

Verified Compilation of Synchronous Dataflow with State Machines

Timothy Bourke Basile Pesin Marc Pouzet

Inria Paris

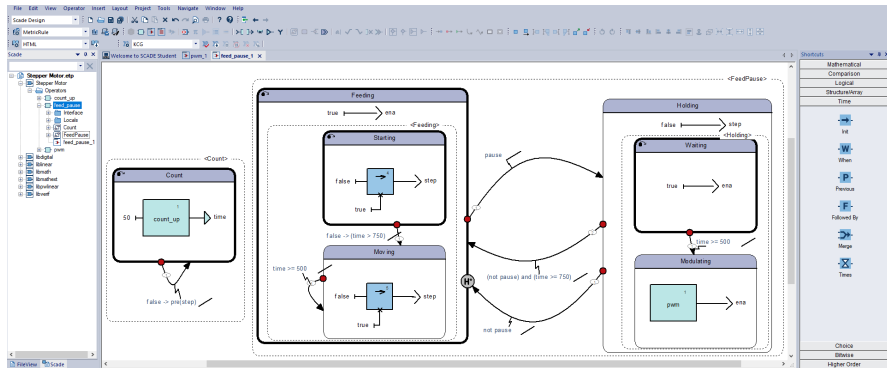
École normale supérieure, CNRS, PSL University

ESWEEK 2023 - EMSOFT

Monday, September 18

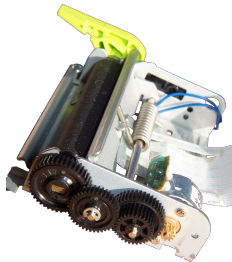
11:22am - 11:47am CET

Block-Diagram Languages for Embedded Systems

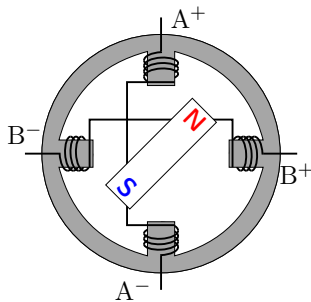
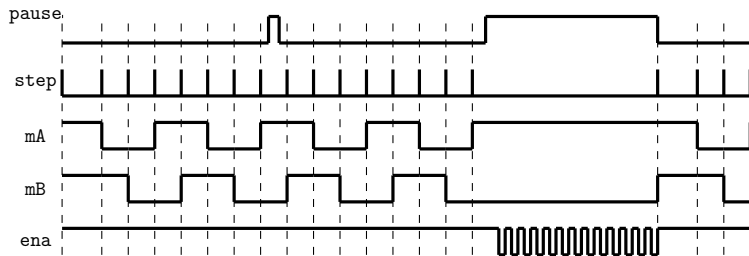
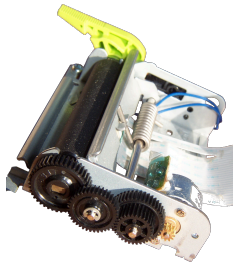


- Widely used in safety-critical applications:
Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear...
- Scade 6: Qualified compiler for Lustre + Control Structures
- Our work: Verified compilation in an Interactive Theorem Prover (Coq)

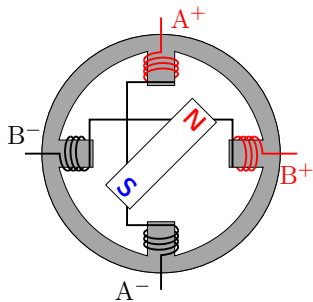
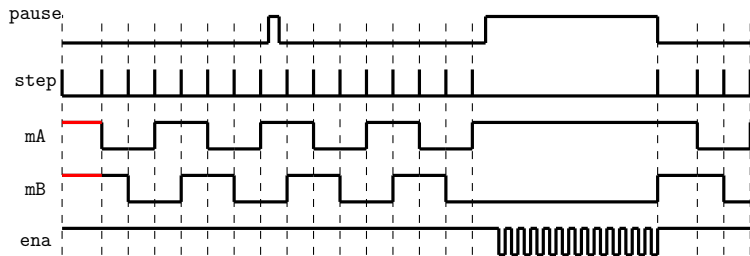
An system example: stepper motor for a small printer



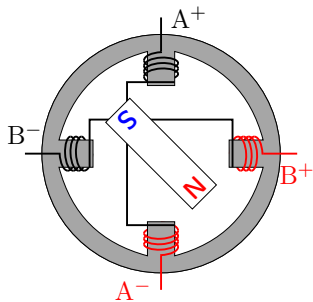
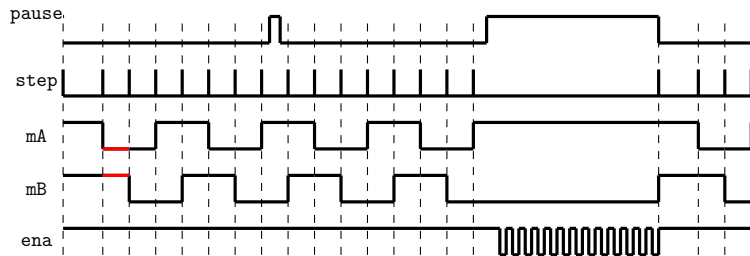
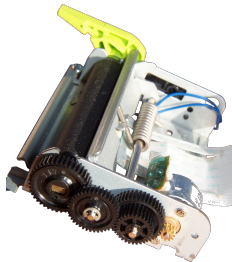
An system example: stepper motor for a small printer



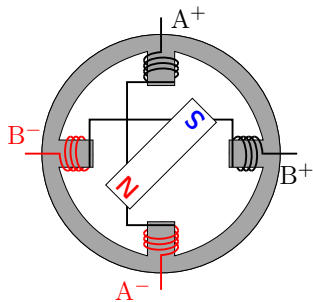
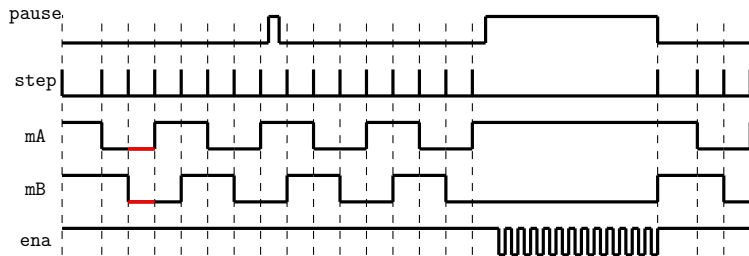
An system example: stepper motor for a small printer



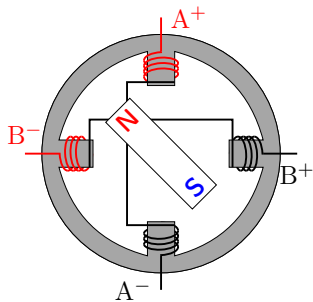
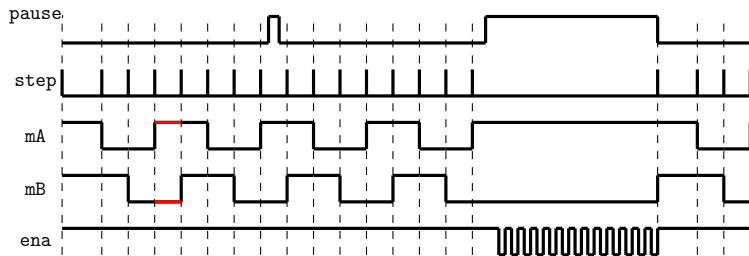
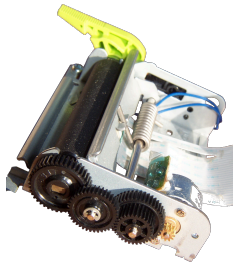
An system example: stepper motor for a small printer



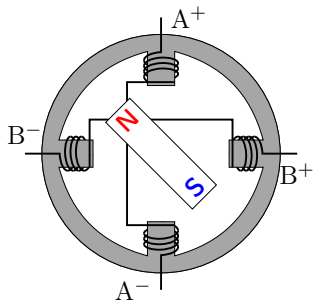
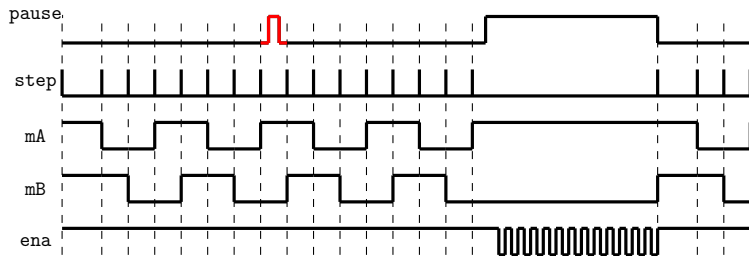
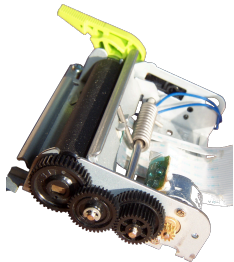
An system example: stepper motor for a small printer



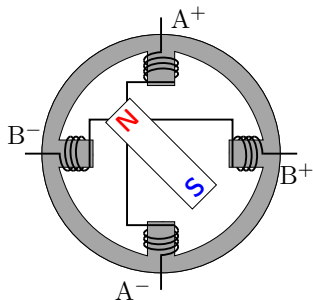
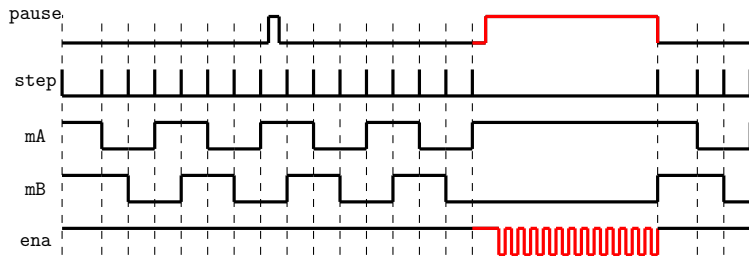
An system example: stepper motor for a small printer



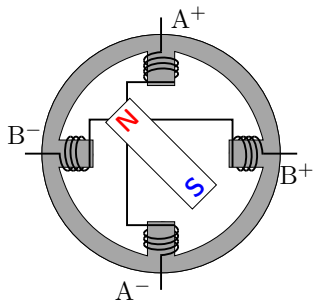
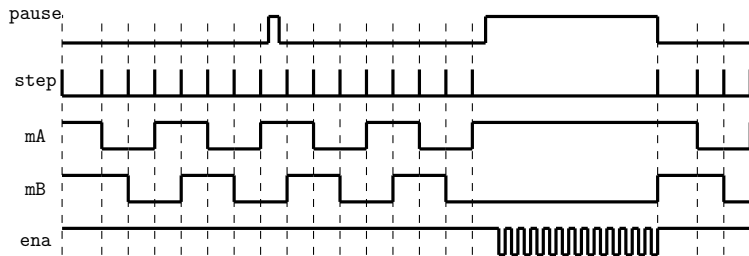
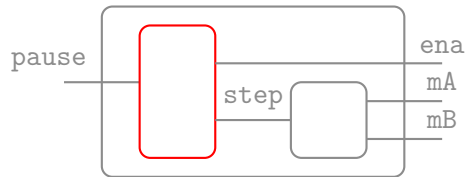
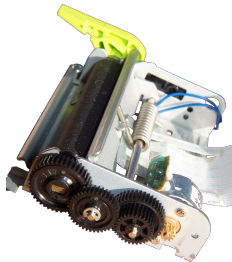
An system example: stepper motor for a small printer



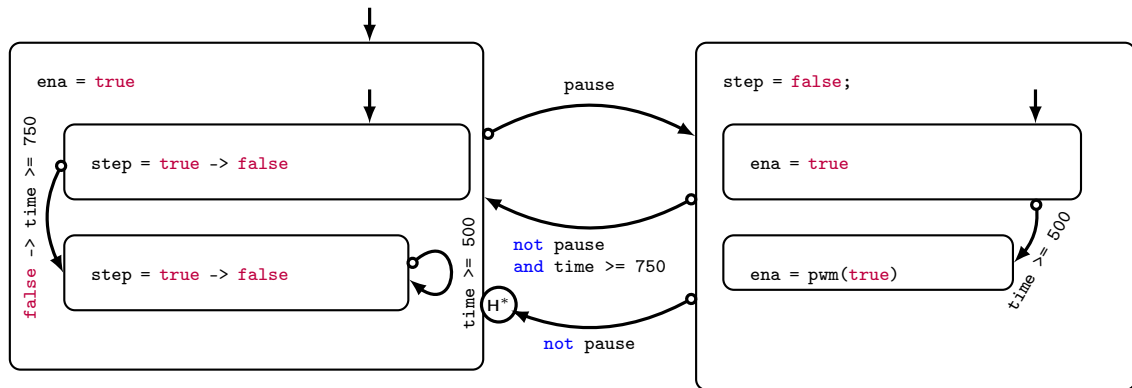
An system example: stepper motor for a small printer



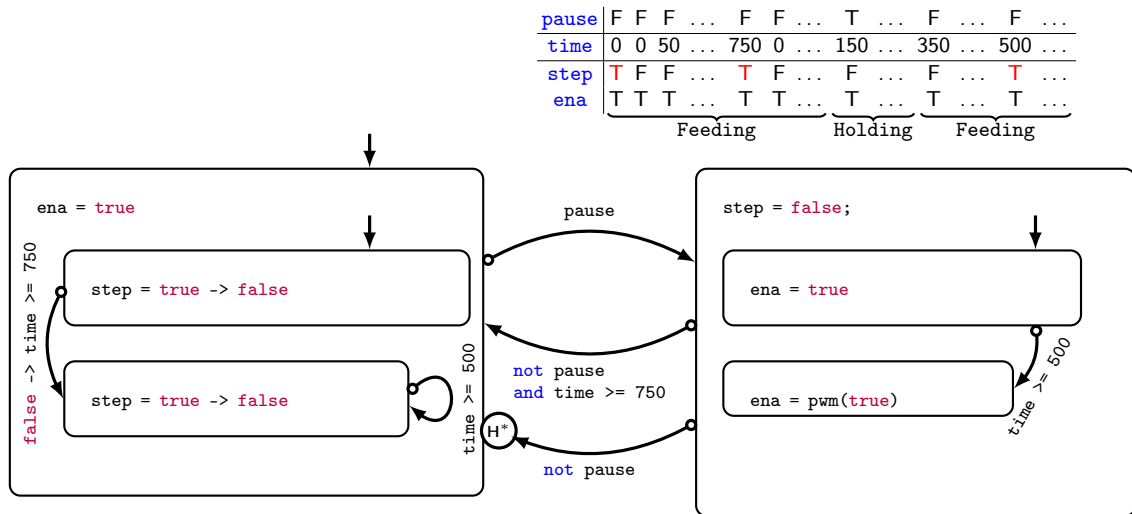
An system example: stepper motor for a small printer



Hierarchical State Machines – Example



Hierarchical State Machines – Example



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
```

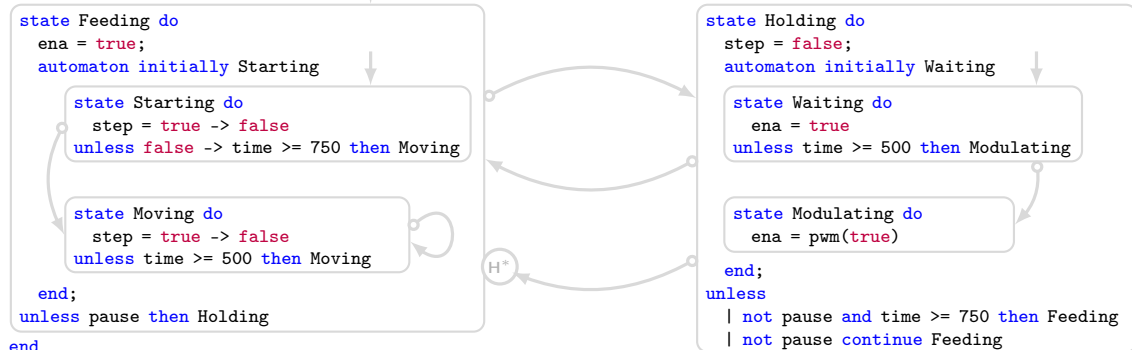
```
end;
```

```
unless pause then Holding
```

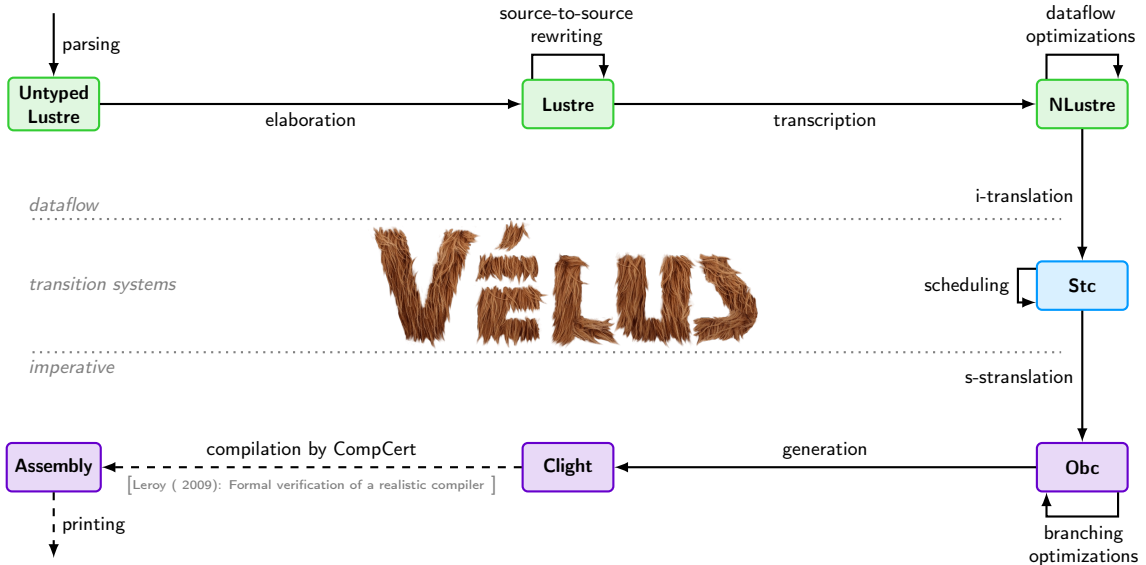
```
end
```

```
tel
```

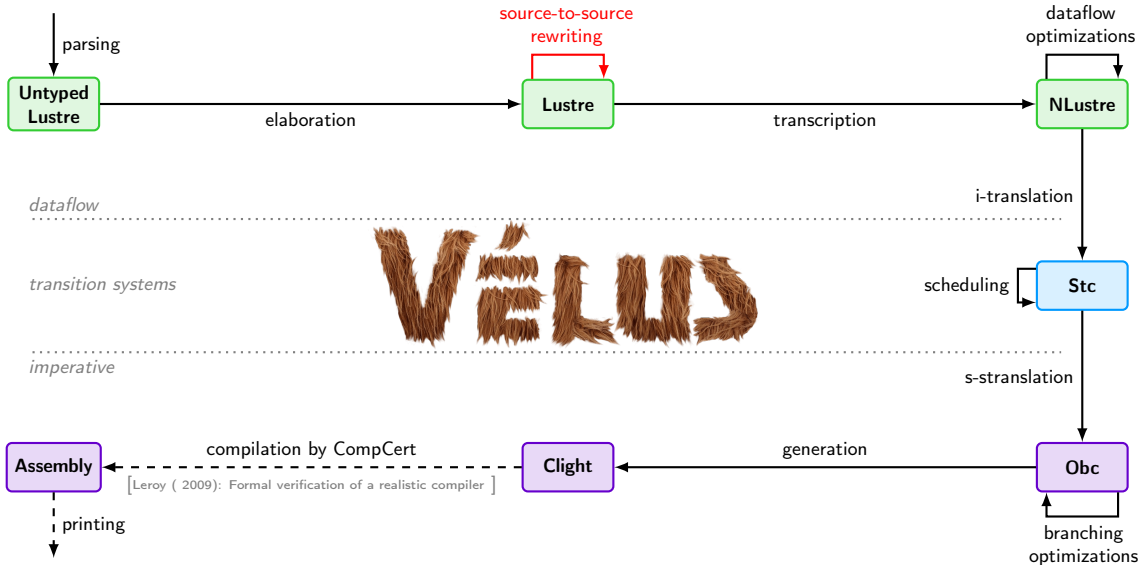
| pause | F | F | F | ... | F | F | ... | T | ... | F | ... | F | ... |
|-------|---------|---|----|-----|-----|---|---------|-----|-----|---------|-----|-----|-----|
| time | 0 | 0 | 50 | ... | 750 | 0 | ... | 150 | ... | 350 | ... | 500 | ... |
| step | T | F | F | ... | T | F | ... | F | ... | F | ... | T | ... |
| ena | T | T | T | ... | T | T | ... | T | ... | T | ... | T | ... |
| | Feeding | | | | | | Holding | | | Feeding | | | |



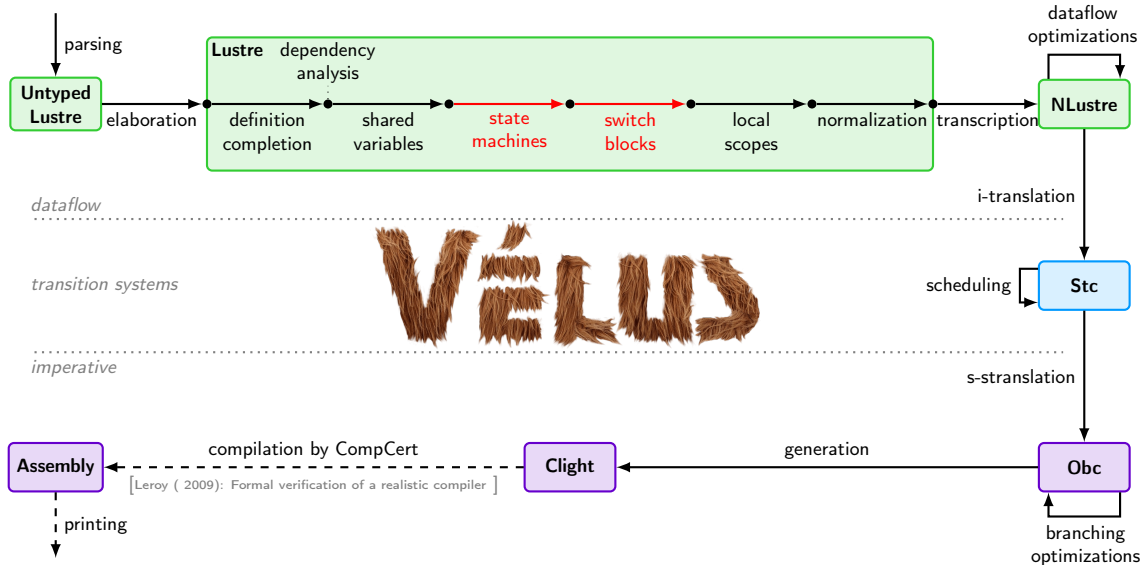
The Vélus Compiler



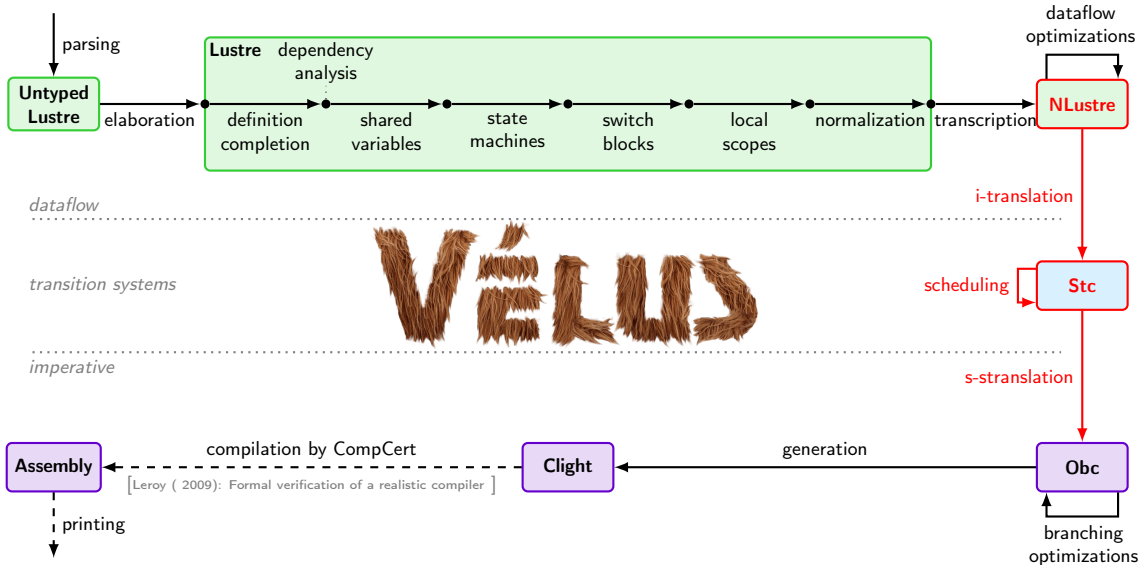
The Vélus Compiler



The Vélus Compiler



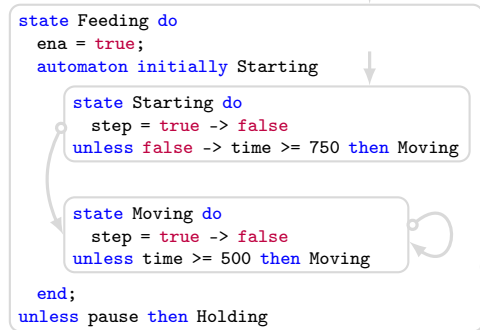
The Vélus Compiler



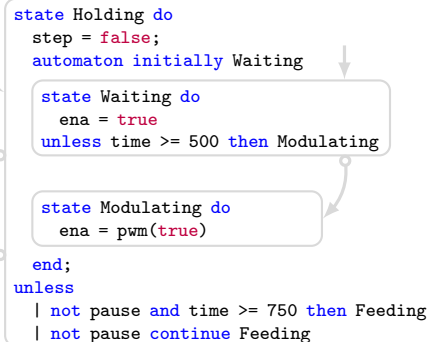
Compilation of state machines

```
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



end
tel



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

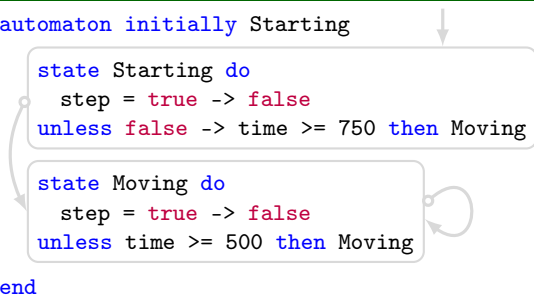
| not pause and time >= 750 then Feeding

| not pause continue Feeding

H*

Compilation of state machines

automaton initially Starting



The diagram illustrates the compilation of a state machine. It features two state blocks: 'Starting' and 'Moving'. The 'Starting' block contains the code 'step = true -> false' and 'unless false -> time >= 750 then Moving'. A curved arrow points from the 'Moving' label in this block to the 'Moving' block below. The 'Moving' block contains 'step = true -> false' and 'unless time >= 500 then Moving'. A curved arrow points from the 'Moving' label in this block back to itself, representing a self-loop. An initial arrow points down to the 'Starting' block. The entire code is enclosed in 'state' and 'end' keywords.

```
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]

$$\begin{array}{l}
\text{C3path}(x) \{ C_1 \rightarrow \{ D_1, N_1 \} \rightarrow \{ C_2 \rightarrow \{ D_2, N_2 \} \} = \\
\quad D_1, \text{end} \} \rightarrow \text{end} \{ D_2 \text{ and} \\
\quad \text{C3path}(x) = \text{end} \\
\quad y_1 = \text{aPath}(x) \\
\quad \quad \{ C_1 \rightarrow \text{path}_{C_1}^{C_1(x)}(\{ D_1 \}) \} \\
\quad \quad \{ C_2 \rightarrow \text{path}_{C_2}^{C_2(x)}(\{ C_1 \}) \} \\
\text{and -- end} \\
\text{ys} = \text{aPath}(x) \\
\quad \quad \{ C_1 \rightarrow \text{path}_{C_1}^{C_1(x)}(\{ D_1 \}) \} \\
\quad \quad \{ C_2 \rightarrow \text{path}_{C_2}^{C_2(x)}(\{ D_2 \}) \} \\
\text{where } \text{ys} = \{ y \mid y \in \text{path} \} \cup \text{path}(\{ D_1 \}) \\
\text{and } \{ y_1 \rightarrow y_2 \} = \{ y_1 \} \cup \{ y_2 \} \\
\text{and } \text{path}(D_1, C_1) = \text{path}_{C_1}^{C_1(x)}(\{ D_1, C_1(x) \})
\end{array}$$

Figure 5: The translation of switch.

[illegible]

(Initialization) $S_1 \rightarrow \{D_1, s_1, \text{env}_1\}$ (D_1, s_1, env_1) \rightarrow \rightarrow
 match env_1 with
 $\rightarrow \text{env}_1 \rightarrow \text{env}_1$ $s = s_1$ and $r = s_1$ and D_1 every env_1
 \rightarrow
 $\rightarrow \text{env}_1 \rightarrow \text{env}_1$ $s = s_1$ and $r = s_1$ and D_1 every env_1
 and
 match s with
 $\rightarrow \text{env}_1 \rightarrow \text{env}_1$ $\text{env} = \text{env}_1$ and $\text{env} = \text{env}_1$ and D_1 every env_1
 \rightarrow
 $\rightarrow \text{env}_1 \rightarrow \text{env}_1$ $\text{env} = \text{env}_1$ and $\text{env} = \text{env}_1$ and D_1 every env_1
 and env_1 every env_1 env_1 env_1
 and env_1 every env_1 env_1 env_1
 and env_1 every env_1 env_1 env_1
 where $\text{env}_1, s, r, \text{env}_1 \in \text{FVar}(\text{env}_1) \cup \text{FVar}(\text{env}_1)$
 where $\text{env}_1, s, r, \text{env}_1 \in \text{FVar}(\text{env}_1) \cup \text{FVar}(\text{env}_1)$

Figure 6: The translation of *argument*²

3.2.2 The Type System

We should first attend the typing rule for the new primitive grammatical constructs. The typing rule should assign the type to each production sentence such that it gives the same type as the productions of the grammar. These rules are only allowed in the typing of the productions. These rules are only allowed in the typing of the productions. These rules are only allowed in the typing of the productions.

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]

```

C3path( $x$ )  $\{C_1 \rightarrow (D_1, N_1) \wedge \dots \wedge (C_n \rightarrow (D_n, N_n)) =$ 
 $D_1 \wedge \dots \wedge D_n$  and
 $\langle x, a \rangle \in c$  and
 $\exists_1 = a \wedge \exists_2 =$ 
 $\{C_1 \rightarrow \text{pre}^{C_1}_{\exists_1}(G_1)\}$ 
 $\dots$ 
 $\{C_n \rightarrow \text{pre}^{C_n}_{\exists_n}(G_n)\}$ 
and -- and
 $\exists_2 = a \wedge \exists_3 =$ 
 $\{C_1 \rightarrow \text{pre}^{C_1}_{\exists_1}(G_1)\}$ 
 $\dots$ 
 $\{C_n \rightarrow \text{pre}^{C_n}_{\exists_n}(G_n)\}$ 
where  $\exists_1$  is if  $\exists_1(x) \wedge \exists_2(x)$ 
and  $\{x_1 \dots x_n\} = \text{Split}(\{C_1 \cup D_1, C_2 \cup D_2\})$ 

```

Figure 5: The translation of switch

`while This code is translated into:
 while a1 < a2
 and opt < 1 -> (pre a1) when Left(a1) = 1
 and opt < 2 -> (pre a2) when Right(a2) = 2
 and a1 = merge < (Left -> 2 + opt) (Right -> 0)
 and a2 = merge <
 (Left -> (pre a2) when Right(a2) = 1)
 (Right -> 0) -> pre a2 when Right(a2)`

This translation highlights the fact that last
 time we were only in the previous value of `a1`

Calculation: $\beta_1 \rightarrow (D_1, \text{true}, \text{true})$ ($D_1, \text{true}, \text{true}$) $\rightarrow \dots$
 $\beta_2 \rightarrow (D_2, \text{true}, \text{true})$ ($D_2, \text{true}, \text{true}$)

match join with

$\beta_1 \rightarrow \text{ghost}, \text{tr} = \text{tr}'_1$ and $\text{tr} = \text{tr}'_1$ and D_1 every join

$\beta_2 \rightarrow \text{ghost}, \text{tr} = \text{tr}'_2$ and $\text{tr} = \text{tr}'_2$ and D_2 every join

and

match join with

$\beta_1 \rightarrow \text{ghost}, \text{tr} = \text{tr}'_1$ and $\text{tr} = \text{tr}'_1$ and D_1 every join

$\beta_2 \rightarrow \text{ghost}, \text{tr} = \text{tr}'_2$ and $\text{tr} = \text{tr}'_2$ and D_2 every join

and stable join $\rightarrow D_1 \vee D_2$

and check join $\rightarrow \text{False}$ (No join)

where $\text{tr}'_1, \text{tr}'_2, \text{tr}'_3$ of $\text{FF}(\text{tr})$ ($\text{FF}(\text{tr})$)

$\text{FF}(\text{tr}) = \text{FF}(\text{tr}) \cup \text{FF}(\text{tr})$

Figure 6: The translation of arguments

Figure 6: The translation of automatics

```
(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

```

C3pathk(x) {C3 → (D3, N3) → (C2 → (D2, N2)) =
  D3 mod ← a3 and D3 mod
  c1a3 ← c and
  b1 ← a3 * x
  {C3 → projC3(a3)(D3)}
  {C3 → projC3(a3)(D3)}
  and -- mod
  y1 ← a3 * x
  {C3 → projC3(a3)(D3)}
  {C3 → projC3(a3)(D3)}
  where "a" is if b ∈ p(D)
  and {y1 ← y2} = Split1(C3, D3, C3, a)

```

Figure 5: The translation of switch

`while This code is translated into:`

```

    while a < b
    and opt < 1 → ((pre a) when Left(a)) = 1
    and opt < 1 → merge < (Left → 2 + opt) (Right → 0)
    and a2 = merge <
      (Left → 1pre a2) when Right(a)
      (Right → 0) → pre a2 when Right(a)
  
```

This translation highlights the fact that last
 value is only used in the previous value of `a`.

[illegible]

2000-01 The translation of *neuronal*

possible to solve and leave strongly disjunctive questions, if
it is more than two iterations. This is a dry dis-
pute with the SYNTAX or SYNTAXER, and let
the program understand and analyze.

3.2.2 The Type System

We should first extend the typing rule for the new
grammar constructs. The typing rule derived from
translation semantics such that it gives the same type
translation semantics as the original rule. In particular,
the typing of the translation rules are only allowed
that newly introduced constructs are only allowed
if they are contained in some full construction.
The following rule is automatically generated:

```
(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]

$\text{C}(\text{path}) \{ \langle \rangle \} \{ C_1 \rightarrow \{ \{ D_1, N_1 \} \}, \{ C_2 \rightarrow \{ \{ D_2, N_2 \} \} \}$
 $\{ D_1 \text{ and } \dots \text{ and } D_2 \text{ and } \dots \}$
 $\{ C_1 \text{ and } \dots \text{ and } C_2 \}$
 $N_1 = \text{newge} \ x$
 $\{ C_1 \rightarrow \text{path}^{(1)}(x) \{ G_1 \} \}$
 $\{ C_2 \rightarrow \text{path}^{(2)}(x) \{ G_2 \} \}$
 $\text{and } \dots \text{ and } \dots$
 $\text{if } x = \text{newge} \ x$
 $\{ C_1 \rightarrow \text{path}^{(1)}(x) \{ G_1 \} \}$
 $\{ C_2 \rightarrow \text{path}^{(2)}(x) \{ G_2 \} \}$
 $\{ G_1 \text{ and } \dots \text{ and } G_2 \}$
 $\text{and } \{ D_1 \rightarrow \text{pr} \} \cup \{ D_2 \rightarrow \text{pr} \}$
 $\text{and } \{ D_1 \rightarrow \text{pr} \} = N_1 \cup \dots \cup N_n$
 $\text{and } \{ D_1 \in D_1, G_1 = \text{split}_{D_1}(\text{C}(\text{path}), D_1, G_1) \}$

`while x = x
 and xpt = 1 -> ((yes a2) when let(a2)) = 1
 and a1 = merge = (let(a1 -> 2 + a2) (right -> 2))
 and a2 = merge =
 (let(a1 -> (yes a2) when right(a2))
 (right -> (1 -> yes (a2 when right(a2)) + 1))`

This translation highlights the fact that `let a1` in the previous code refers to the previous value of `a1` whenever `merge` calls it.

(Observation) $\bar{X}_1 \rightarrow (D_1, \text{val}_1, \text{err}_1)$, $(D_1, \text{val}_1, \text{err}_1) \rightarrow \dots \rightarrow (D_n, \text{val}_n, \text{err}_n)$, $(D_n, \text{val}_n, \text{err}_n) \rightarrow \bar{X}_2$
 match pos with \bar{X}_1
 $\bar{X}_1 \rightarrow \text{error}$, $\text{err} = \text{err}_1$ and $\text{val} = \text{val}_1$ and D_1 always pos
 $\bar{X}_1 \rightarrow \text{error}$, $\text{err} = \text{err}_1$ and $\text{val} = \text{val}_1$ and D_1 always pos
 match neg with \bar{X}_1
 $\bar{X}_1 \rightarrow \text{error}$, $\text{err} = \text{err}_1$ and $\text{val} = \text{val}_1$ and D_1 always neg
 $\bar{X}_1 \rightarrow \text{error}$, $\text{err} = \text{err}_1$ and $\text{val} = \text{val}_1$ and D_1 always neg
 match pos with \bar{X}_2
 $\bar{X}_2 \rightarrow \text{error}$, $\text{err} = \text{err}_n$ and $\text{val} = \text{val}_n$ and D_n always pos
 $\bar{X}_2 \rightarrow \text{error}$, $\text{err} = \text{err}_n$ and $\text{val} = \text{val}_n$ and D_n always pos
 match neg with \bar{X}_2
 $\bar{X}_2 \rightarrow \text{error}$, $\text{err} = \text{err}_n$ and $\text{val} = \text{val}_n$ and D_n always neg
 $\bar{X}_2 \rightarrow \text{error}$, $\text{err} = \text{err}_n$ and $\text{val} = \text{val}_n$ and D_n always neg
 match pos with \bar{X}_1 and \bar{X}_2
 match neg with \bar{X}_1 and \bar{X}_2
 match pos with \bar{X}_1 and \bar{X}_2
 match neg with \bar{X}_1 and \bar{X}_2
 where $\bar{X}_1, \text{err}_1, \text{val}_1, \text{err}_n, \text{val}_n, D_1, D_n \in \mathcal{F}(\Sigma, \Pi)$

Figure 4: The translation of our match

possible to *give* and *have* strongly distinctive emotion, that is, to *give* more than *one* argument. This is a key difference with the SYNTAXIC or SYNTACTIC, and largely explains program understanding and analysis.

```
(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`

also, this code is translated into:

```
switch * *
  case 1 -> { case 1: when left(1) = 1
    and left = merge * { state = 2 * } }
  case 2 -> merge *
    { state = 1 * }
  case 3 -> { case 3: when left(1) = 1
    and left = merge * { state = 2 * } }
  case 4 -> { case 4: when left(1) = 1
    and left = merge * { state = 2 * } }
```

The code highlights the fact that `case 1` is the first case to be executed, and that `case 2` is the second case to be executed.

Figure 6: The translation of `switch`

possible to define and have strongly dualized one function, that is, it runs more than one time. That is a key idea in the design of the `SwitchCase` or `SwitchCase`, and largely simplifies program understanding and analysis.

3.1.2 The Type System

We should first extend the typing rule for the new primitive operation. The typing rule should assume the primitive operation only that it gives the same type as the typing of the translation. (There is no need to check the typing of the translation, as only when the type is the same, the operation is correct.) For each operation, we should automatically compute 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

[illegible]

Figure 5: The translation of switch

`while x = 0
 and opt = 1 -> (pre opt) when left(a) = 1
 and a1 = merge a (left a) 2 = opt (right a) 2
 and a2 = merge a
 (left a) -> (pre a2) when right(a) = 1
 (right a) -> (1 -> pre (a2 when right(a) = 1))`

This translation highlights the fact that last

[illegible]

Figure 6: The translation of automata

3.2.2 The Type System

We should first extend the typing rule for the new grammar constructs. The typing rule should assign grammatical semantics such that it gives the same type translation semantics as the translation. This rule sets in place the typing of the translation. These rules set in place the typing of the translation. These rules set in place the typing of the translation.

```
(pst, pres) = (Starting, false) fby (st, res);
```

```
switch pst
```

```
| Starting do
```

reset

$$(\text{st}, \text{res}) =$$

```
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

```

C3pathk (x) (C1 → (D1, N1)) → (C2 → (D2, N2)) =
  D1, and ← and D2, and
  C1 → C2 = ← and
  N1 = ← page →
    { C1 → preqD1C1(x)(D1) }
    { C2 → preqD2C2(x)(D2) }
  and ← and
  N2 = ← page →
    { C1 → preqD1C1(x)(D1) }
    { C2 → preqD2C2(x)(D2) }
  where N1 is if D1(x) ∈ D1
  and { N1 → N2 } = D1(x) ∈ D2
  and { D1, D2 } = D1(x) ∈ D1, D2(x) ∈ D2

```

Figure 5: The translation of switch

`while (This code is translated into:
 while (a[low] < a[high])
 and opt = 1 -> ((pre a[low] when Left(a[low]) = 1
 and a[low] = merge a[low] -> 2 + opt) Right(a[low] -> 2)
 and a[low] = merge a[low]
 (Left a[low] -> 2) when Right(a[low])
 Right(a[low] -> 2) -> pre a[low] when Right(a[low])`

This translation highlights the fact that last-
 element access is not needed in the previous version of `at`.

(Substitution) $\frac{X_1 \rightarrow (D_1, \text{env}_1, \text{env}_2) \quad (D_2, \text{env}_1, \text{env}_2) \rightarrow \dots}{X_2 \rightarrow (D_2, \text{env}_1, \text{env}_2) \quad (D_2, \text{env}_1, \text{env}_2)}$
 match env with
 $X_1 \rightarrow \text{env}$ if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_1 always env
 $X_2 \rightarrow \text{env}$ if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_2 always env
 and
 match env with
 $X_1 \rightarrow \text{env}$ if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_1 every env
 $X_2 \rightarrow \text{env}$ if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_2 every env
 and
 match env with X_3 if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_3 every env
 and
 match env with X_4 if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_4 every env
 and
 match env with X_5 if $\text{env}_1 = \text{env}'_1$ and $\text{env}_2 = \text{env}'_2$ and D_5 every env
 where $\text{env}_1, \text{env}_2, \text{env}'_1, \text{env}'_2 \in \text{Env}$ and $D_1, D_2, D_3, D_4, D_5 \in \text{Env}$

Figure 6: The translation of *matchenv*

2000-01 The translation of *neuronal*

possible to refine and leave strongly diagnostic structure, if it is more dense than two transitions. This is a key difference with the `SYNCHART` or `SYNTECHART`, and lets the simplifier program understand and analyze.

3.2.2 The Type System

We should first extend the typing rule for the new grammatical constructs. The typing rule should assign a translation semantic such that it gives the same type translation semantic for the translation. These rules state in part that the typing of the translation is only allowed that newly introduced constructs are only assigned that semantic (they are embedded in syntactic constructs), and the translation is assigned automatically.

```
(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 resS;
```

- sampling explicited by `when`

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

Figure 5: The topological of spectra

```

defn. This code is translated into:
  stack ← 0
  and opt ← 1 → (if (pos a1) when (left(a1)) + 1
  and a1 ← merge ← (left ← 2 + opt) (right → 0)
  and a2 ← merge ←
    (left ← (pos a2) when (right(a2))
    (right → 0) → pos (a2 when (right(a2)) + 1))

```

This translation highlights the fact that `last` of `ls` in the expression refers to the previous value of `ai` whereas `pos` refers to the previous value of `opt`.

[illegible]

Figure 6: The translation of argument²

possible to *allow* and *have* strongly disjunctive structure, that is, to *have* more than two tagsets. This is a key difference with the *SYN-CATEGORY* or *SYN-CATEGORY* and *category* simplification programs (understanding and analysis).

1.2.2 The Type System

We should first attend the typing rule for the new pre-gramming constructs. The typing rule states the same types as translation semantics such that it is the same type as in particular the typing of the translation. These rules are only added in a that newly introduced in *category* (understanding and analysis). But note that they are not added in *category* (understanding and analysis).

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]

Figure 5: The translation of switch.

also, this code is translated into:

```

when s = 1
  resS = 1 -> (resS && !st) else (resS && st) = 1
  resM = merge s (resS && !st) else (resM && st) = 1
  step = merge s (stepS && !st) else (step && st) = 1
  stepS = 1 -> (stepS && !st) else (stepS && st) = 1

```

This code highlights the fact that 'step' is set in the 'when' clause to the previous value of 'step'.

Figure 6: The translation of assignment.

possible to define and have strongly dualized one another. That is, a few days in, it seems more than one assignment. That is, a few days in, it seems more than one assignment. That is, a few days in, it seems more than one assignment.

- sampling explicited by **when**
- choice explicited 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;
```

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

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

Figure 5: The resolution of a graph.

also. This code is translated into

```

    a[low] = a[high]
    mid = (low + high) / 2
    if (a[mid] < x)
        low = mid + 1
    else
        high = mid
    return low
}

```

This translation highlights the fact that least $\alpha 1$ in the common ancestor refers to the posterior value of $\alpha 1$ whereas least $\alpha 1$ in the descendant refers to the value of $\alpha 1$ along a line segment.

```

CAdaptation:  $S_1 \rightarrow (D_1, s_{1,0}, \text{env}_1) \parallel (D_1', s_{1,0}', \text{env}_1') =$ 
 $S_2 \rightarrow (D_2, s_{2,0}, \text{env}_2) \parallel (D_2', s_{2,0}', \text{env}_2')$ 
match  $S_1$  with
 $S_1 \rightarrow \text{envlet } x = v_1^x \text{ and } r = v_1^r \text{ and } D_1' \text{ every } \text{env}$ 
 $S_2 \rightarrow x \text{ envlet } x = v_2^x \text{ and } r = v_2^r \text{ and } D_2' \text{ every } \text{env}$ 
and
match  $S_2$  with
 $S_2 \rightarrow \text{envlet } \text{dis} = v_2^{\text{dis}} \text{ and } \text{tr} = v_2^{\text{tr}} \text{ and } D_2' \text{ every } \text{env}$ 
 $S_2 \rightarrow \text{envlet } \text{sp} = v_2^{\text{sp}} \text{ and } \text{sp}' = v_2^{\text{sp}'}$ 
and  $D_2'$  every  $\text{env}$ 
and,  $\text{dis}(\text{dis}) \text{env} = \text{dis}$ ,  $\text{tr}(\text{tr}) = \text{tr}$ ,  $\text{sp}(\text{sp}) = \text{sp}$ ,  $\text{sp}'(\text{sp}') = \text{sp}'$ ,  $\text{env}(\text{env}) = \text{env}$ ,  $\text{env}'(\text{env}') = \text{env}'$ 
and  $\text{clock}(\text{tr}) = \text{Fix}(\text{env}) \parallel \text{Fix}(\text{env}')$ 
where  $\forall n, n', n'', n''', \text{env} \vdash \text{Fix}(\text{env}) \parallel \text{Fix}(\text{env}') \vdash n$ 
 $\text{envlet } \text{tr} = v_2^{\text{tr}}, \text{sp} = v_2^{\text{sp}}, \text{sp}' = v_2^{\text{sp}'}$ 
 $\text{envlet } \text{tr} = v_2^{\text{tr}}, \text{sp} = v_2^{\text{sp}}, \text{sp}' = v_2^{\text{sp}'}$ 

```

Figure 6: The translation of *argument*²

possible to infer and hence strongly disjunctive notation, that is, to cover more than two hypotheses. This is a key difference with the SYNTAX and SYNTAX2, and largely simplifies program understanding and analysis.

3.2.2 The Type System

We should first attend the typing rule for the new programming constructs. The typing rule should ensure the programmer estimates such that it gives the same types as the translation of the translation. These rules state in particular the typing of the translation. These rules state in particular the typing of the translation. These rules state in particular the typing of the translation.

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]

Figure 5: The translation of switch.

also, this code is translated into:

```

state = 0
step = 1 -> (step <= state) ? 1 : 0
step = 1 -> (step <= state) ? 1 : 0
step = 1 -> (step <= state) ? 1 : 0

```

The code highlights the fact that step is set in the first iteration and then remains constant.

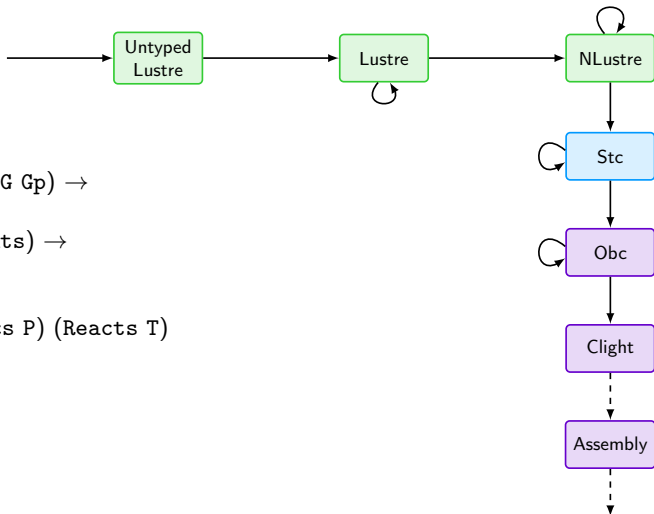
Figure 6: The translation of merge.

possible to define more strongly typed constructs, that is, to ensure that the type system is conservative. That is, a type system that is conservative means that it does not allow any operation that is not allowed by the hardware. This is achieved by the use of the type system. The type system is defined as follows:

- sampling explicited by **when**
- choice explicited by **merge**
- constants are also sampled
- only **reset** blocks remain

Main Correctness Theorem

Theorem `behavior_asm`:

$$\begin{aligned} &\forall D \ G \ Gp \ P \ \text{main} \ \text{ins} \ \text{outs}, \\ &\quad \text{elab_declarations } D = \text{OK} \ (\text{exist_} _ \ G \ Gp) \rightarrow \\ &\quad \text{compile } D \ \text{main} = \text{OK} \ P \rightarrow \\ &\quad \text{sem_node } G \ \text{main} \ (\text{pStr} \ \text{ins}) \ (\text{pStr} \ \text{outs}) \rightarrow \\ &\quad \text{wt_ins } G \ \text{main} \ \text{ins} \rightarrow \\ &\quad \text{wc_ins } G \ \text{main} \ \text{ins} \rightarrow \\ &\quad \exists T, \ \text{program_behaves} \ (\text{Asm.semantics } P) \ (\text{Reacts } T) \\ &\quad \wedge \ \text{bisim_IO } G \ \text{main} \ \text{ins} \ \text{outs} \ T. \end{aligned}$$


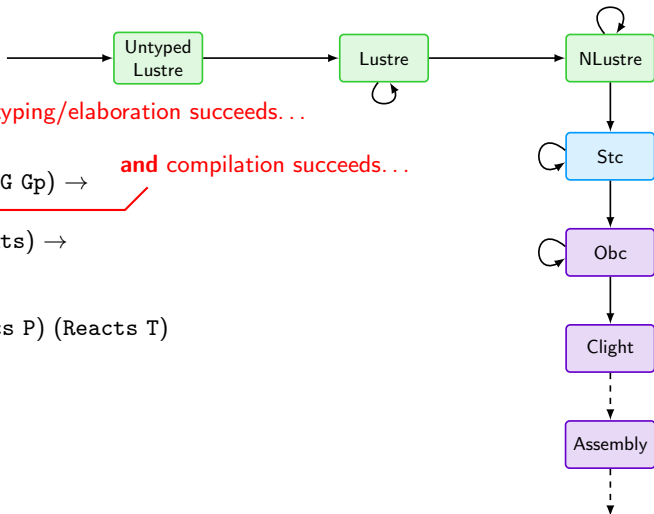
Main Correctness Theorem

Theorem behavior_asm:

$$\begin{aligned} &\forall D \ G \ Gp \ P \ \text{main ins outs}, \\ &\quad \text{elab_declarations } D = \text{OK } (\text{exist_ } G \ Gp) \rightarrow \\ &\quad \text{compile } D \ \text{main} = \text{OK } P \rightarrow \\ &\quad \text{sem_node } G \ \text{main } (pStr \ \text{ins}) \ (pStr \ \text{outs}) \rightarrow \\ &\quad wt_ins \ G \ \text{main ins} \rightarrow \\ &\quad wc_ins \ G \ \text{main ins} \rightarrow \\ &\quad \exists T, \ \text{program_behaves } (\text{Asm.semantics } P) \ (\text{Reacts } T) \\ &\quad \wedge \text{bisim_IO } G \ \text{main ins outs } T. \end{aligned}$$

if typing/elaboration succeeds...

and compilation succeeds...

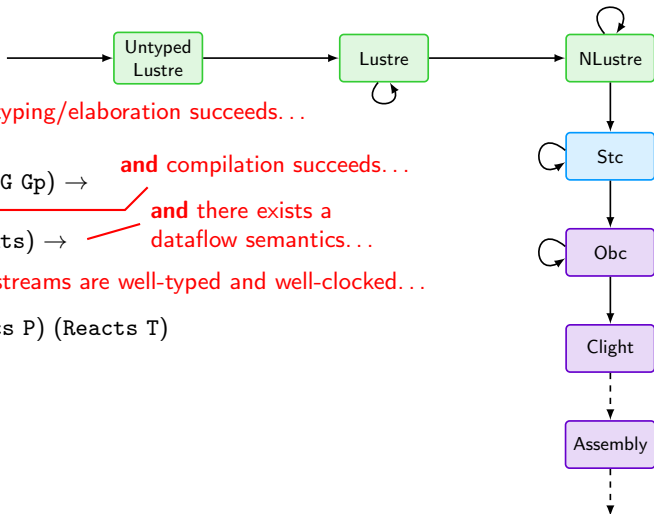


Main Correctness Theorem

Theorem behavior_asm:

$\forall D \ G \ Gp \ P \ main \ ins \ outs,$
 $elab_declarations \ D = OK \ (exist_G \ Gp) \rightarrow$
 $compile \ D \ main = OK \ P \rightarrow$
 $sem_node \ G \ main \ (pStr \ ins) \ (pStr \ outs) \rightarrow$
 $wt_ins \ G \ main \ ins \rightarrow$
 $wc_ins \ G \ main \ ins \rightarrow$
 $\exists T, \ program_behaves \ (Asm.semantics \ P) \ (Reacts \ T)$
 $\wedge \ bisim_IO \ G \ main \ ins \ outs \ T.$

if typing/elaboration succeeds...
and compilation succeeds...
and there exists a dataflow semantics...
and input streams are well-typed and well-clocked...



Main Correctness Theorem

Theorem `behavior_asm`:

$\forall D \ G \ Gp \ P \ main \ ins \ outs,$
 $elab_declarations \ D = OK \ (exist_G \ Gp) \rightarrow$
 $compile \ D \ main = OK \ P \rightarrow$
 $sem_node \ G \ main \ (pStr \ ins) \ (pStr \ outs) \rightarrow$
 $wt_ins \ G \ main \ ins \rightarrow$
 $wc_ins \ G \ main \ ins \rightarrow$
 $\exists T, \ program_behaves \ (Asm.semantics \ P) \ (Reacts \ T)$
 $\wedge \ bisim_IO \ G \ main \ ins \ outs \ T.$

if typing/elaboration succeeds...

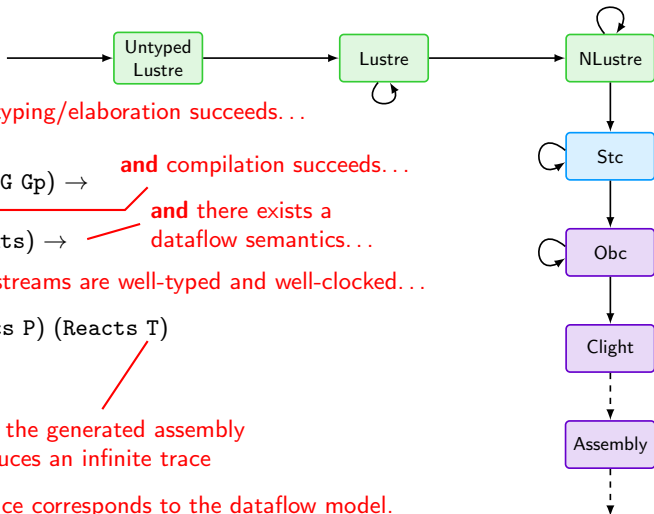
and compilation succeeds...

and there exists a dataflow semantics...

and input streams are well-typed and well-clocked...

then the generated assembly produces an infinite trace

and the trace corresponds to the dataflow model.



Dataflow relational semantics

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

Dataflow relational semantics

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

| | | | | | | | | | | | | | |
|-------|---|---|----|-----|-----|---|-----|-----|-----|-----|-----|-----|-----|
| 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

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

| | | | | | | | | | | | | | |
|-------|---|---|----|-----|-----|---|-----|-----|-----|-----|-----|-----|-----|
| 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

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

| | | | | | | | | | | | | | |
|-------|---|---|----|-----|-----|---|-----|-----|-----|-----|-----|-----|-----|
| 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 | ... |

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

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

Dataflow relational semantics

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

| | | | | | | | | | | | | | |
|-------|---|---|----|-----|-----|---|-----|-----|-----|-----|-----|-----|-----|
| 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 | ... |

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

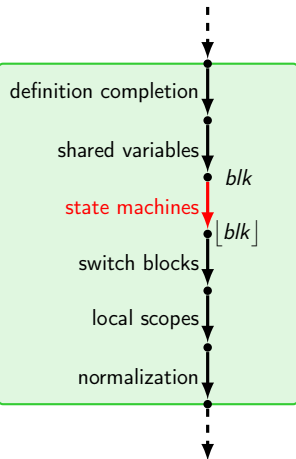
$$\frac{G, H \vdash e \Downarrow [vs] \quad \forall i, G, (\text{when}^{C_i} vs H) \vdash blks_i}{G, H \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}} \text{sem_switch}$$

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

Compilation correctness – state machines

Lemma (State machines correctness)

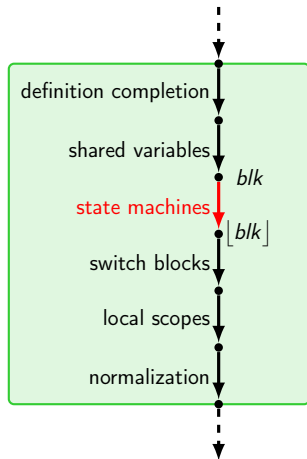
if $G, H \vdash blk$ then $G, H \vdash [blk]$



Compilation correctness – state machines

Lemma (State machines correctness)

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



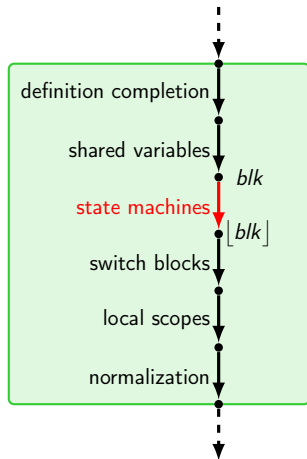
Works well:

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

Compilation correctness – state machines

Lemma (State machines correctness)

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



Works well:

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

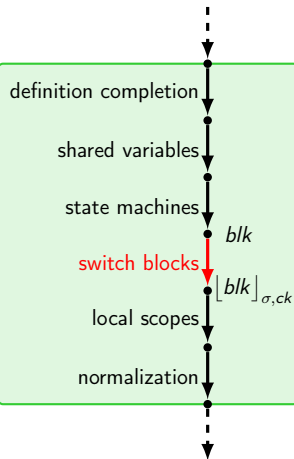
Works less well:

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

Compilation correctness – switch blocks

Lemma (Switch correctness)

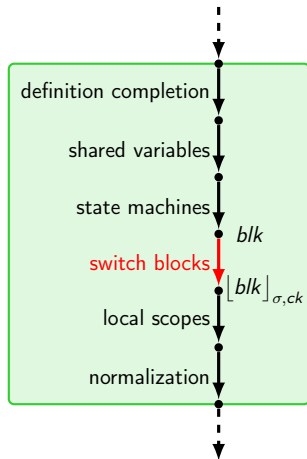
if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_{\sigma} H_2$ then $G, H_2 \vdash [blk]_{\sigma, ck}$



Compilation correctness – switch blocks

Lemma (Switch correctness)

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



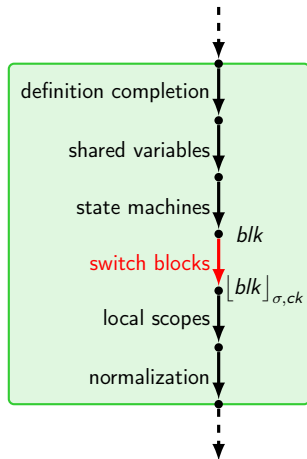
Works less well:

- 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 $G, H_2 \vdash [blk]_{\sigma, ck}$



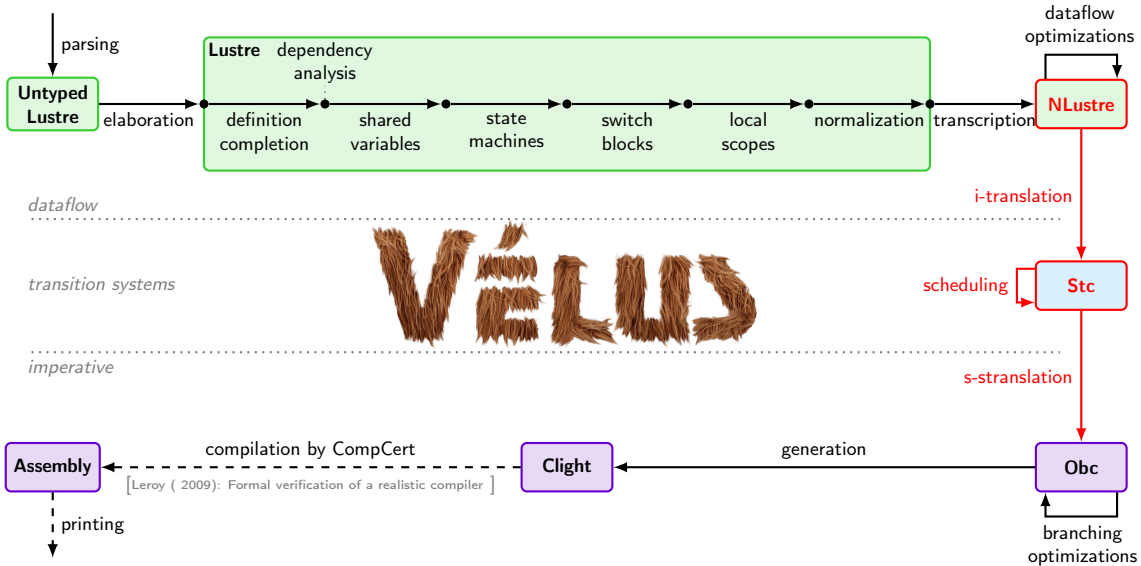
Works well:

- 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

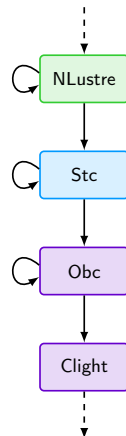
The Vélus Compiler



Generation of imperative code

```
resS = res when (st=Starting);  
reset  
  stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every resS;  
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 resS;
```

```
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;
```

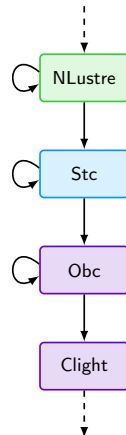
```
    break;
```

```
  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 resS;
```

```
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;
```

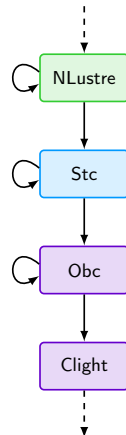
```
    break;
```

```
  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 resS;  
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;
```

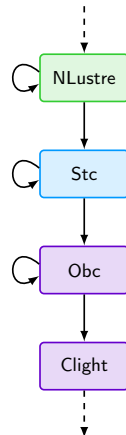
```
    break;
```

```
  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 resS;  
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;
```

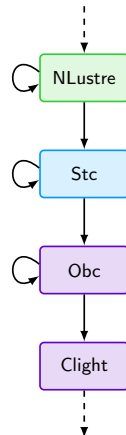
```
    break;
```

```
  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

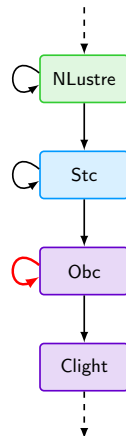


Generation of imperative code

```
resS = res when (st=Starting);  
reset  
  stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every resS;  
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: ...  
}
```

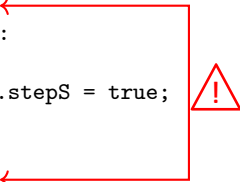
- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



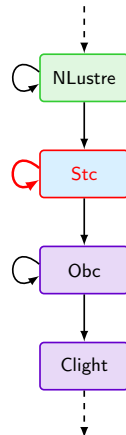
Generation of imperative code

```
resS = res when (st=Starting);  
reset  
  stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every resS;  
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

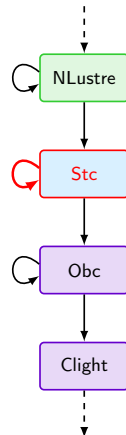


Generation of imperative code

```
resS = res when (st=Starting);  
reset  
  stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every resS;  
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

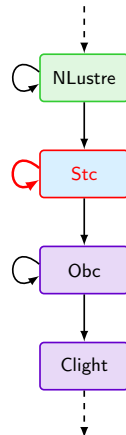


Generation of imperative code

```
resS = res when (st=Starting);  
reset  
  stepS = (true when (st=Starting)) fby (false when (st=Starting))  
every resS;  
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>

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

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

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

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

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

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

Semantics – Hierarchical State Machines

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

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

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

Semantics – Transitions

$$\begin{array}{c}
 G, H, bs \vdash e \Downarrow [ys] \\
 \text{bools-of } ys \equiv bs' \quad G, H, bs \vdash_I \text{autinits} \Downarrow sts \\
 sts' \equiv \text{first-of}_F^C bs' sts \\
 \hline
 G, H, bs \vdash_I C \text{ if } e; \text{autinits} \Downarrow sts'
 \end{array}$$

$$\begin{array}{c}
 sts \equiv \text{const } bs (C, F) \\
 \hline
 G, H, bs \vdash_I \text{otherwise } C \Downarrow sts
 \end{array}$$

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

$$\begin{array}{c}
 sts \equiv \text{const } bs (C_i, F) \\
 \hline
 G, H, bs, C_i \vdash_{\text{TR}} \epsilon \Downarrow sts
 \end{array}$$

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

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

Semantics – local blocks and last variables

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

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

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

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