Verified Compilation of Synchronous Dataflow with State Machines

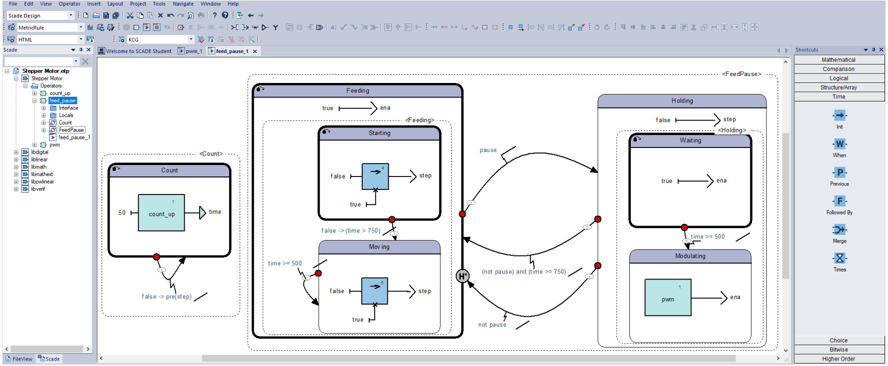
Timothy Bourke   Basile Pesin   Marc Pouzet

Inria Paris

École normale supérieure, CNRS, PSL University

ESWEEK 2023 - EMSOFT
Monday, September 18
11:22am - 11:47am CET

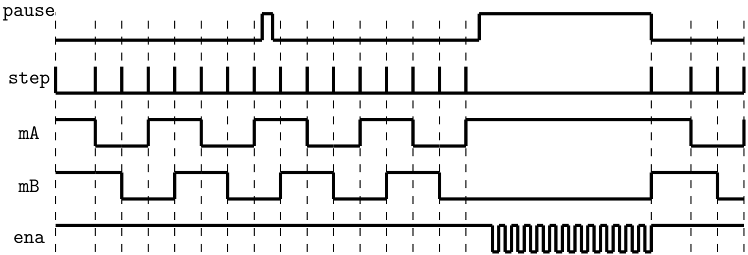
Block-Diagram Languages for Embedded Systems

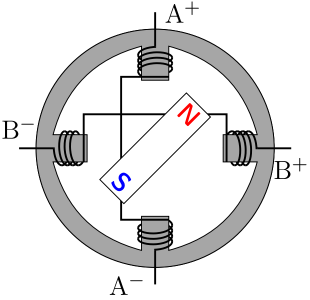


- Widely used in safety-critical applications:
  Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear. . .
- Scade 6: Qualified compiler for Lustre + Control Structures
- Our work: Verified compilation in an Interactive Theorem Prover (Coq)

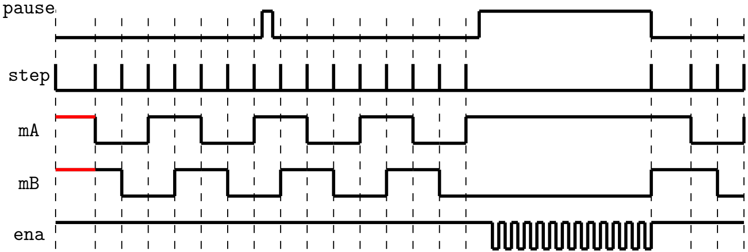
An system example: stepper motor for a small printer

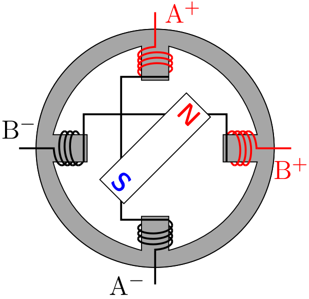
An system example: stepper motor for a small printer



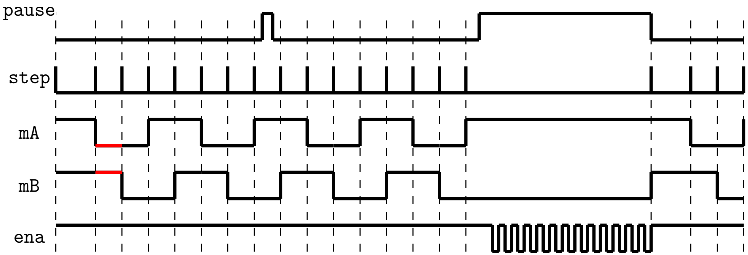


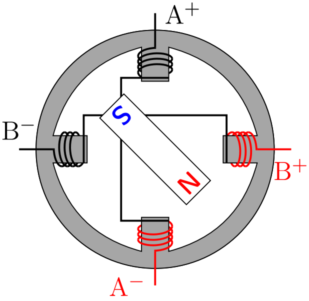
An system example: stepper motor for a small printer

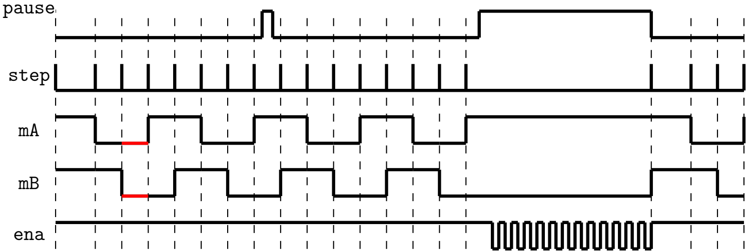


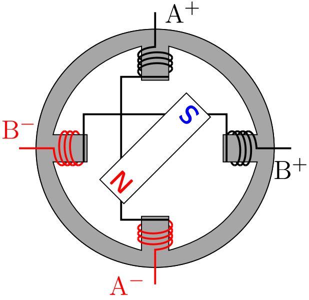


An system example: stepper motor for a small printer

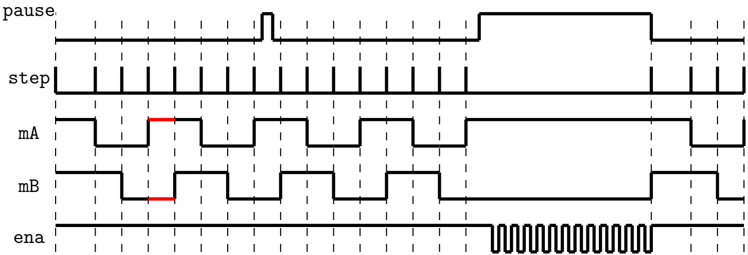


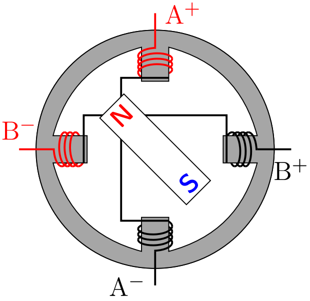


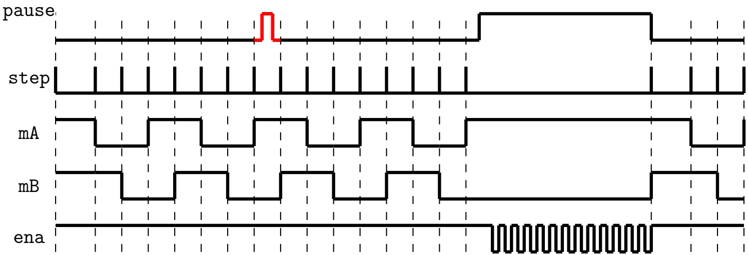

pause
step
mA
mB
ena

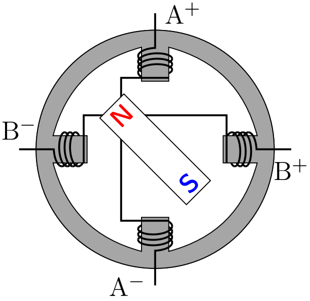

A+
B−
B+
S
N
A−

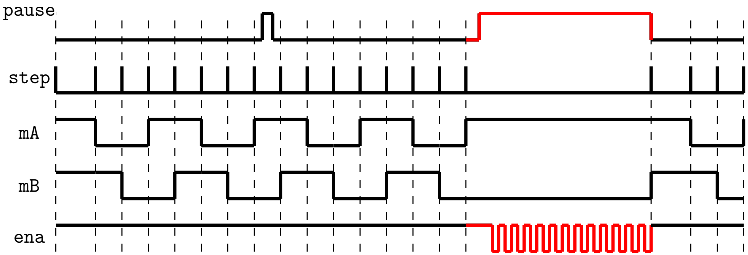
An system example: stepper motor for a small printer

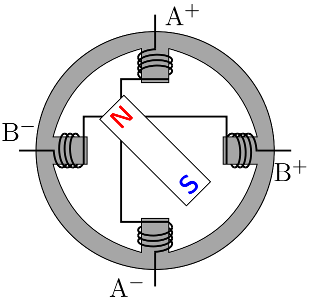




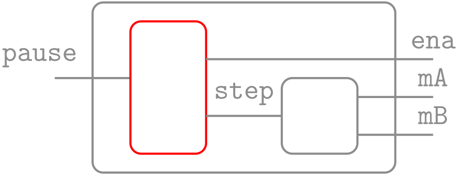

pause
step
mA
mB
ena

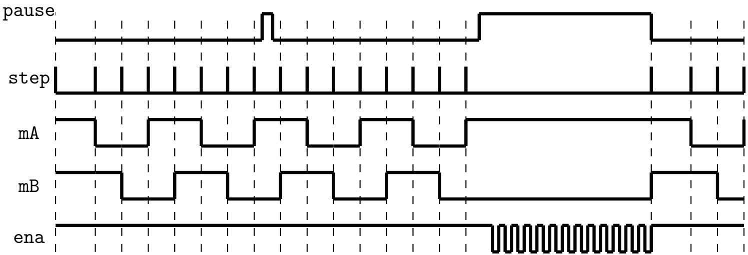

A+
B−
B+
A−
N
S

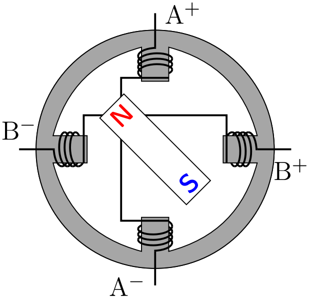

pause
step
mA
mB
ena


A^+
B^-
N
S
B^+
A^-

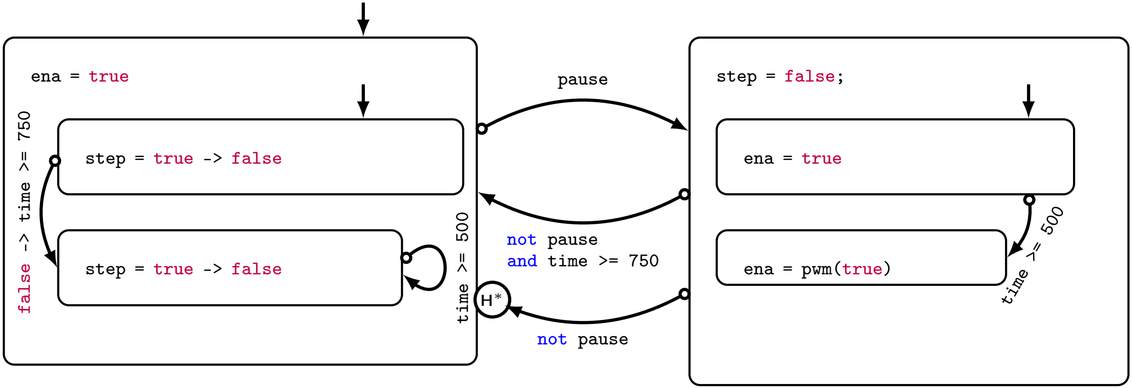
An system example: stepper motor for a small printer





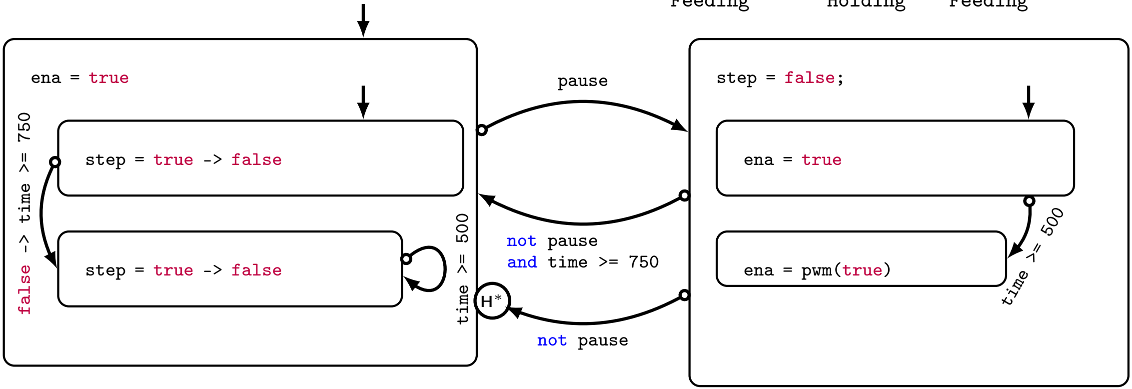


Hierarchical State Machines – Example



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			

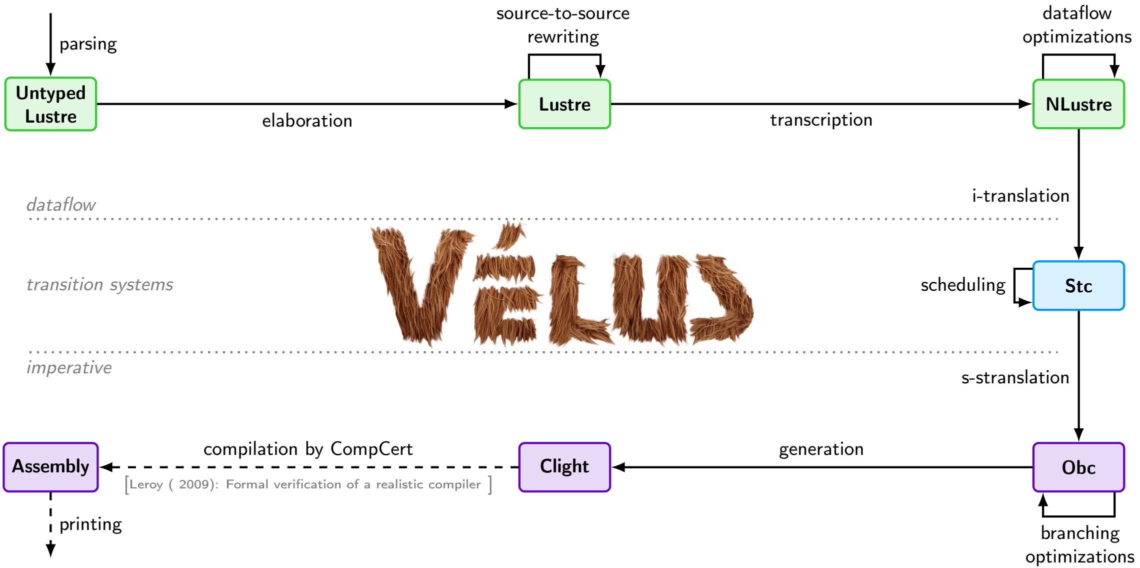


Hierarchical State Machines – Example

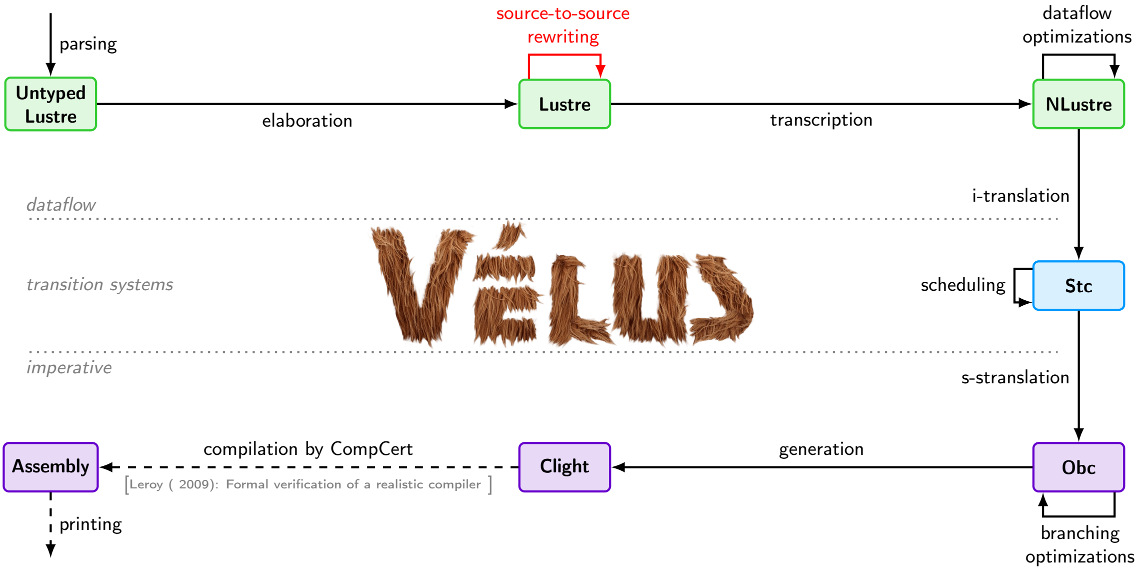
```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);

  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

  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
  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			

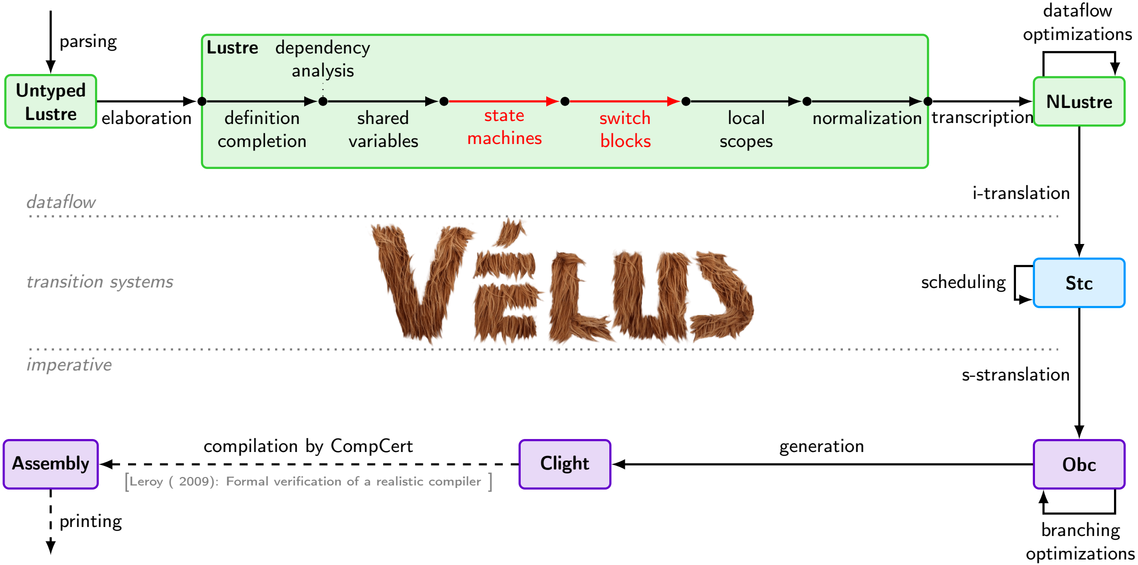
The Vélus Compiler



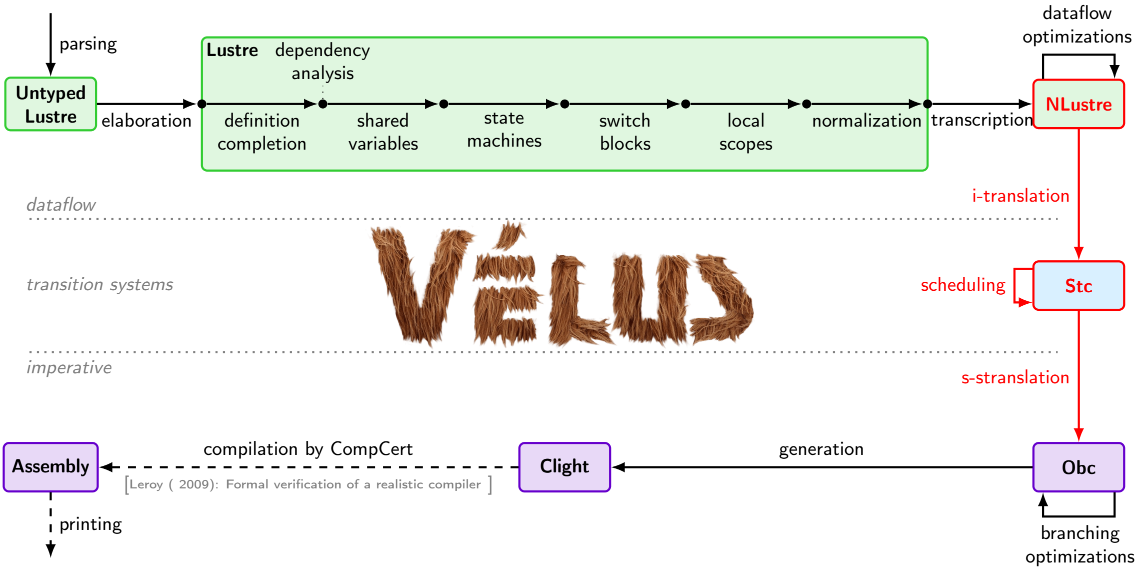
The Vélus Compiler



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

  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
  end
tel
```

H*

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

  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
  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]

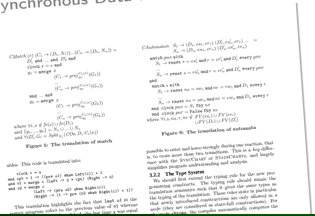


Figure 5: The translation of match

ables. This code is translated into:

```
clock c = x
and cpt = 1 -> ((pre o1) when Left(c)) + 1
and o1 = merge c (Left -> 2 * cpt) (Right -> 0)
and o2 = merge c
      (Left -> (pre o2) when Right(c))
      (Right -> (1 -> pre (o2 when Right(c)) + 1))
```

This translation highlights the fact that last o1 in the source program refers to the previous value of o1 whereas

Figure 6: The translation of automata

possible to enter and leave strongly during one reaction, that is, to cross more than two transitions. This is a *key* difference with the SyncChart or StateCharts, and largely simplifies program understanding and analysis.

3.2.2 The Type System

We should first extend the typing rule for the new programming constructs. The typing rule should mimic the translation semantics such that it gives the same types as the typing of the translation. These rules state in particular that newly introduced constructions are only allowed in a node (they are considered as state-full constructions). For the node chrono, the compiler automatically computes 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
```

```
(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
```

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

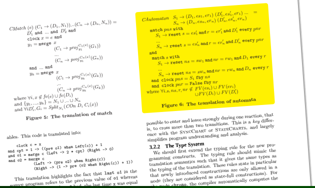


Figure 5: The translation of match

Figure 6: The translation of automata

3.2.2 The Type System

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
```

```
(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
```

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

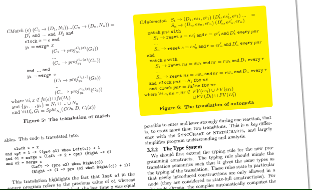


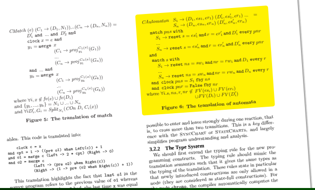
Figure 5: The translation of match

Figure 6: The translation of automata

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]




```
(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
```

```
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]

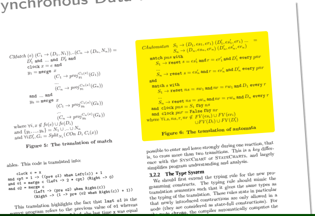


Figure 5: The translation of match

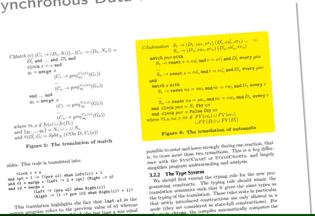
Figure 6: The translation of automata

```
(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]



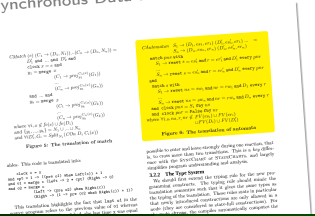

```
(pst, pres) = (Starting, false) fby (st, res);
switch pst
| Starting do
  reset
    (st, res) =
      if false -> time >= 750
      then (Moving, true)
      else (Starting, false)
  every pres
| Moving do ...
end;
switch st
| Starting do
  reset
    step = true -> false
  every res
| Moving do ...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

```
(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
```

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



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]

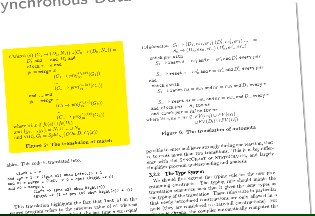


Figure 5: The translation of match

Figure 6: The translation of automata

- sampling explicited 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]

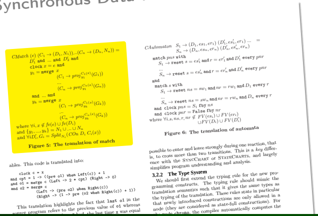


Figure 5: The translation of match

Figure 6: The translation of automata

- 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;
```

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

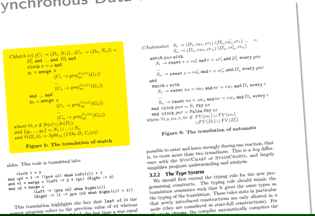


Figure 5: The translation of match

Figure 6: The translation of automata

3.2.2 The Type System

- sampling explicited by when
- choice explicited 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]

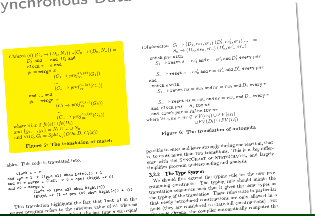
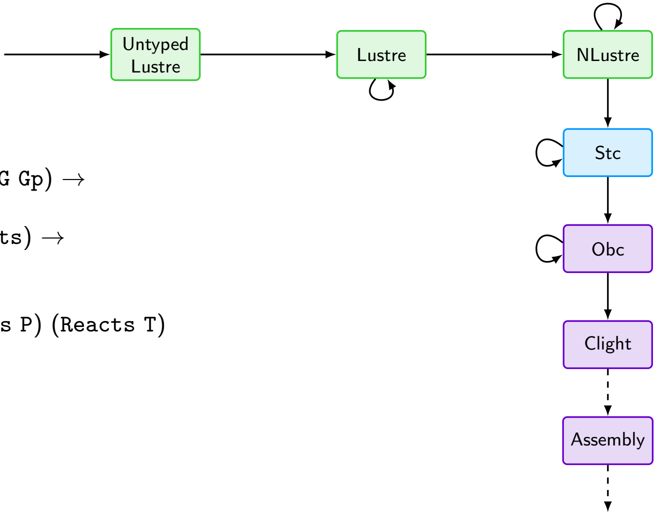


Figure 5: The translation of match

Figure 6: The translation of automata

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

Main Correctness Theorem



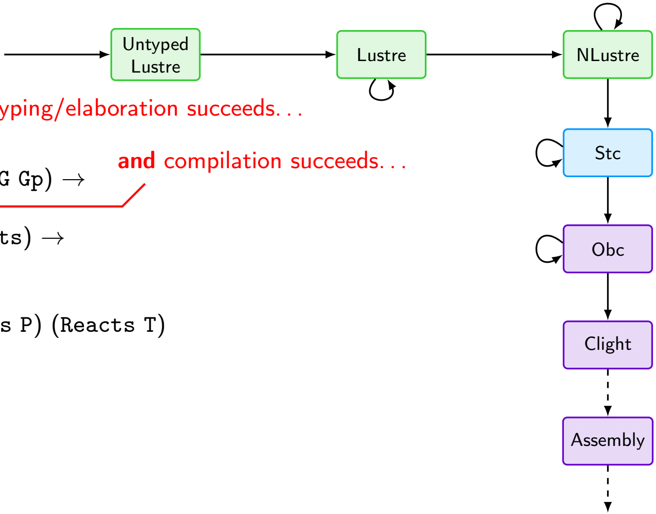

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    compile D main = OK P →
    sem_node G main (pStr ins) (pStr outs) →
    wt_ins G main ins →
    wc_ins G main ins →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_IO G main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

Main Correctness Theorem

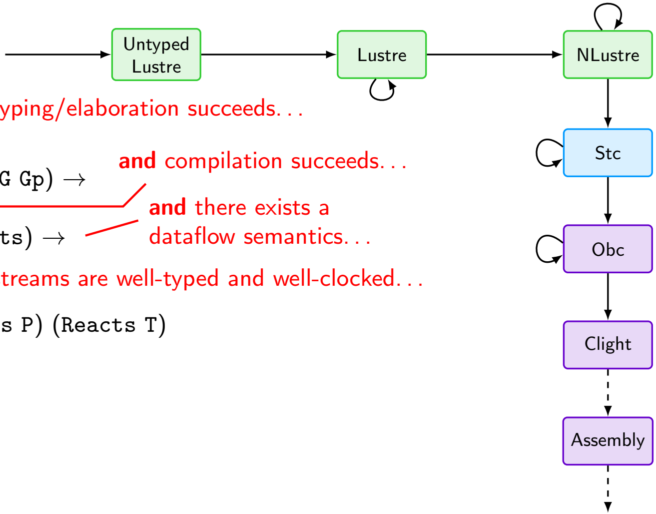
```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    compile D main = OK P →
    sem_node G main (pStr ins) (pStr outs) →
    wt_ins G main ins →
    wc_ins G main ins →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
          ∧ bisim_IO G main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

if typing/elaboration succeeds. . .

and compilation succeeds. . .



Main Correctness Theorem




```
Theorem behavior_asm:
  ∀ 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 →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
          ∧ bisim_IO G main ins outs T.
```

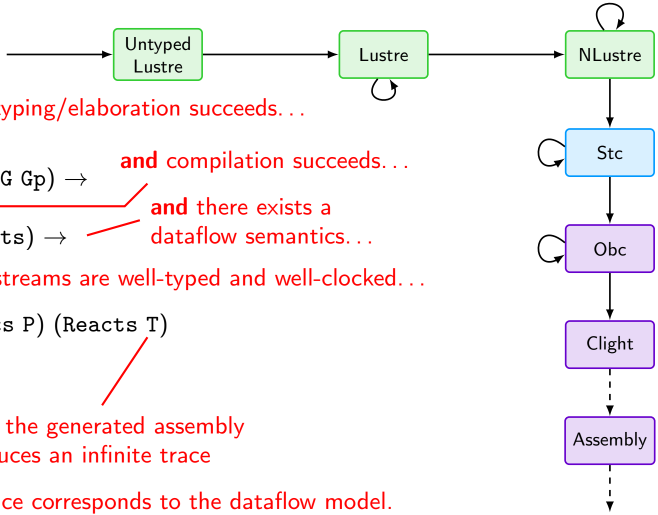
if typing/elaboration succeeds. . .

and compilation succeeds. . .

and there exists a dataflow semantics. . .

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

Main Correctness Theorem




```
Theorem behavior_asm:
  ∀ 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 →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
          ∧ 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

$$\frac{G(f) = \texttt{node}\ f\,(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \qquad \forall i, H(x_i) \equiv xss_i \qquad \forall j, H(y_j) \equiv yss_j \qquad G, H \vdash blk}{G \vdash f(xss) \Downarrow yss}\ \texttt{sem\_node}$$

Dataflow relational semantics

$$\frac{G(f) = \texttt{node}\ f\,(x_1,\ldots,x_n)\ \texttt{returns}\ (y_1,\ldots,y_m)\ blk \qquad \forall i, H(x_i) \equiv xss_i \qquad \forall j, H(y_j) \equiv yss_j \qquad G, H \vdash blk}{G \vdash f(xss) \Downarrow yss}\ \texttt{sem\_node}$$

`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

$$\frac{G(f) = \texttt{node}\ f\,(x_1,\ldots,x_n)\ \texttt{returns}\ (y_1,\ldots,y_m)\ blk \qquad \forall i, H(x_i) \equiv xss_i \qquad \forall j, H(y_j) \equiv yss_j \qquad G, H \vdash blk}{G \vdash f(xss) \Downarrow yss}\ \texttt{sem\_node}$$

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \qquad G, H \vdash es \Downarrow vss}{G, H \vdash xs\ =\ es}\ \texttt{sem\_equation}$$

Dataflow relational semantics

$$\frac{G(f) = \texttt{node}\ f\,(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \qquad \forall i, H(x_i) \equiv xss_i \qquad \forall j, H(y_j) \equiv yss_j \qquad G, H \vdash blk}{G \vdash f(xss) \Downarrow yss}\ \texttt{sem\_node}$$

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \qquad G, H \vdash es \Downarrow vss}{G, H \vdash xs\ =\ es}\ \texttt{sem\_equation}$$

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

Dataflow relational semantics

$$\frac{G(f) = \texttt{node}\ f\,(x_1,\ldots,x_n)\ \texttt{returns}\ (y_1,\ldots,y_m)\ blk \qquad \forall i, H(x_i) \equiv xss_i \qquad \forall j, H(y_j) \equiv yss_j \qquad G, H \vdash blk}{G \vdash f(xss) \Downarrow yss}\ \texttt{sem\_node}$$

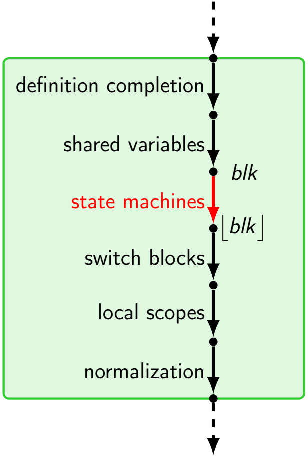
pause	F	F	F	...	F	F	...	T	...	F	...	F	...
time	0	0	50	...	750	0	...	150	...	350	...	500	...
step	T	F	F	...	T	F	...	F	...	F	...	T	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \qquad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es}\ \texttt{sem\_equation}$$

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

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

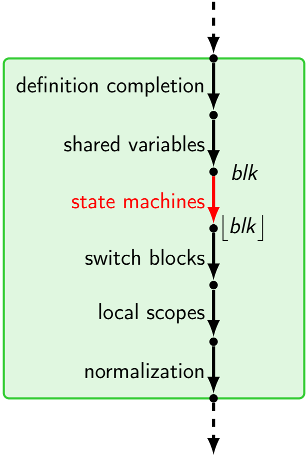
Compilation correctness – state machines



Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Compilation correctness – state machines



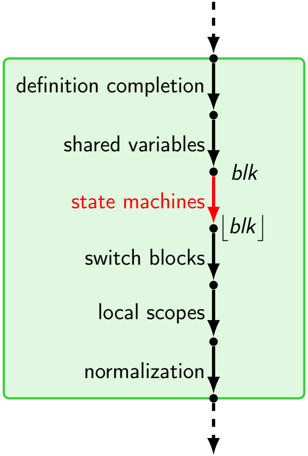
Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Works well:

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

Compilation correctness – state machines



Lemma (State machines correctness)

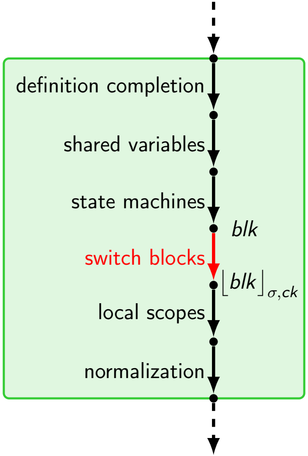
$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Works well:

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

Works less well:

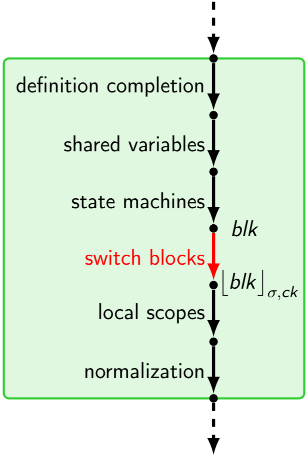
- static invariants (typing, clock-typing, . . .)
- fresh identifiers



Lemma (Switch correctness)

if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_\sigma H_2$ then $G, H_2 \vdash \lfloor blk \rfloor_{\sigma,ck}$

Compilation correctness – switch blocks



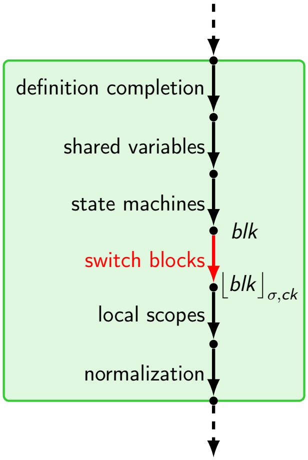
Lemma (Switch correctness)

if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_\sigma H_2$ then $G, H_2 \vdash \lfloor blk \rfloor_{\sigma,ck}$

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 $G, H_2 \vdash \lfloor blk \rfloor_{\sigma,ck}$

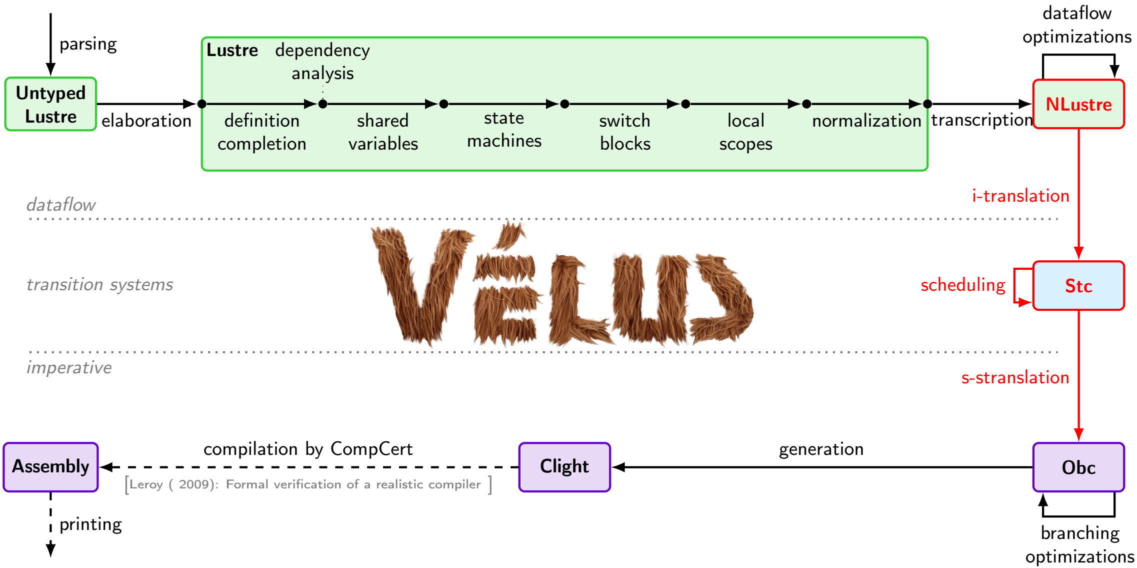
Works well:

- 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

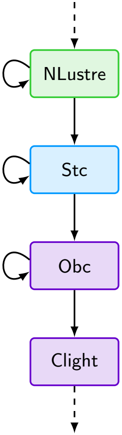
The Vélus Compiler



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]

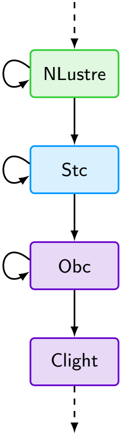


Generation of imperative code

```
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

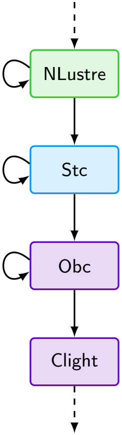


Generation of imperative code

```
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

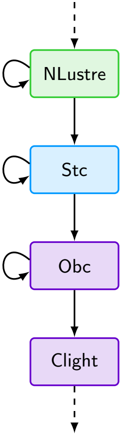


Generation of imperative code

```
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

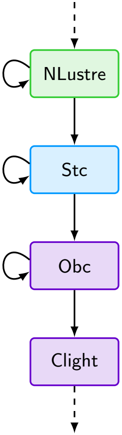


Generation of imperative code

```
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

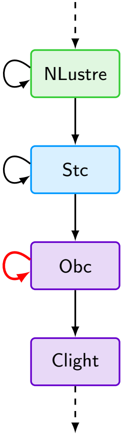


Generation of imperative code

```
            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

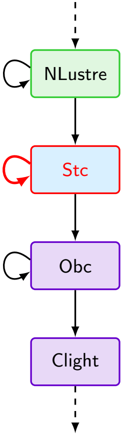


Generation of imperative code

```
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

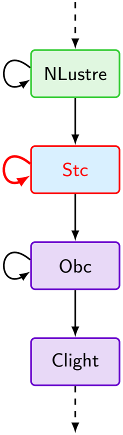


Generation of imperative code

```
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

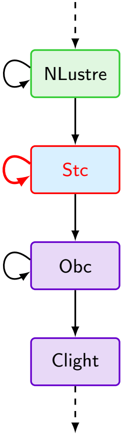


Generation of imperative code

```
          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

	Vélus	*Hept+CompCert*		*Hept+gcc*		*Hept+gcci*	
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

	Vélus	*Hept+CompCert*		*Hept+gcc*		*Hept+gcci*	
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

	Vélus	*Hept+CompCert*		*Hept+gcc*		*Hept+gcci*	
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 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

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

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

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

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

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

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

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

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

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

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \text{bools-of } ys \equiv bs' \qquad G, H, bs \vdash_{\mathrm{I}} autinits \Downarrow sts \qquad sts' \equiv \text{first-of}^{C}_{\mathtt{F}}\ bs'\ sts}{G, H, bs \vdash_{\mathrm{I}} C\ \mathtt{if}\ e; autinits \Downarrow sts'}$$

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

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

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

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

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

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

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

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

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