

Verified Compilation of Synchronous Dataflow with State Machines

Timothy Bourke

Basile Pesin

Marc Pouzet

Inria Paris

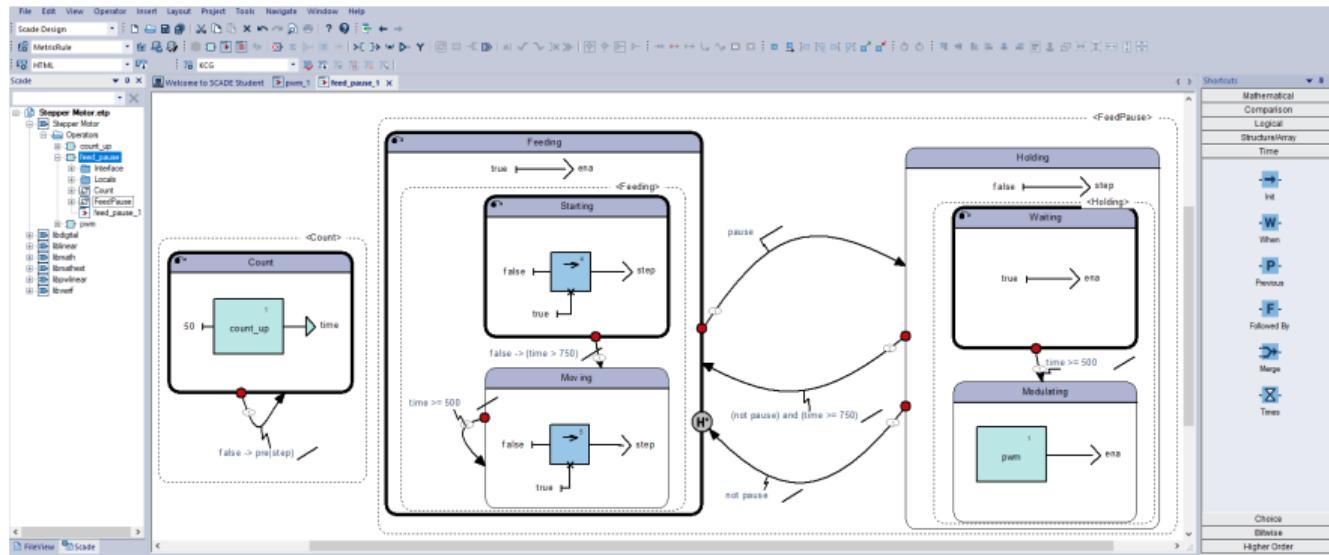
École normale supérieure, CNRS, PSL University

ESWEEK 2023 - EMSOFT

Monday, September 18

11:22am - 11:47am CET

Block-Diagram Languages for Embedded Systems

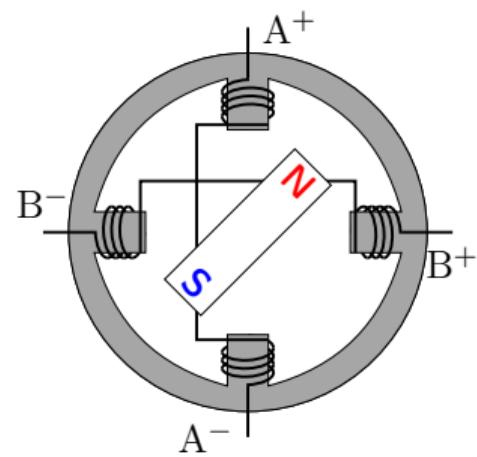
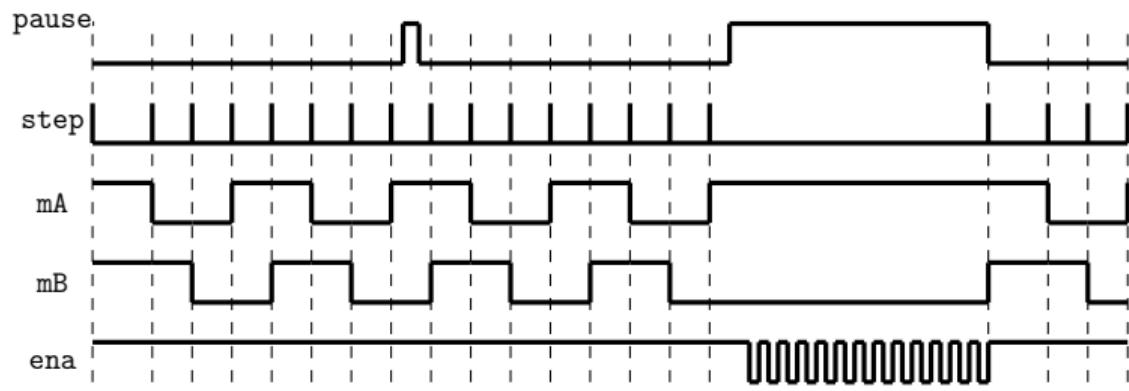


- Widely used in safety-critical applications:
Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear...
- Scade 6: Qualified compiler for Lustre + Control Structures
- Our work: Verified compilation in an Interactive Theorem Prover (Coq)

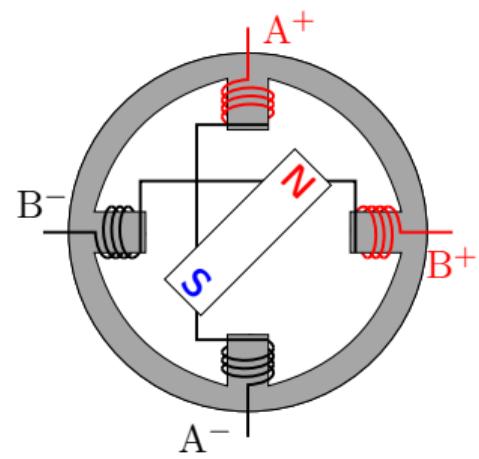
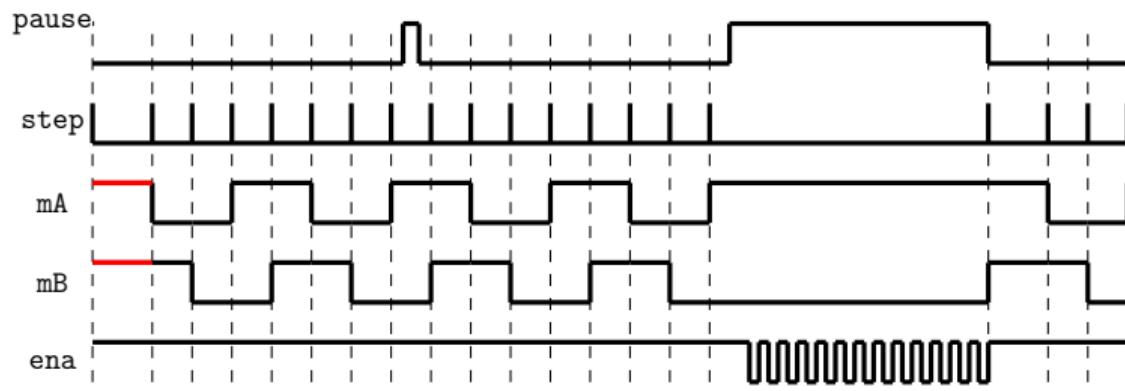
An system example: stepper motor for a small printer



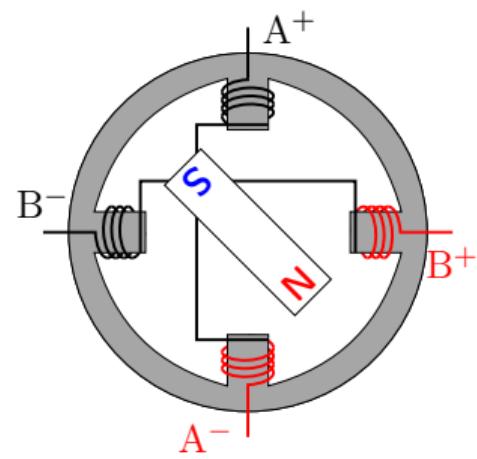
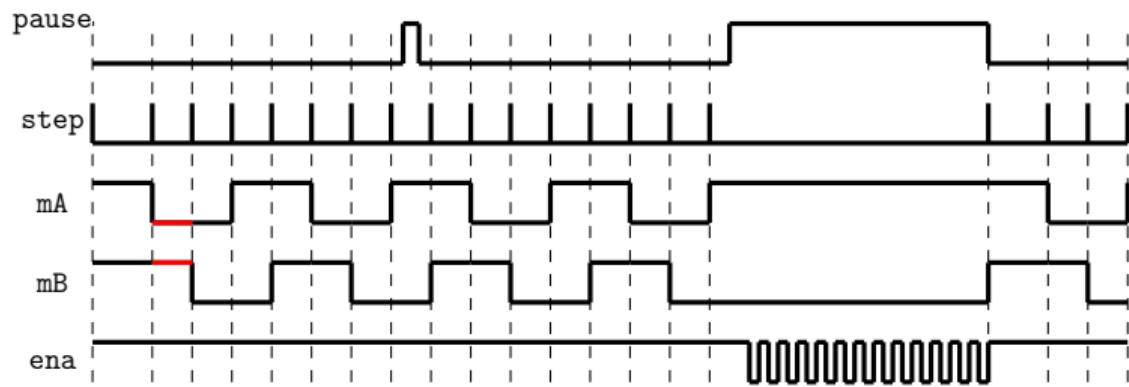
An system example: stepper motor for a small printer



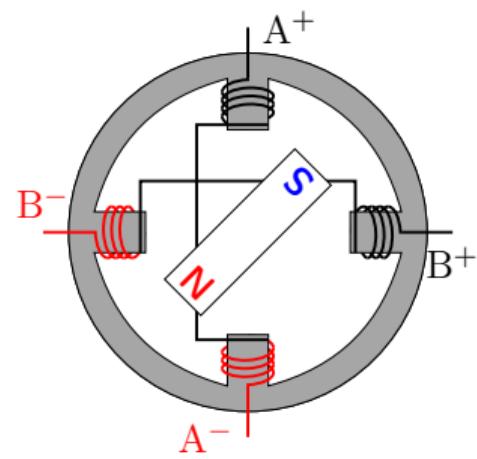
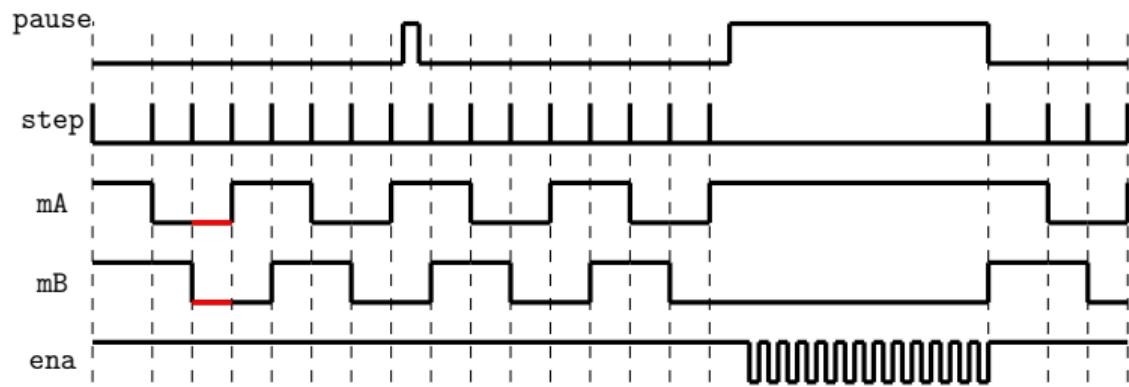
An system example: stepper motor for a small printer



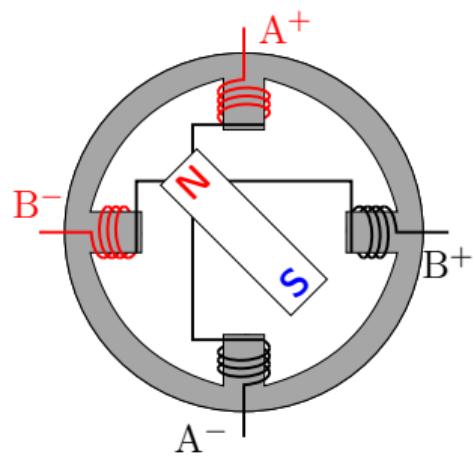
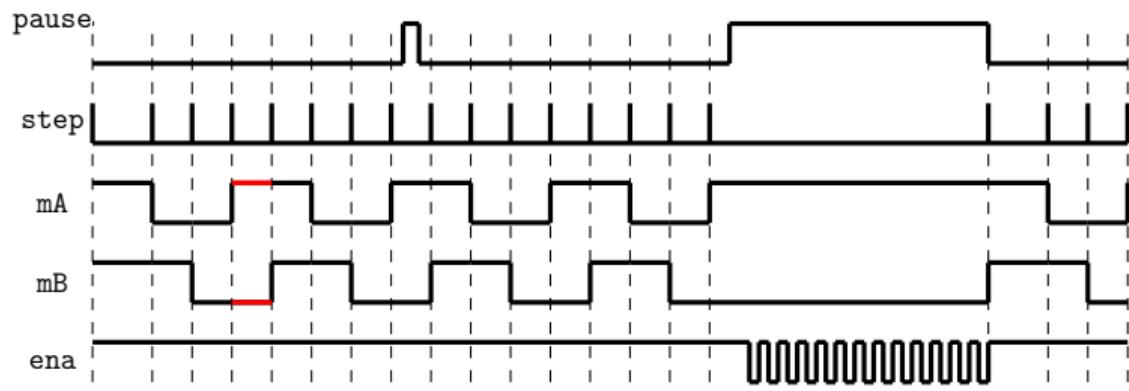
An system example: stepper motor for a small printer



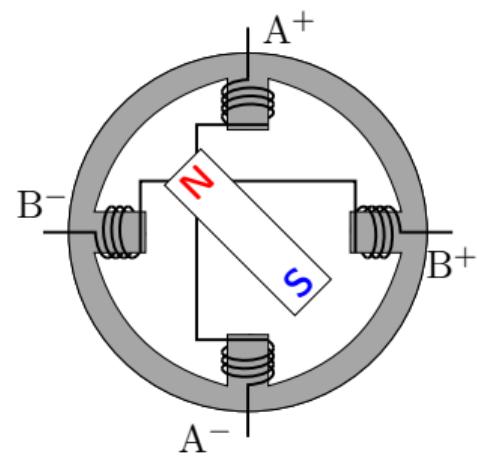
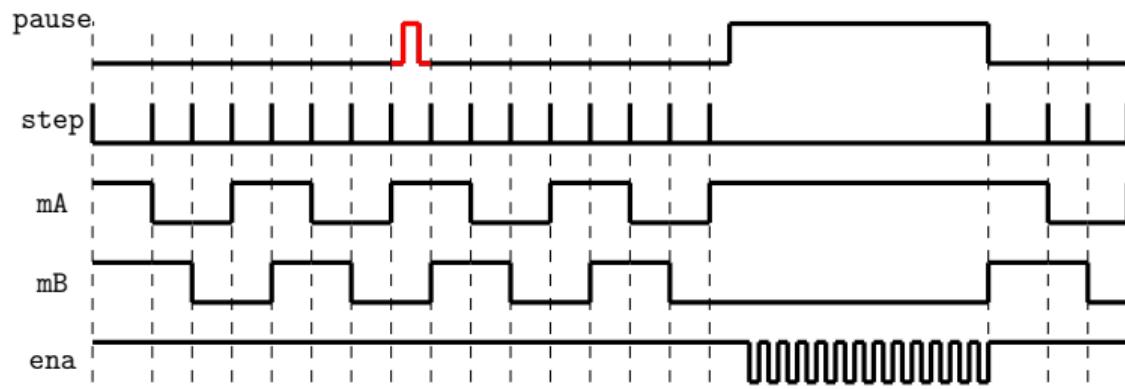
An system example: stepper motor for a small printer



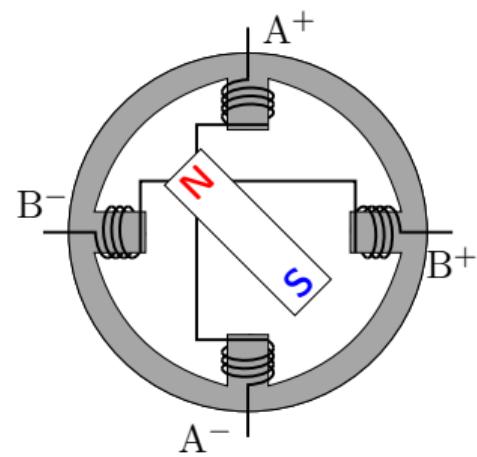
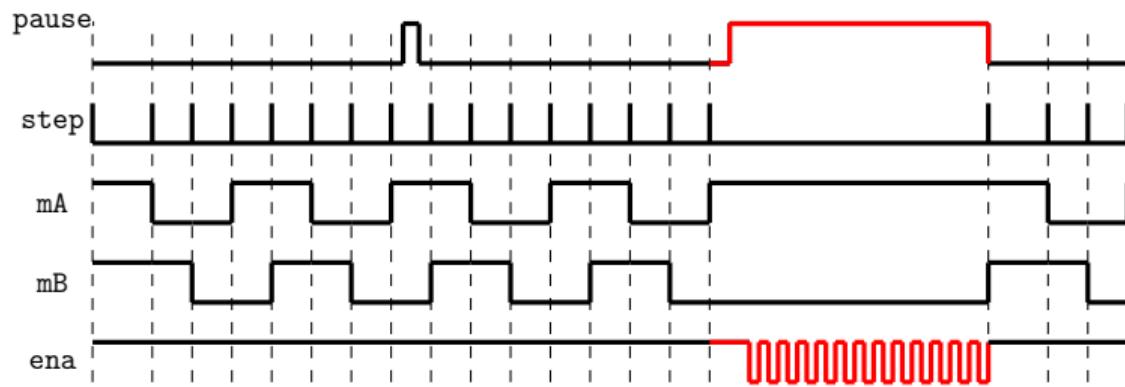
An system example: stepper motor for a small printer



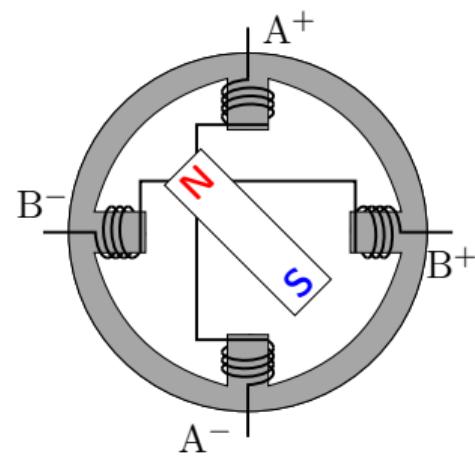
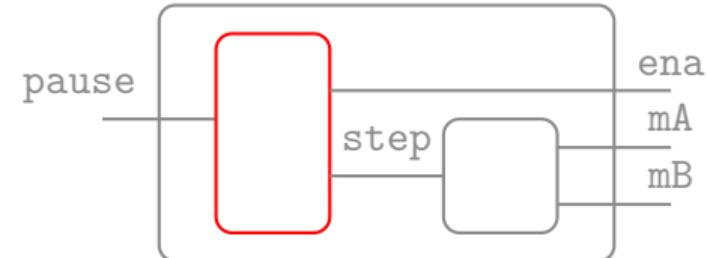
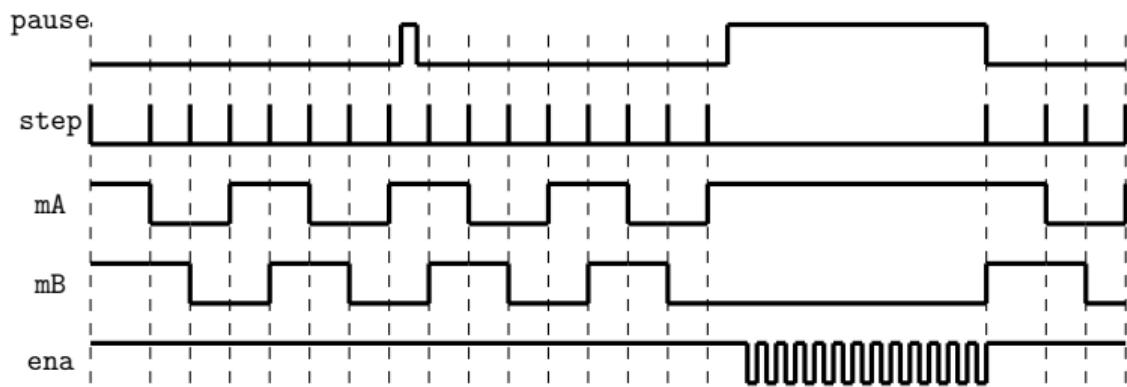
An system example: stepper motor for a small printer



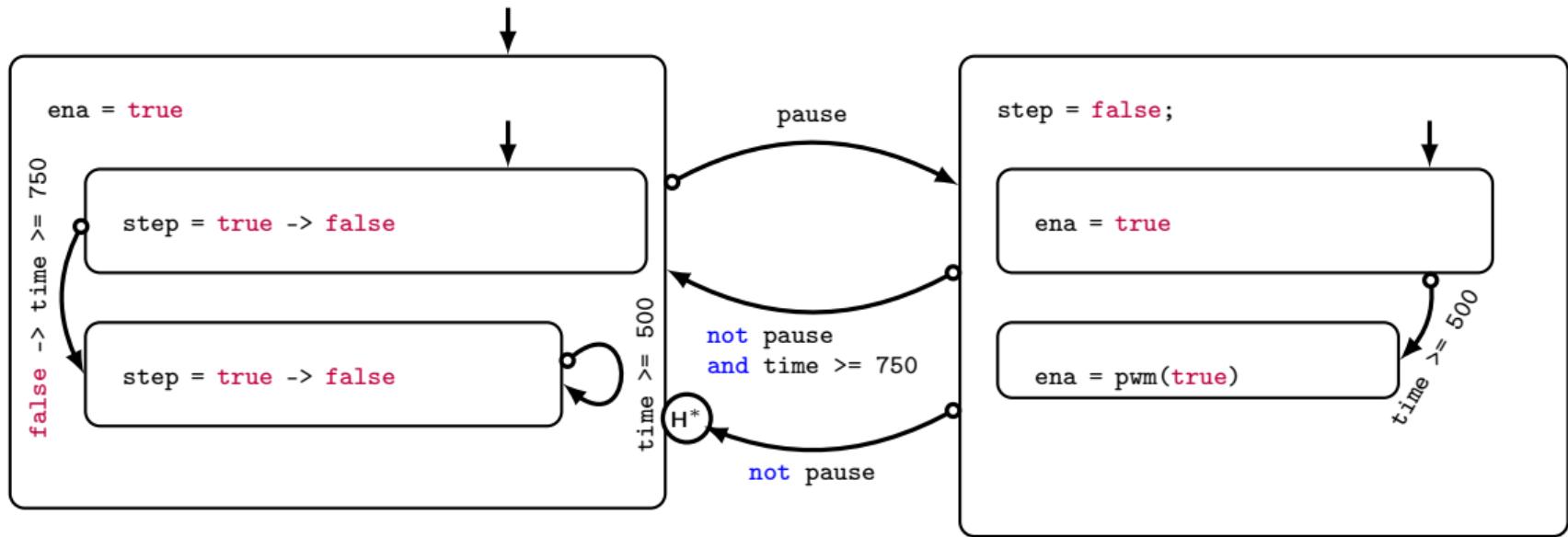
An system example: stepper motor for a small printer



An system example: stepper motor for a small printer



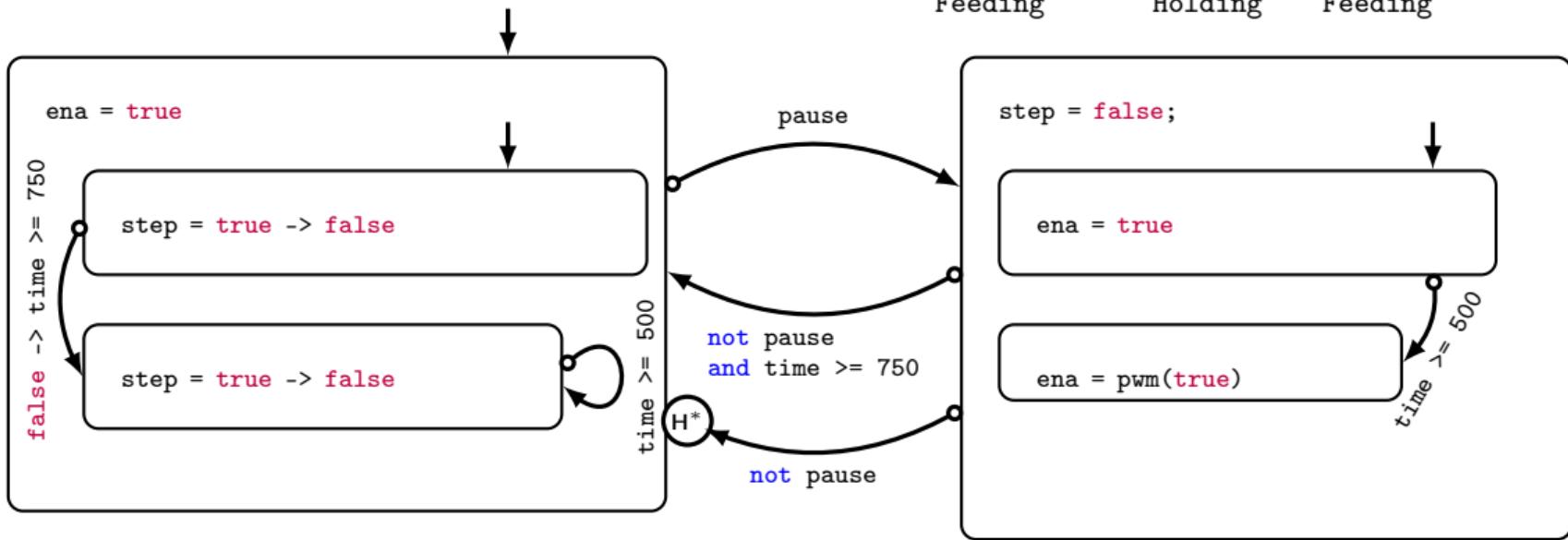
Hierarchical State Machines – Example



Hierarchical State Machines – Example

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 – Example

```

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

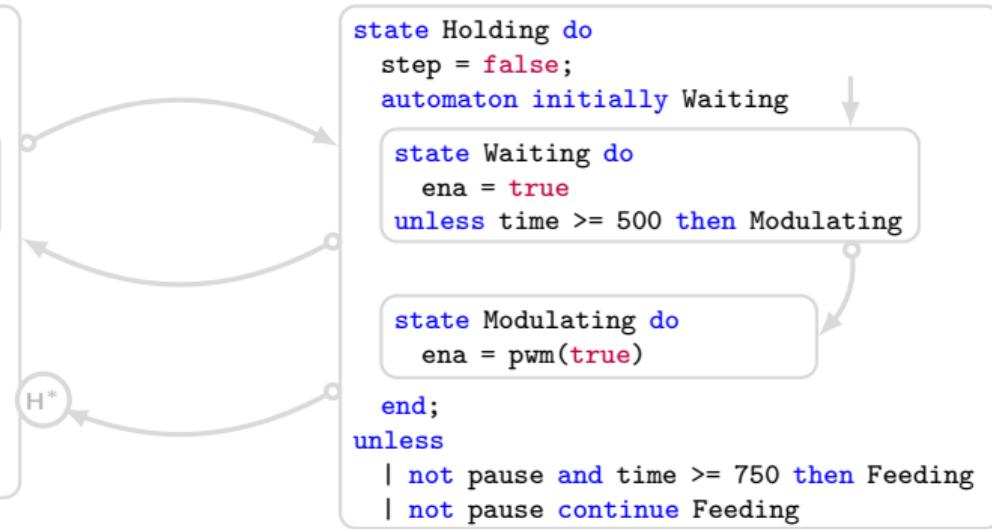
```

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

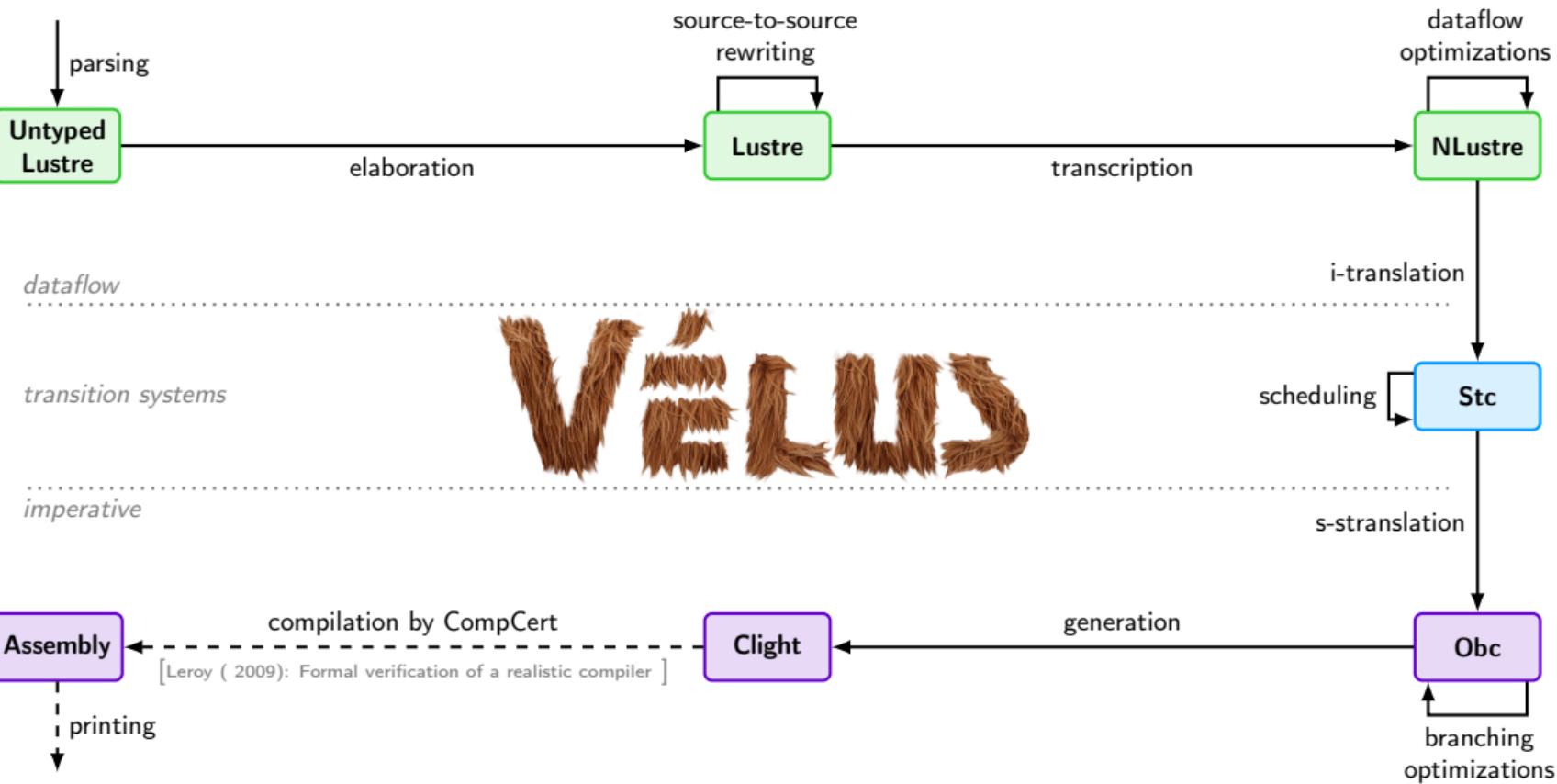
```

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 ... F ... T ...
ena	T T T ... T T ... T ... T ... T ...

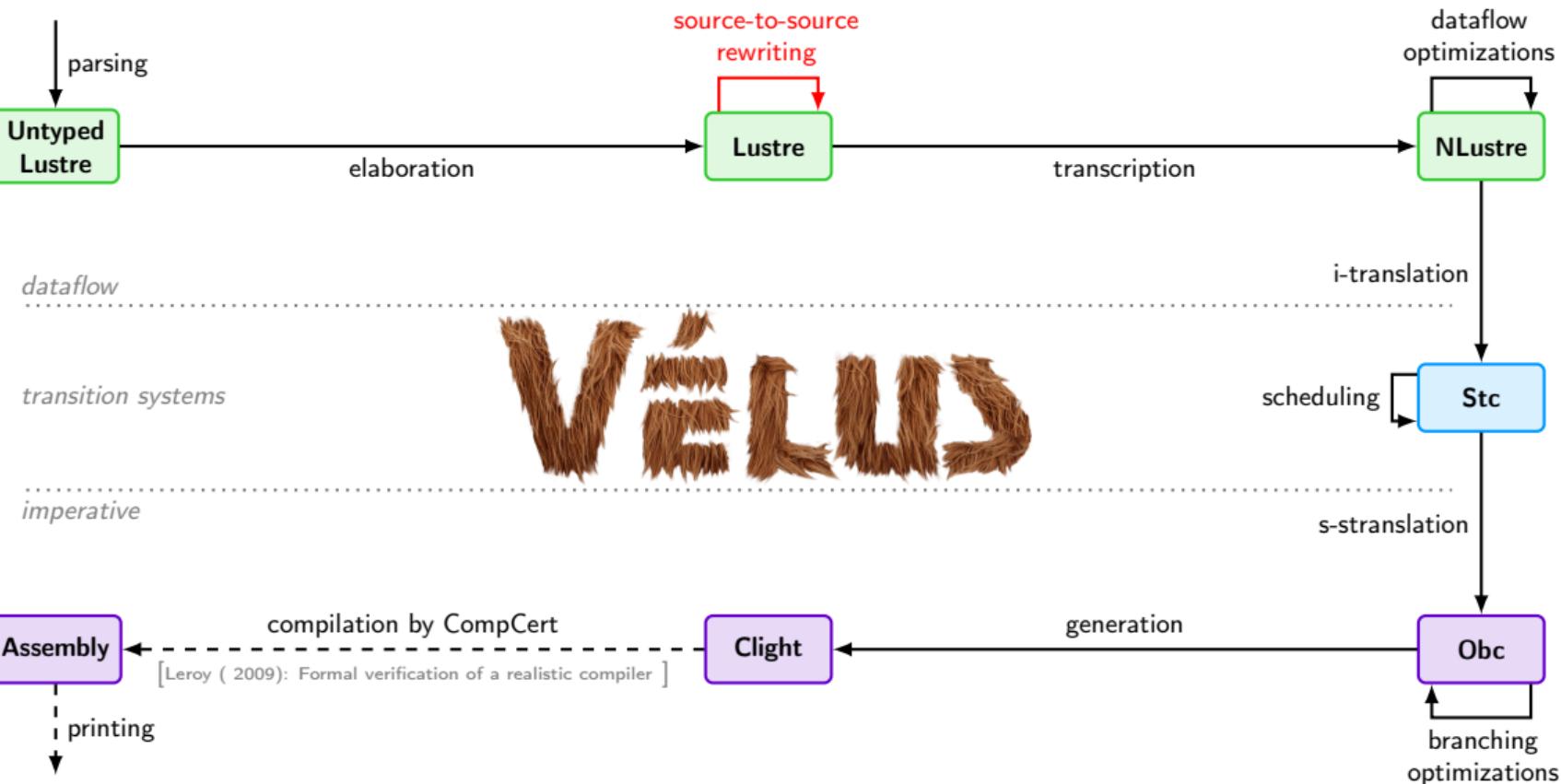
Feeding Holding Feeding



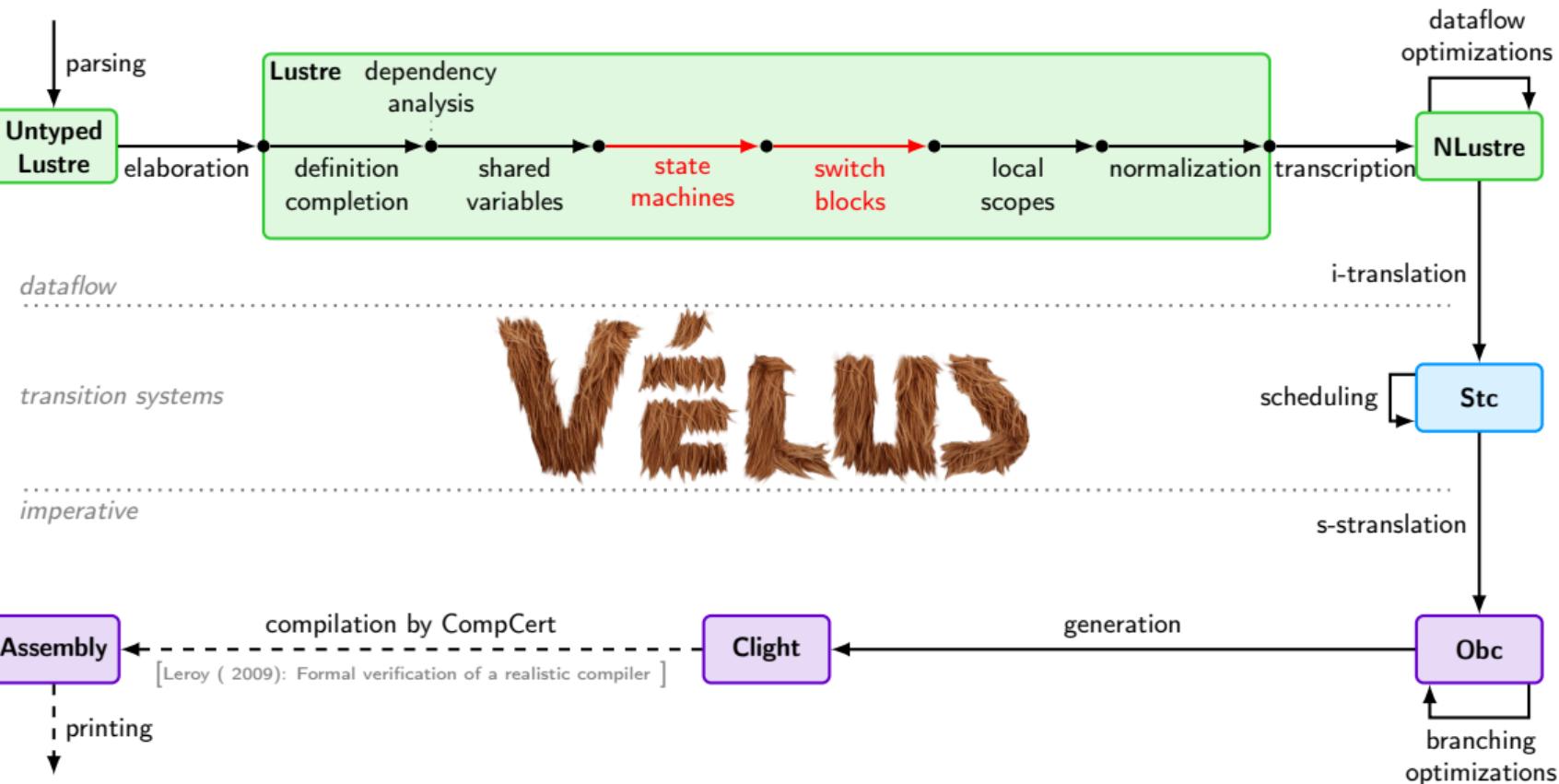
The Vélus Compiler



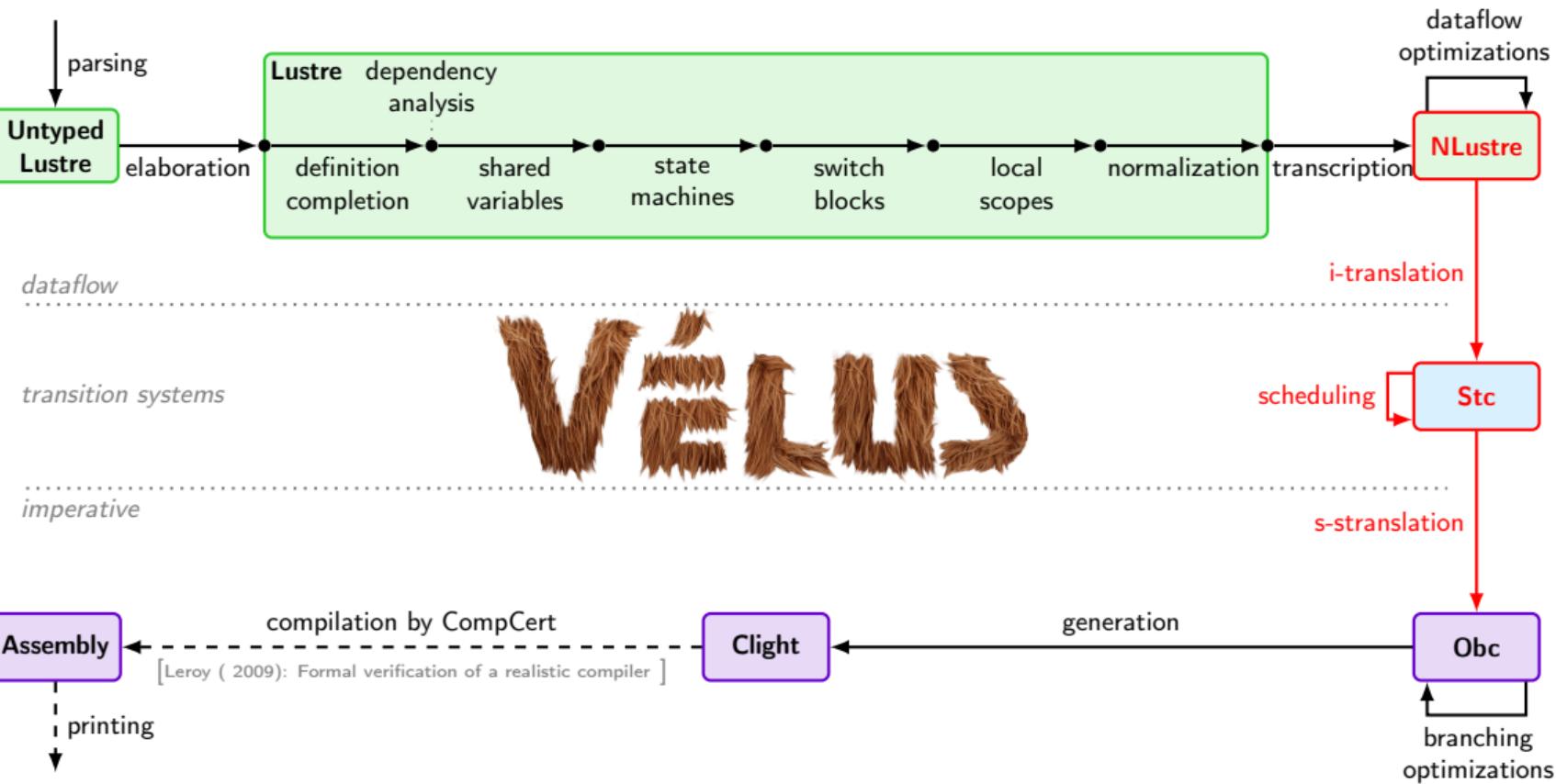
The Vélus Compiler



The Vélus Compiler



The Vélus Compiler



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 -> false
        unless false -> time >= 750 then Moving
    state Moving do
        step = true -> 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 -> false
    unless false -> time >= 750 then Moving
      state Moving do
        step = true -> false
        unless time >= 500 then Moving
          end;
          unless pause then Holding
        end
      end
    end
  end
tel
```



Compilation of state machines

```
automaton initially Starting
```



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

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



```
end
```

Compilation of state machines

automaton initially Starting

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

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

end

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

```
CModd(1) {C1 -> (D1, N1), C2 -> (D2, N2)} =
  D1 and D2 and
  C1 and x = v and
  D1 = p(x) and
  C2 = p(x)
  {C1 -> p(x)C1(x)(D1)}
  and
  p(x) = p(C1(x)(D1))
  and
  p(x) = p(C2(x)(D2))
  and
  p(x) = p(C1(x)(D1)) + p(C2(x)(D2))
  where x <= f(p(x)) and p(D1)
  and {x, ..., p(x)} ⊂ N1 ∪ {v} and
  and {x, ..., p(x)} ⊂ N2 ∪ {v} and D1, C1(x)(D1)
  and {x, ..., p(x)} ⊂ N1 ∪ {v} and D2, C2(x)(D2)}
```

Figure 5: The translation of switch

aline. This code is precompiled into:

```
clock = ? + 1
and up = ? + 1
and down = ? + 1
and all = merge +
  (up <= 1000 and right) +
  (right <= 11 - 1000 and down) +
  (left <= 11 - 1000 and right) +
  (right <= 11 - 1000 and down)
```

This translation highlights the left and right et al. switches

possible to update and leave unchanged during compilation. That is, it is possible to have a switch that is never triggered, and hence always with the same value, or a switch that is always triggered.

2. The System

We should first mention the typical rule for the new programming constructs. In fact, one should update the programming constructs in such a way that it gives the same system as the original one. This is the case of the translation of the switch. The new switch construction is only allowed in a strict world where the old construction is also allowed. If old and new switches are implemented as optimal components, old and new switches can coexist and correctly compute the



Compilation of state machines

automaton initially Starting

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

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

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

Figure 5: The translation of switch

switch st
 S₁ := (D₁, ..., D_n), (C₁ → (D_m, ..., D_p))
 S₂ := (D₁', ..., D_n'), (C₂' → (D_m', ..., D_p'))
 switch st = S₁ do
 D₁ := p₁(C₁) and
 D₂ := p₂(C₁)
 ...
 D_n := p_n(C₁)
 and
 D_m := p_m(C₂) and
 D_{n+1} := p_{n+1}(C₂)
 ...
 D_p := p_p(C₂)
 end;
 switch st = S₂ do
 D₁' := p₁(C₂) and
 D₂' := p₂(C₂)
 ...
 D_n' := p_n(C₂)
 and
 D_{m'} := p_{m'}(C₁) and
 D_{n+1'} := p_{n+1}(C₁)
 end;
 and
 (S₁ → p₁(C₁) and ... and p_n(C₁) and D_m → p_m(C₂) and ... and p_p(C₂))
 and
 (S₂ → p₁(C₂) and ... and p_n(C₂) and D_{m'} → p_{m'}(C₁) and ... and p_{n+1'}(C₁))

Figure 5: The translation of switch

possible to enable and disable transitions. This is a key difference with the SyncChart or SystemC/SysML, and largely motivates the need for a conservative extension of synchronous data-flow.

2.1. The System

We should first explain the type rule for the new programming constructs. In fact, one should update the programming constructs in such a way that it gives the system type safety. The type system of the compiler is based on the typing of the translation. These types are only obtained in a strict world where no mutation is allowed. For example, if a function is implemented as a global computation, it will not be able to compute automatically because the

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

Compilation of state machines

automaton initially Starting

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

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

end
```

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

CMCode₁($\{C_1 \rightarrow (D_1, N_1), C_2 \rightarrow (D_2, N_2)\}$) =
 $\frac{D_1 \text{ and } D_2 \text{ and } \exists x \text{ such that } C_1 \rightarrow x \text{ and } C_2 \rightarrow x}{\{C_1 \rightarrow p(C_1 \rightarrow p(C_2 \rightarrow p(\ldots)))\}}$
and
 $\frac{D_1 \text{ and } D_2 \text{ and } \exists x \text{ such that } C_1 \rightarrow x \text{ and } C_2 \rightarrow x}{\{C_1 \rightarrow p(C_1 \rightarrow p(C_2 \rightarrow p(\ldots)))\}}$
and
 $\frac{D_1 \text{ and } D_2 \text{ and } \exists x \text{ such that } C_1 \rightarrow x \text{ and } C_2 \rightarrow x}{\{C_1 \rightarrow p(C_1 \rightarrow p(C_2 \rightarrow p(\ldots)))\}}$
where $\forall i, x \notin f(c_i)$ and $f(D_i)$
and $\{x \rightarrow p(x) : N_1 \cup \{N_1\} \cup N_2 \cup \{N_2\}\}$
and $\{f(D_1) \cup \{f(D_1)\} \cup f(D_2) \cup \{f(D_2)\}\}$

Figure 5: The translation of switch

This code is precompiled into:
clock = ? * n
n = 1 -> (true, n); n = 2 -> (true, 2); n = 3 -> (true, 3)
and n = 4 -> (true, 4); n = 5 -> (true, 5); n = 6 -> (true, 6)
and n = 7 -> (true, 7); n = 8 -> (true, 8); n = 9 -> (true, 9)
n = 10 -> (true, 10); n = 11 -> (true, 11); n = 12 -> (true, 12)

This translation highlights the fact that state est in the
switch condition is the last value of est whenever

CMDataflow₁: $S_1 \rightarrow (D_1, N_1), S_2 \rightarrow (D_2, N_2) \vdash \neg$
switch pres with
 $\frac{S_1 \rightarrow p(S_1 \rightarrow p(S_2 \rightarrow p(\ldots)))) \text{ and } D_1 \text{ always } p(D_1)}$
 $\frac{S_1 \rightarrow p(S_1 \rightarrow p(S_2 \rightarrow p(\ldots)))) \text{ and } D_2 \text{ always } p(D_2)}$
switch x with
 $\frac{S_1 \rightarrow p(S_1 \rightarrow p(S_2 \rightarrow p(\ldots)))) \text{ and } D_1 \text{ always } p(D_1) \text{ and } D_2 \text{ always } p(D_2)}$
 $\frac{S_1 \rightarrow p(S_1 \rightarrow p(S_2 \rightarrow p(\ldots)))) \text{ and } D_2 \text{ always } p(D_2) \text{ and } D_1 \text{ always } p(D_1)}$
and
 $\frac{S_1 \rightarrow p(S_1 \rightarrow p(S_2 \rightarrow p(\ldots)))) \text{ and } f(D_1) \text{ always } p(f(D_1)) \text{ and } f(D_2) \text{ always } p(f(D_2))}{S_1 \rightarrow p(S_1 \rightarrow p(S_2 \rightarrow p(\ldots))))}$

possible to update and leave unchanged the clock estimation that
is to say, the last value of est. This is a key difference
with the Sync-CASL or SyncHDL compiler, and largely
depends on the programmatic nature of the system.

2. The System
We should first explain the type rule for the new pro-
gramming constructs. In fact, one should update the
grammatical rules such that it gives the static types of
the typed version of the translation. These types are obvi-
ously needed for the compilation of the code. The original
rule would be something like “any expression is typed” but
this is too general. Instead, we implement a “fold” rule
which can be used to compile automatically complex expres-

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

Compilation of state machines

automaton initially Starting

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

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

end
```

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

Figure 5: The translation of switch

This translation highlights the left part of state st in the transition rule by the following value of res:

Figure 6: The translation of switch

This translation highlights the left part of state st in the transition rule by the following value of res:

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

Compilation of state machines

automaton initially Starting

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

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

end
```

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

Figure 5: The translation of switch

Figure 6: The translation of switch

This code is precompiled into:

clock = ? * n
n := 1 -> (true, n);
n := 2 -> (false, n);
n := 3 -> (true, n);
n := 4 -> (false, n);
n := 5 -> (true, n);
n := 6 -> (false, n);
n := 7 -> (true, n);
n := 8 -> (false, n);
n := 9 -> (true, n);
n := 10 -> (false, n);
n := 11 -> (true, n);
n := 12 -> (false, n);

This translation highlights the fact that the value of n is used in the comparison right in the condition of the if statement.

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

Compilation of state machines

automaton initially Starting

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

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

end

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

CModd₁($\{C_1 \rightarrow (D_1, N_1), C_2 \rightarrow (D_2, N_2)\}$) =
 $D_1 \rightarrow \text{and } D_2 \text{ and}$
 $x_1 \rightarrow \text{and } x_2 \text{ and}$
 $D_1 \rightarrow \text{and } D_2 \text{ and}$
 $x_1 \rightarrow \text{and } x_2 \text{ and}$
 $(C_1 \rightarrow p(C_1))$
 $(C_1 \rightarrow p(C_1))$
 $(C_1 \rightarrow p(C_1))$
 $\text{and } \text{and }$
 $y_1 \rightarrow \text{and } y_2 \text{ and}$
 $(C_2 \rightarrow p(C_2))$
 $(C_2 \rightarrow p(C_2))$
 $(C_2 \rightarrow p(C_2))$
where $y_1 \neq f(x_1)$ and $y_2 \neq f(x_2)$
and $\{x_1 \rightarrow y_1\} \subseteq N_1 \cup \{y_1 \rightarrow D_1\}$
and $\{x_2 \rightarrow y_2\} \subseteq N_2 \cup \{y_2 \rightarrow D_2\}$

Figure 5: The translation of switch

This translation highlights the left part of state est in the program right to the assignment value of est

CModd₂($S_1 \rightarrow (D_1, N_1), S_2 \rightarrow (D_2, N_2)$) =
 $D_1 \rightarrow \text{and } D_2 \text{ and}$
 $x_1 \rightarrow \text{and } x_2 \text{ and}$
 $D_1 \rightarrow \text{and } D_2 \text{ and}$
 $x_1 \rightarrow \text{and } x_2 \text{ and}$
 $(S_1 \rightarrow p(S_1))$
 $(S_1 \rightarrow p(S_1))$
 $(S_1 \rightarrow p(S_1))$
 $\text{and } \text{and }$
 $y_1 \rightarrow \text{and } y_2 \text{ and}$
 $(S_2 \rightarrow p(S_2))$
 $(S_2 \rightarrow p(S_2))$
 $(S_2 \rightarrow p(S_2))$
where $y_1 \neq f(x_1)$ and $y_2 \neq f(x_2)$
and $\{x_1 \rightarrow y_1\} \subseteq N_1 \cup \{y_1 \rightarrow D_1\}$
and $\{x_2 \rightarrow y_2\} \subseteq N_2 \cup \{y_2 \rightarrow D_2\}$

Figure 6: The translation of switch

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

Compilation of state machines

automaton initially Starting

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

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

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

CMatch (x) ($C_1 \rightarrow (D_1, N_1)$, ..., $(C_n \rightarrow (D_n, N_n))$) =
 $\frac{D_1 \text{ and } ... \text{ and } D_n \text{ and}}{C_{1,2} \rightarrow C \text{ and } C_{1,2} \rightarrow C}$
 $y_1 = \text{match } x$
 $\quad \text{with } C_1 \rightarrow p \text{ of } C_1(x) \cup \{G_1\}$
 $\quad \dots$
 $\quad \text{with } C_n \rightarrow p \text{ of } C_n(x) \cup \{G_n\}$
 $\quad \text{and } ... \text{ and }$
 $\quad y_1 = \text{match } x$
 $\quad \text{with } C_1 \rightarrow p \text{ of } C_1(x) \cup \{G_1\}$

where $\pi_0(x) \neq \{x\} \cup f_D(D_i)$
 and $\{\pi_1, \dots, \pi_k\} = N_1 \cup \dots \cup N_n$
 and $\pi_0(D_i, G_i) = 2\pi_0(\pi_i(CDn, D_i, C(x)))$

This translation highlights the fact that `last` is always greater than or equal to the previous value of `st`.

possible to refine and lower strength deductions practice; that is, to run more tests, to run longer. This is a very large and difficult problem. We can't CLAIM or SYSTEMatically analyze all possible program understanding and analysis.

3.2 The Type System

We should first attend the typing rule for the new programming construct. The type system should assign types to the objects of the language. These also stay in memory so that selected deductions can always be applied to them. They are implemented as system-level constructs.

The type system is a complex semantically complex

```

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

```

Compilation of switch blocks

```
switch st
| Starting do
  reset
    step = true -> 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) -> false when (st=Starting)
every res;
```

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

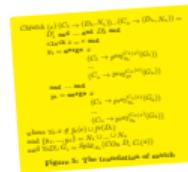


Figure 4: The compilation of switch

above. This code is precompiled into:

```
clock = 1;
if (x == 0) {
  y = 1;
  z = 2;
} else if (x == 1) {
  y = 2;
  z = 1;
} else if (x == 2) {
  y = 3;
  z = 4;
}
return y + z;
```

This translation highlights the fact that state, st in the original code is now replaced by a time signal

possible to order and hence change local computation, that is, to implement local computation. This is a key difference between the Sync-Cauer or SyncHDL compilers, and largely absent in the synchronous program compilation systems.

2.2. The System

We should first explain the type rule for the new programming constructs. In fact, one should update the grammar construct, such that it gives the static types an explicit meaning. The following figure illustrates the typing of the translation. These rules are very similar to the standard typing rules, but they are often altered in a way that reflects the semantics of the language. For example, loops are implemented as explicit tail-recursive functions, so a compiler automatically computes the

- sampling elicited by `when`

Compilation of switch blocks

```
switch st
| Starting do
  reset
    step = true -> 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) -> false when (st=Starting)
every resS;

```

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

CMatch (x) $\langle G \rangle = \langle D_1, N_1 \rangle, \langle G' \rangle = \langle D_2, N_2 \rangle$ if
 D_1 and D_2 and
 $N_1 \sqsubseteq N_2$ and
 $x \in \text{range } x$
 $\text{range } x \rightarrow \text{range}^{(x \rightarrow x)}(G\langle x \rangle)$
 \vdots
 $\text{range } x \rightarrow \text{range}^{(\text{range } x)}(G\langle x \rangle)$
 and
 $y = \text{match } x$
 $\quad \text{with } y \in \text{range } x$
 $\quad \quad \quad \rightarrow \text{range}^{(y \rightarrow y)}(G\langle y \rangle)$
 \vdots
 $\text{range } x \rightarrow \text{range}^{(\text{range } x)}(G\langle x \rangle)$
 where $\langle y \rangle = \langle D_1, N_1 \rangle, \langle G \rangle = \langle D_2, N_2 \rangle$
 and $\langle D_1, N_1 \rangle \sqsubseteq \langle D_2, N_2 \rangle$
 $\text{and } \langle D_1, N_1 \rangle, \langle G \rangle = \langle D_2, N_2 \rangle, \langle G \rangle$

ables. This code is presented in

and $\text{opt} = 1 \rightarrow$ Optimal when $\text{LeditAll} = 1$
 and $\text{opt} = 2 \rightarrow$ Optimal when $\text{LeditAll} = 2$
 and $\text{opt} = \text{merge} \rightarrow$ Optimal when $\text{LeditAll} = 3$

This translation highlights the fact that `last st` in the source program refers to the previous value of `st`, whereas `last st` in the target code refers to the last value of `st` since `st` was equal

possible to solve and leave strongly desired one question, that is, to prove some time after its publication. This is a very difficult task, especially with the SYNTACTIC or SEMANTIC, and having difficulties problem understanding and analysis.

- sampling explicated by `when`
 - choice explicated by `merge`

Compilation of switch blocks

```
switch st
| Starting do
  reset
    step = true -> 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) -> false when (st=Starting)
every resS;
```

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

CMsink (α) (\mathcal{G}) \rightarrow (D_0, N_0), $\{\mathcal{G}'\} \rightarrow$ (D_0, N_0) \in
 β -closed and
 β -closed $x =$ and
 $y =$ where
 $\beta =$ $\text{setpred } x$
 $\quad \cup \{x \mapsto g^{\text{pred}}(\beta)(x)\}$
 $\quad \cup \{y \mapsto g^{\text{pred}}(\beta)(y)\}$
 $\text{and } \beta =$
 $\beta = \text{setpred } x$
 $\quad \cup \{x \mapsto g^{\text{pred}}(\beta)(x)\}$
 $\quad \cup \{y \mapsto g^{\text{pred}}(\beta)(y)\}$
 $\text{where } \alpha \neq \emptyset \text{ and } \beta \in \text{pred}(D)$
 $\text{and } \{\mathcal{G}'\} = N_0 \cup \dots \cup N_k$
 $\text{and } (\mathcal{G}', D, G) = \text{pred}(\mathcal{G}, D, G, C(\mathcal{G}))$

able. This code is translated into

```

    class = " "
    and opt = 1 -> ((pre n0) when Left(n1)) + 1
    and n1 = merge = (Left -> 2 * opt) (Right -> 0)
    and n2 = merge =
        (Left -> (pre n2) when Right(n3))

```

This translation highlights the fact that last π_1 in the source program refers to the previous value of π_1 whereas the last π_1 in the target program has since x was equal

(Assumption) $S_1 = \{D_1, \dots, D_n\}$, D_1^*, \dots, D_n^* , \dots
 $S_2 = \{D_{n+1}, \dots, D_m\}$, D_{n+1}^*, \dots, D_m^* ,
 which plus with
 $S_3 = \{D_{m+1}, \dots, D_{m+n}\}$ and $D_{m+1}^*, \dots, D_{m+n}^*$ where $m \neq n$
 $\exists_{i \in S_1} \text{ such that } x = e_i^* \text{ and } y = e_i^* \text{ and } D_i \text{ where } y \in D_i$
 and
 which plus with
 $S_4 = \{D_{m+n+1}, \dots, D_{m+n+m}\}$ and $D_{m+n+1}^*, \dots, D_{m+n+m}^*$
 $\exists_{i \in S_4} \text{ such that } x = e_i^* \text{ and } y = e_i^* \text{ and } D_i \text{ where } y \in D_i$
 $\exists_{i \in S_1} \text{ such that } x = e_i^* \text{ and } y = e_i^* \text{ and } D_i \text{ where } y \in D_i$
 and
 which plus with
 $S_5 = \{D_{m+n+m+1}, \dots, D_{m+n+m+n}\}$ and $D_{m+n+m+1}^*, \dots, D_{m+n+m+n}^*$
 $\exists_{i \in S_5} \text{ such that } x = e_i^* \text{ and } y = e_i^* \text{ and } D_i \text{ where } y \in D_i$
 and
 $\text{clock_pos} = \text{none}$ and $\text{clock_val} = \text{none}$ and D_1 exist
 and
 $\text{clock_pos} = \text{none}$ and $\text{clock_val} = \text{none}$ and D_2 exist
 and
 $\text{clock_pos} = \text{none}$ and $\text{clock_val} = \text{none}$ and D_3 exist
 where $x, y, n, m, i \in \{1, \dots, m+n\}$ and
 $FV(x) \cup FV(y) \subseteq FV(e)$
 $\cup FV(e^*) \cup FV(e_i)$
 $\cup FV(e_i^*)$

possible to infer and leave strongly diagnostic traces. That is, it can trace more than n transitions. This is a great advantage with the SYNTACER or SYNTACER₂, and helped us to implement program understanding and analysis.

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

Compilation of switch blocks

```
switch st
| Starting do
  reset
    step = true -> 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) -> false when (st=Starting)
every resS;

```

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

CMatch (x) $\langle G \rangle = \langle D_1, N_1 \rangle, \langle G' \rangle = \langle D_2, N_2 \rangle$ if
 D_1 and D_2 and
 $N_1 \sqsubseteq N_2$ and
 $x \in \text{range } x$
 $\text{range } x \rightarrow \text{range}^{(x \rightarrow x)}(G\langle x \rangle)$
 \vdots
 $\text{range } x \rightarrow \text{range}^{(\text{range } x)}(G\langle x \rangle)$
 and
 $y = \text{match } x$
 $\quad \text{with } y \in \text{range } x$
 $\quad \quad \quad \rightarrow \text{range}^{(y \rightarrow y)}(G\langle y \rangle)$
 \vdots
 $\text{range } x \rightarrow \text{range}^{(\text{range } x)}(G\langle x \rangle)$
 where $\langle y \rangle = \langle D_1, N_1 \rangle, \langle G \rangle = \langle D_2, N_2 \rangle$
 and $\langle D_1, N_1 \rangle \sqsubseteq \langle D_2, N_2 \rangle$
 $\text{and } \langle D_1, N_1 \rangle, \langle G \rangle = \langle D_2, N_2 \rangle, \langle G \rangle = \langle D_2, N_2 \rangle$

This ratio is translated into

and opt = 1 \rightarrow (open all) when Left(0:T) = 1
and all = merge = (Left \rightarrow 2 * opt) Right \rightarrow 0)

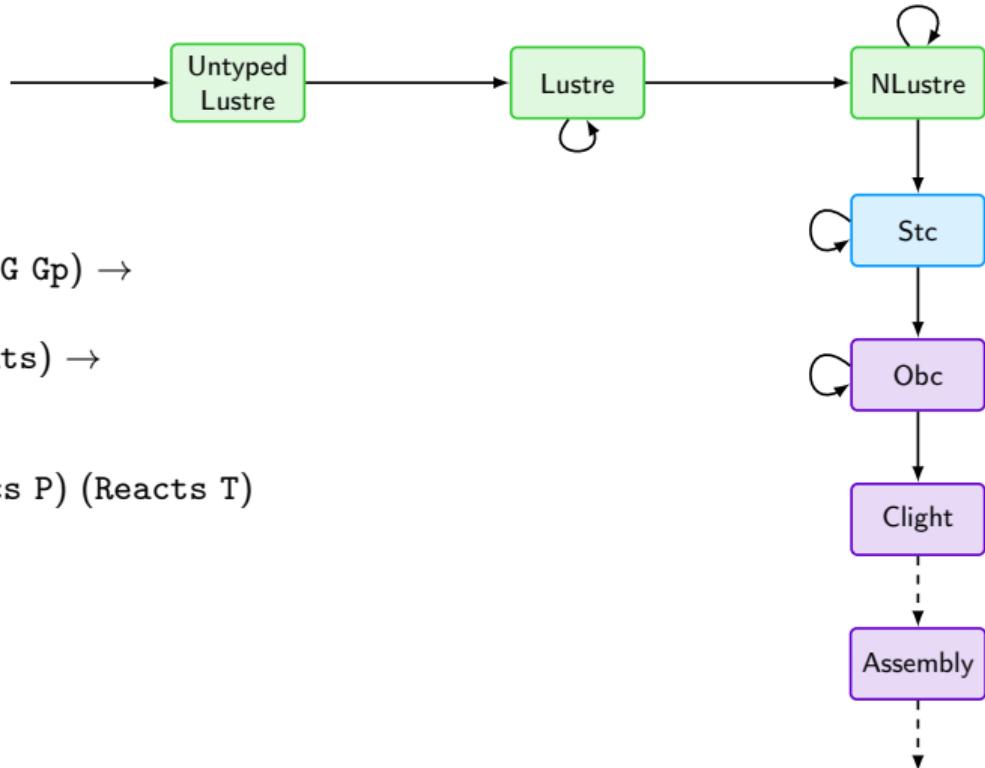
This translation highlights the fact that, last, $a2$ in the source program refers to the previous value of $a1$ whereas $a2$ in the target program refers to a local variable $a2$ declared after $a1$.

possible to enter and leave strongly shaded one section, that is, 10 times more often than expected. This is a very strong effect with the SYNT-CREATE or SYSTEMCREATE, and largely simplifies problem understanding and analysis.

- sampling explicated by `when`
 - choice explicated by `merge`
 - constants are also sampled
 - only `reset` blocks remain

Verified Compilation of Synchronous Dataflow with State Machines

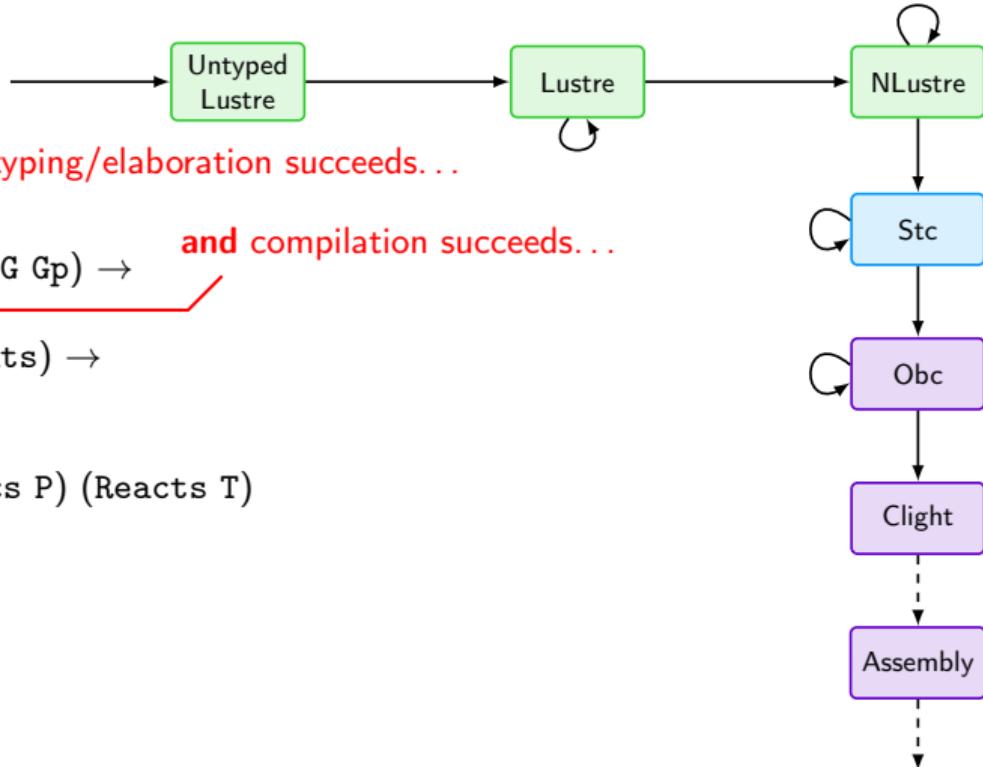
Main Correctness Theorem



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

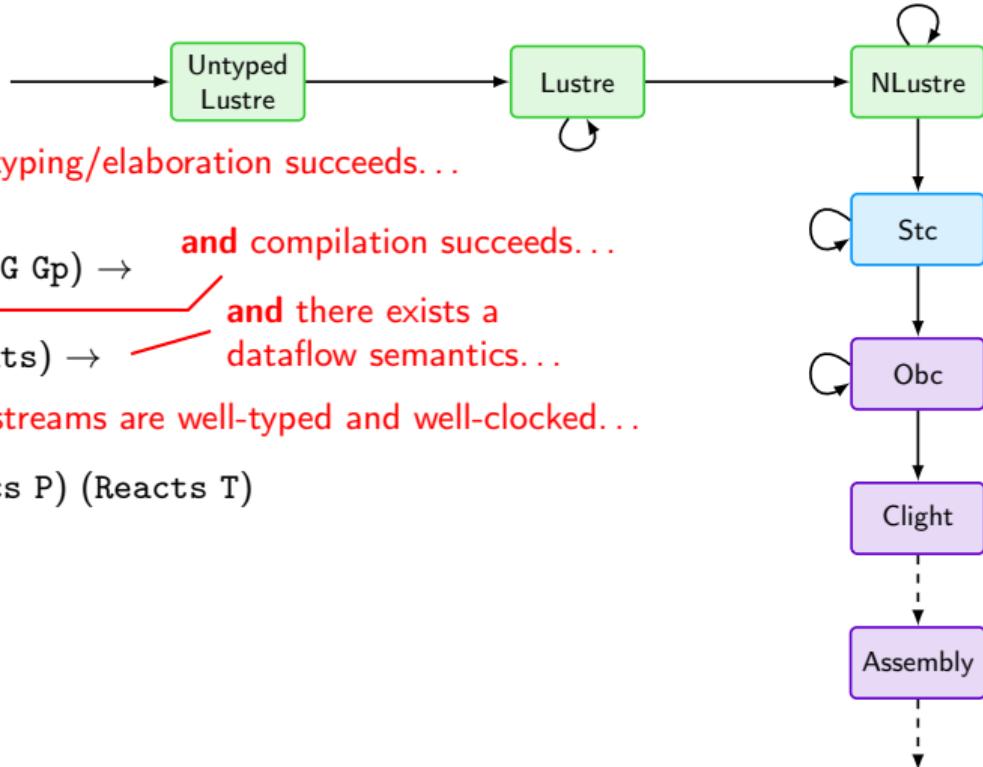


Theorem behavior_asm:

$\forall D G Gp P \text{ main ins outs},$
elab_declarations $D = \text{OK} (\text{exist } _- G Gp) \rightarrow$
compile $D \text{ main} = \text{OK} P \rightarrow$ and compilation succeeds...
sem_node $G \text{ main} (pStr \text{ ins}) (pStr \text{ outs}) \rightarrow$
wt_ins $G \text{ main ins} \rightarrow$
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...

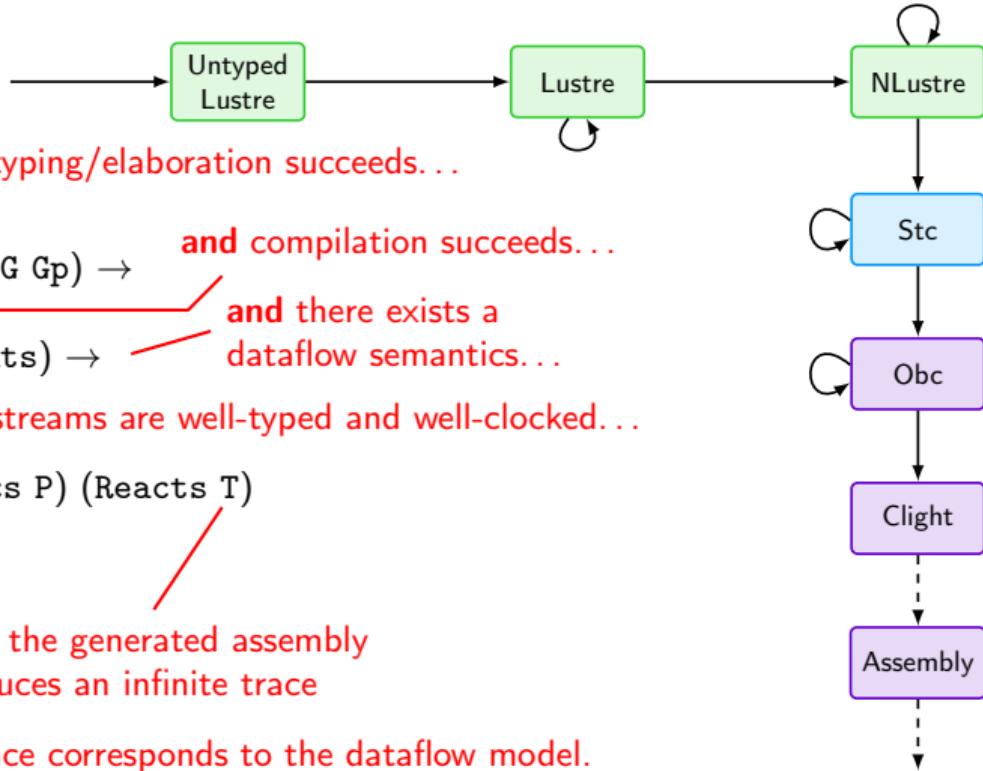
Main Correctness Theorem



Theorem behavior_asm:

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

Main Correctness Theorem



Theorem behavior_asm:

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

if typing/elaboration succeeds...
then the generated assembly
produces an infinite trace

and the trace corresponds to the dataflow model.

Dataflow relational semantics

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

Dataflow relational semantics

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

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	...

Dataflow relational semantics

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

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \quad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es} \text{ sem_equation}$$

Dataflow relational semantics

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

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \quad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es} \text{ sem_equation}$$

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

Dataflow relational semantics

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

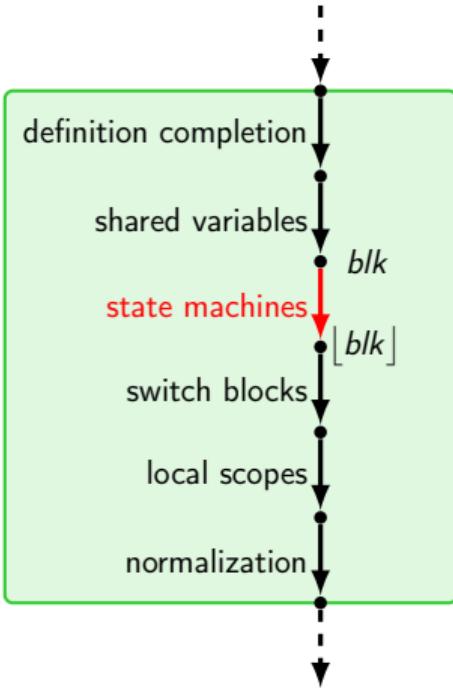
pause	F	F	F	...	F	F	...	T	...	F	...	F	...
time	0	0	50	...	750	0	...	150	...	350	...	500	...
step	T	F	F	...	T	F	...	F	...	F	...	T	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \quad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es} \text{ sem_equation}$$

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

- when for **switch** blocks
- mask for **reset** blocks
- select for state machines

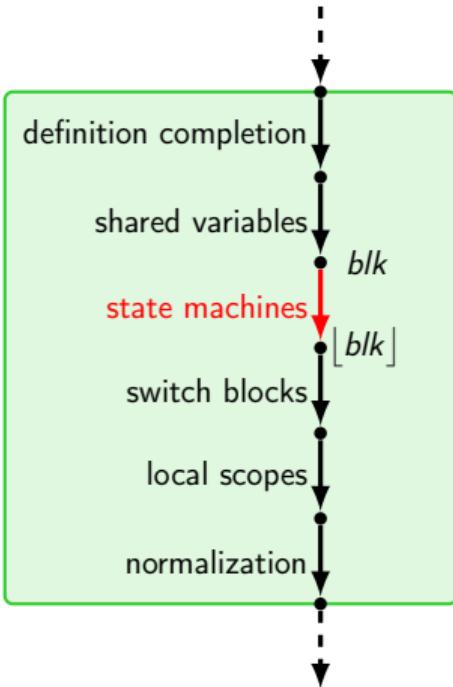
Compilation correctness – state machines



Lemma (State machines correctness)

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

Compilation correctness – state machines



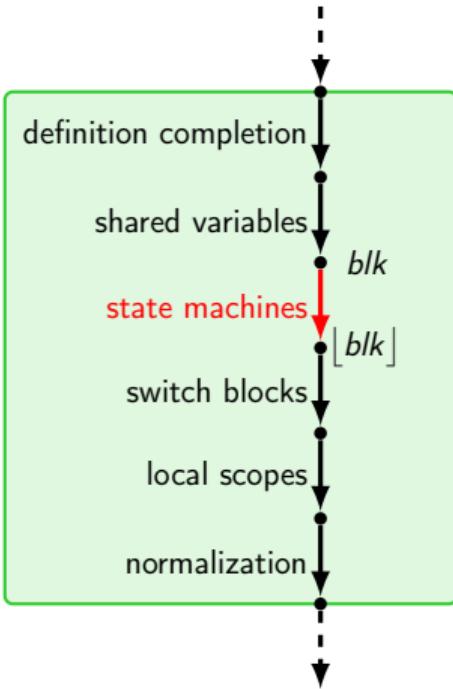
Lemma (State machines correctness)

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

Works well:

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

Compilation correctness – state machines



Lemma (State machines correctness)

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

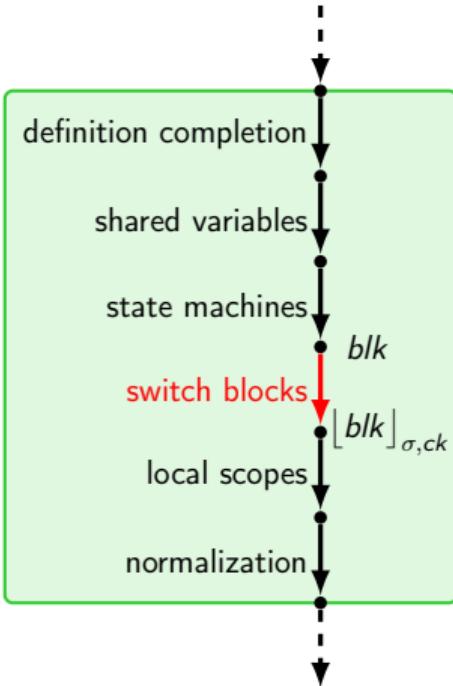
Works well:

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

Works less well:

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

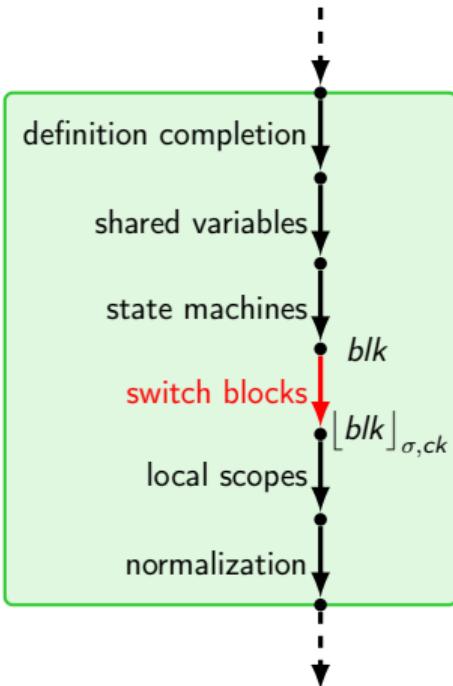
Compilation correctness – switch blocks



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 correctness – switch blocks



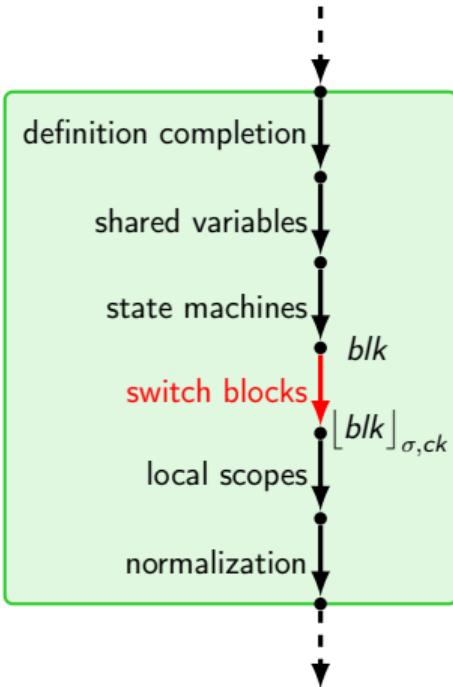
Lemma (Switch correctness)

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

Works less well:

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

Compilation correctness – switch blocks



Lemma (Switch correctness)

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

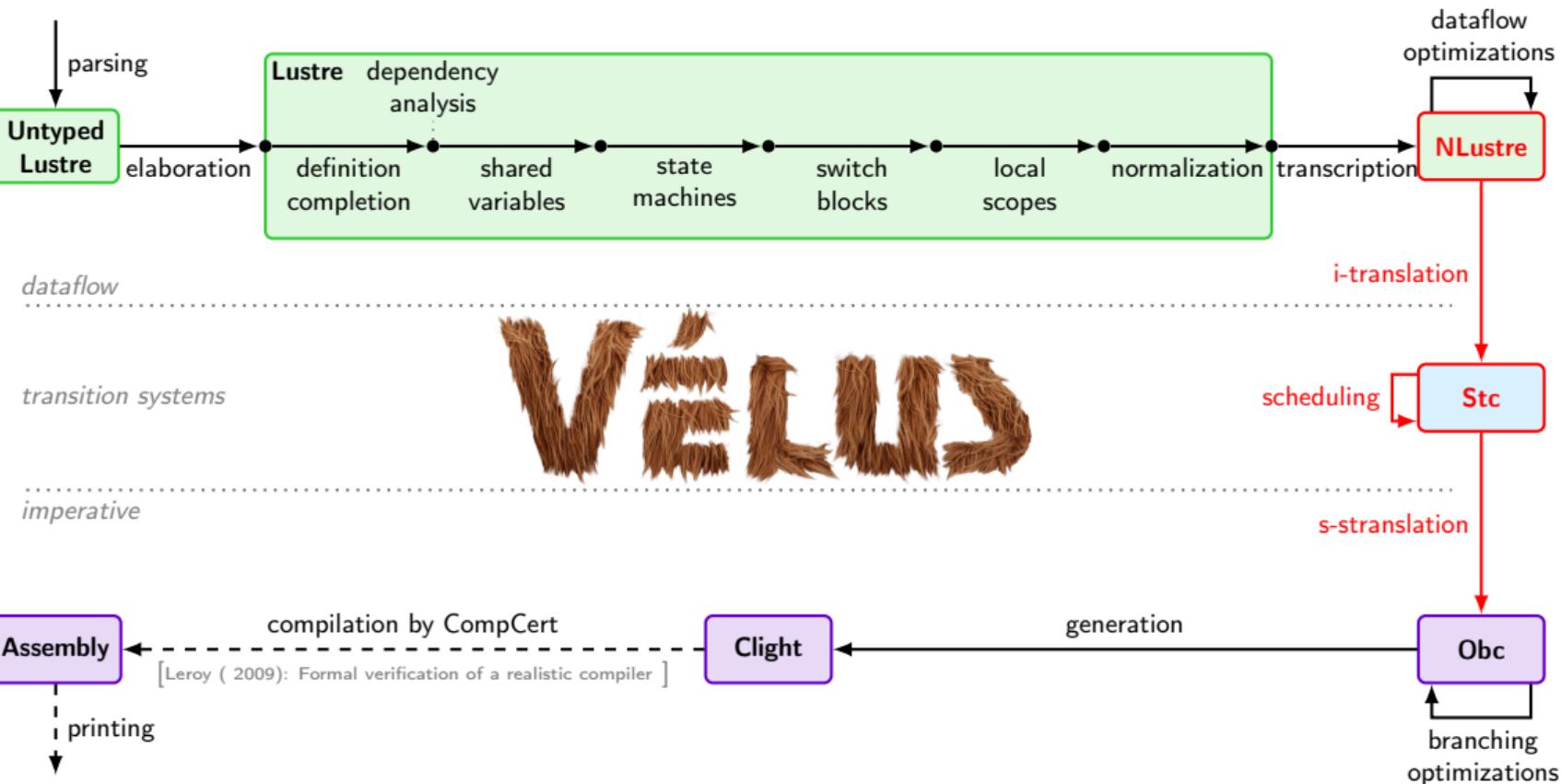
Works well:

- 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

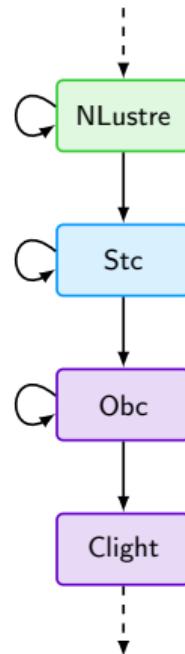
The Vélus Compiler



Generation of imperative code

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

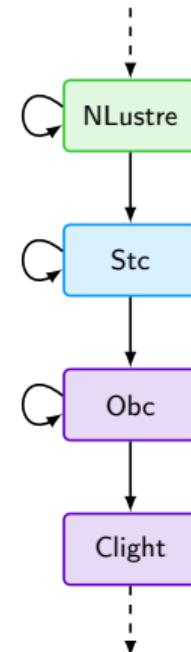
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
switch(st) { case Starting: resS = res; }  
switch(st) {  
    case Starting:  
        if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) {  
    case Starting:  
        step = st.stepS;  
        break;  
    case Moving: ...  
}  
switch(st) { case Starting: st.stepS = false; }
```

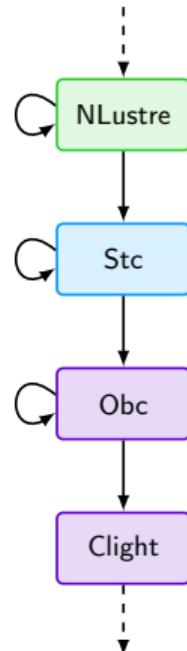
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
switch(st) { case Starting: resS = res; }  
switch(st) {  
    case Starting:  
        if(resS) st.stepS = true;  
}  
  
ena = true;  
switch(st) {  
    case Starting:  
        step = st.stepS;  
        break;  
    case Moving: ...  
}  
  
switch(st) { case Starting: st.stepS = false; }
```

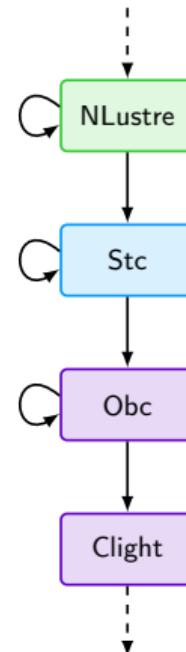
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
switch(st) { case Starting: resS = res; }  
switch(st) {  
    case Starting:  
        if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) {  
    case Starting:  
        step = st.stepS;  
        break;  
    case Moving: ...  
}  
switch(st) { case Starting: st.stepS = false; }
```

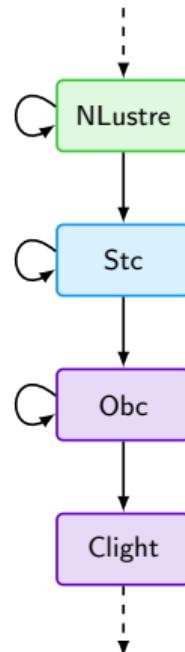
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
switch(st) { case Starting: resS = res; }  
switch(st) {  
    case Starting:  
        if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) {  
    case Starting:  
        step = st.stepS;  
        break;  
    case Moving: ...  
}  
  
switch(st) { case Starting: st.stepS = false; }
```

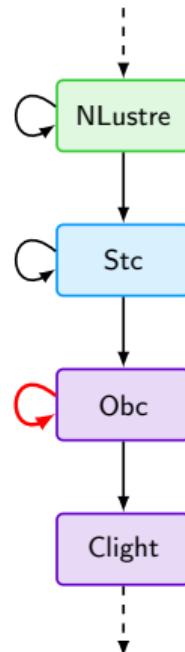
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
switch(st) {  
    case Starting:  
        resS = res;  
        if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) {  
    case Starting:  
        step = st.stepS;  
        st.stepS = false;  
        break;  
    case Moving: ...  
}
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction

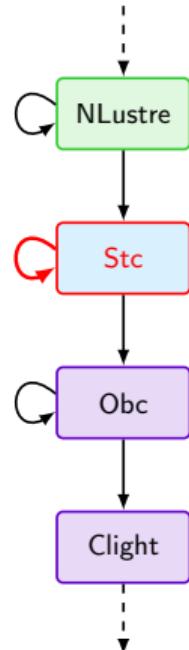


Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
switch(st) { ←  
    case Starting:  
        resS = res;  
        if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) { ←  
    case Starting:  
        step = st.stepS;  
        st.stepS = false;  
        break;  
    case Moving: ...  
}
```



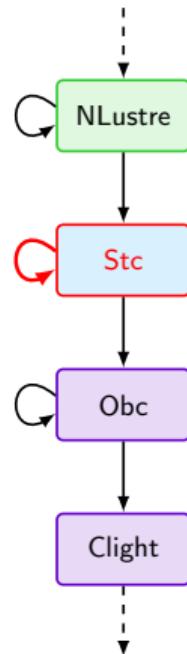
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
ena = true;  
switch(st) {  
    case Starting:  
        resS = res;  
        if(resS) st.stepS = true;  
        step = st.stepS;  
        st.stepS = false;  
        break;  
    case Moving: ...  
}
```

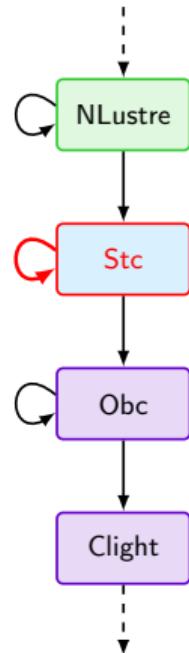
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction
- Quality of fusion depends on the scheduling



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
    stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every res;  
ena = true;  
step = merge st (Starting => stepS) (Moving => stepM);  
  
ena = true;  
switch(st) {  
    case Starting:  
        resS = res;  
        if(resS) st.stepS = true;  
        step = st.stepS;  
        st.stepS = false;  
        break;  
    case Moving: ...  
}
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction
- Quality of fusion depends on the scheduling
- Extensions of Stc: reset on state variables, multiple reset conditions



Performances

	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcc</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 (LNCS 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+gcc</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 (LNCS 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+gcc</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 (LNCS 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

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
 - `switch` blocks
 - `reset` blocks
 - state machines
- 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
- a verified implementation of an efficient compilation scheme for these blocks

<https://velus.inria.fr/emsoft2023>

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

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

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

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

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

Semantics – Hierarchical State Machines

$$\frac{H, bs \vdash ck \Downarrow bs' \quad G, H, bs' \vdash_{\perp} autinit \Downarrow sts_0}{\text{fby } sts_0 \text{ } sts_1 \equiv sts \quad \forall i, \forall k, G, (\text{select}_0^{C_i, k} \text{sts}(H, bs)), C_i \vdash_{\text{w}} autscope_i \Downarrow (\text{select}_0^{C_i, k} \text{sts } sts_1)}$$

$$G, H, bs \vdash \text{automaton initially } autinit^{ck} [\text{state } C_i \text{ autscope}_i]^i \text{ end}$$

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

$$\frac{\begin{array}{c} H, bs \vdash ck \Downarrow bs' \quad \text{fby } (\text{const } bs'(C, F)) \text{sts}_1 \equiv sts \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} \text{sts}(H, bs)), C_i \vdash_{\text{TR}} trans_i \Downarrow (\text{select}_0^{C_i, k} \text{sts } sts_1) \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} \text{sts}_1(H, bs)) \vdash blks_i \end{array}}{G, H, bs \vdash \text{automaton initially } C^{ck} [\text{state } C_i \text{ do } blks_i \text{ unless } trans_i]^i \text{ end}}$$

Semantics – Transitions

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \\ \text{bools-of } ys \equiv bs' \quad G, H, bs \vdash_{\text{I}} \text{autunits} \Downarrow sts \\ sts' \equiv \text{first-of}_{\text{F}}^C bs' sts \end{array}}{G, H, bs \vdash_{\text{I}} C \text{ if } e; \text{autunits} \Downarrow sts'}$$

$$\frac{sts \equiv \text{const } bs(C, F)}{G, H, bs \vdash_{\text{I}} \text{otherwise } C \Downarrow sts}$$

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

$$\frac{sts \equiv \text{const } bs(C_i, F)}{G, H, bs, C_i \vdash_{\text{TR}} \epsilon \Downarrow sts}$$

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs' \\ G, H, bs, C_i \vdash_{\text{TR}} \text{trans} \Downarrow sts \\ sts' \equiv \text{first-of}_{\text{F}}^C bs' sts \end{array}}{G, H, bs, C_i \vdash_{\text{TR}} \text{if } e \text{ continue } C \text{ trans} \Downarrow sts'}$$

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs' \\ G, H, bs, C_i \vdash_{\text{TR}} \text{trans} \Downarrow sts \\ sts' \equiv \text{first-of}_{\text{T}}^C bs' sts \end{array}}{G, H, bs, C_i \vdash_{\text{TR}} \text{if } e \text{ then } C \text{ trans} \Downarrow sts'}$$

Semantics – local blocks and last variables

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

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

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

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