Heptagon, but worse than Heptagon+GCC
- Inlining of CompCert not fine tuned to small functions generated by Vélus
- Some possible improvements
 - Better compilation of `last` to reduce useless updates (done in unpublished version)
 - Memory optimization: state variables in mutually exclusive states can be reused