Formal verification of a code generator for a modeling language: the Velus project

Xavier Leroy
(joint work with Timothy Bourke, Lélio Brun,
Pierre-Évariste Dagand, Marc Pouzet, and Lionel Rieg)

Inria, Paris

MARS/VPT workshop, ETAPS 2018

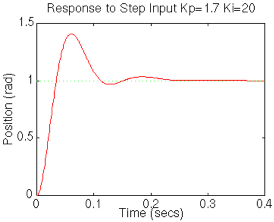
In this talk…

Velus is a formally-verified code generator,
producing C code from the Lustre modeling language,
connected with the CompCert verified C compiler.

Lustre is a declarative, synchronous language,
oriented towards cyclic control software,
usable for programming, modeling, and verification,
at the core of the SCADE suite from ANSYS/Esterel Technologies.

Control laws

"Hello, world" example: PID controller.



Error $e(t) = \text{desired state}(t) - \text{current state}(t)$.

$$\text{Action } a(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{d}{dt} e(t)$$

(Proportional) (Integral) (Derivative)

Implementing a control law

Mechanical (e.g. pneumatic):

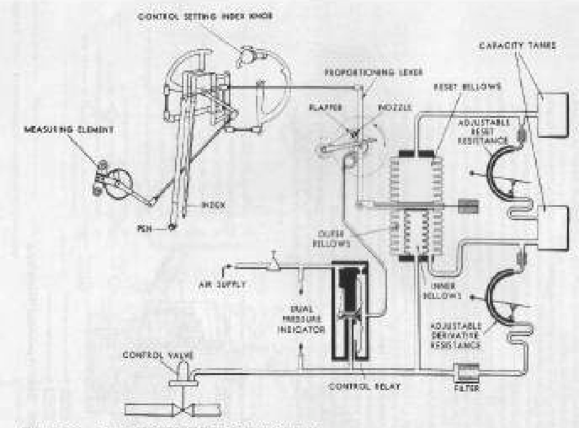
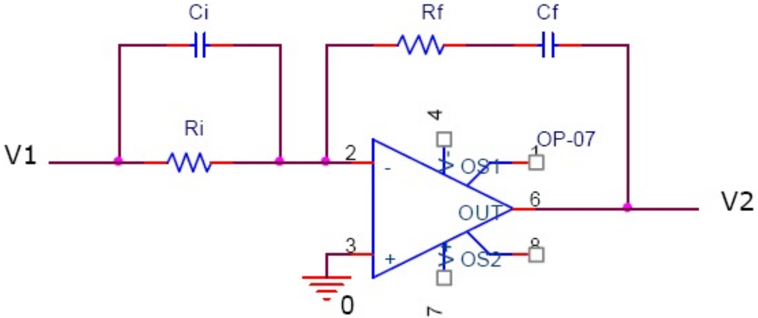


Figure 2-28A. Proportional-Plus-Reset-Plus-Derivative Controller

Implementing a control law

Analog electronics:



Implementing a control law

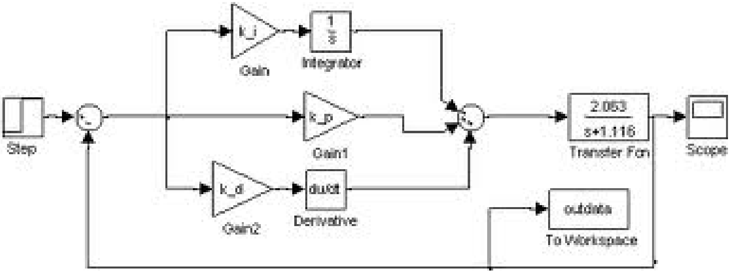
In software (today's favorite solution):

```
previous_error = 0; integral = 0
loop forever:
    error = setpoint - actual_position
    integral = integral + error * dt
    derivative = (error - previous_error) / dt
    output = Kp * error + Ki * integral + Kd * derivative
    previous_error = error
    wait(dt)
```

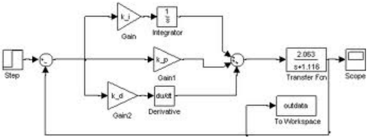
Block diagrams

(Simulink, Scade, Scicos, etc)

This kind of code is rarely hand-written, but rather auto-generated from block diagrams:



Block diagrams and reactive languages



In the case of Scade, this diagram is a graphical syntax for the Lustre reactive language:

```
error = setpoint - position
integral = (0 fby integral) + error * dt
derivative = (error - (0 fby error)) / dt
output = Kp * error + Ki * integral + Kd * derivative
```

(= Time-indexed series defined by recursive equations.)

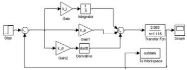
Block diagrams and reactive languages

Control law

$$a(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{d}{dt} e(t)$$

(modeling)

Block diagram



(discretization)

Recursive sequences

(syntax)

Lustre code

(semantics)

$$\begin{aligned} i_n &= i_{n-1} + e_n.dt \\ d_n &= (e_n - e_{n-1})/dt \\ o_n &= K_p e_n + K_i i_n + K_d d_n \end{aligned}$$

(code generation)

(hand-coding)

C code

Outline

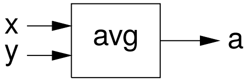
1. Prologue: control software and block diagrams
2. The Lustre reactive, synchronous language and its compilation
3. The Velus formally-verified Lustre compiler
4. Perspectives

Outline

Lustre: the dataflow core

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

```
node avg(x, y: real)
   returns (a: real)
let
  a = 0.5 * (x + y);
tel
```



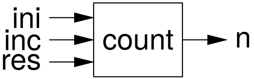
A node is a set of equations *var* $=$ *expr*. It defines a function between input and output streams.

Semantic model: streams of values, synchronized on time steps.

x	0	1	5	3	...
y	2	7	2	0	...
a	1	4	3.5	1.5	...

Lustre: temporal operators

```
node count(ini, inc: int; res: bool)
   returns (n: int)
let
  n = if (true fby false) or res
      then ini
      else (0 fby n) + inc
tel
```



cst `fby` *e* is the value of *e* at the previous time step, except at time 0 where it is *cst*.

`ini`	0	0	0	0	0	0	0	...
`inc`	0	1	2	1	2	3	0	...
`res`	F	F	F	F	T	F	F	...
`true fby false`	T	F	F	F	F	F	F	...
`0 fby n`	0	0	1	3	4	0	3	...
`n`	0	1	3	4	0	3	3	...

Lustre: derived temporal operators

a at the first time step and *b* forever after:

$$a \texttt{ -> } b \stackrel{\mathit{def}}{=} \texttt{if } (\texttt{true fby false}) \texttt{ then } a \texttt{ else } b$$

The value of *a* at the previous time step:

$$\texttt{pre}(a) \stackrel{\mathit{def}}{=} \texttt{nil fby } a$$

where `nil` is a default value of the correct type.

```
node count(ini, inc: int; res: bool)
   returns (n: int)
let
  n = if res then ini else ini -> (pre(n) + inc)
tel
```

Lustre: instantiation and sampling

```
node avgvelocity (delta: int; sec: bool)
         returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time = count((1, 1, false) when sec);
  v = merge sec ((dist when sec) / time)
                ((0 fby v) when not sec)
tel
```

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
dist	0	1	3	4	6	9	9	12	...

Lustre: instantiation and sampling

```
node avgvelocity (delta: int; sec: bool)
         returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time = count((1, 1, false) when sec);
  v = merge sec ((dist when sec) / time)
                ((0 fby v) when not sec)
tel
```

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
dist	0	1	3	4	6	9	9	12	...
time	-	-	-	1	-	2	3	-	...

Lustre: instantiation and sampling

```
node avgvelocity (delta: int; sec: bool)
         returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time = count((1, 1, false) when sec);
  v = merge sec ((dist when sec) / time)
                ((0 fby v) when not sec)
tel
```

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
dist	0	1	3	4	6	9	9	12	...
time	-	-	-	1	-	2	3	-	...
(dist when sec) / time	-	-	-	4	-	4	3	-	...

Lustre: instantiation and sampling

```
node avgvelocity (delta: int; sec: bool)
         returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time = count((1, 1, false) when sec);
  v = merge sec ((dist when sec) / time)
                ((0 fby v) when not sec)
tel
```

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
dist	0	1	3	4	6	9	9	12	...
time	-	-	-	1	-	2	3	-	...
(dist when sec) / time	-	-	-	4	-	4	3	-	...
(0 fby v) when not sec	0	0	0	-	4	-	-	3	...

Lustre: instantiation and sampling

```
node avgvelocity (delta: int; sec: bool)
         returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time = count((1, 1, false) when sec);
  v = merge sec ((dist when sec) / time)
                ((0 fby v) when not sec)
tel
```

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
dist	0	1	3	4	6	9	9	12	...
time	-	-	-	1	-	2	3	-	...
(dist when sec) / time	-	-	-	4	-	4	3	-	...
(0 fby v) when not sec	0	0	0	-	4	-	-	3	...
v	0	0	0	4	4	4	3	3	...

Compilation 1: normalization

Introduce a fresh variable for each `fby` expression, and lift the `fby` expression in its own equation.

Initial code:

```
node count(ini, inc: int; res: bool)
   returns (n: int)

let
  n = if (true fby false) or res
      then ini
      else (0 fby n) + inc;
tel
```

Normalized code:

```
var t: bool; u: int;
let
  t = true fby false;
  u = 0 fby n;
  n = if t or res
      then ini
      else u + inc;
tel
```

Trivia: the number of `fby` expressions is exactly the amount of memory used by the node.

Compilation 2: scheduling

Lustre nodes must be causal:

- No immediate dependency cycles such as `x = x + 1` or `x = y + 1; y = x - 1`.
- All dependency cycles must go through a `fby` node, as in `x = 0 fby (x + 1)`.

Scheduling a node consists in executing sequentially the computations of a node in a certain order (the schedule).

For a causal node, a schedule always exists. Some schedules may lead to more efficient compiled code than others.

Compilation 2: scheduling

For normalized nodes, scheduling is equivalent to ordering the equations so that

- normal variables are defined before being read;
- `fby` variables are read before being defined.

```
node count(ini, inc: int; res: bool)
   returns (n: int)
   var t: bool; u: int;
let
  t = true fby false;
  u = 0 fby n;
  n = if f or res
      then ini
      else t2 + inc;
tel
```

Not scheduled

```
let
  n = if t or res
      then ini
      else u + inc;
  t = true fby false;
  u = 0 fby n;
```

Scheduled

Compilation 3: translation to OO code

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

Each node becomes a class (in a small object-oriented intermediate language called Obc), with:

- One instance variable per `fby` variable, recording the current value of this variable.
- A `reset` method to initialize the instance variables at $t = 0$.
- A `step` method that takes inputs at time t, produces outputs at time t, and updates the instance variables for time $t + 1$.
- If the node calls other nodes, one instance variable per node called, recording its state.

Compilation 3: translation to OO code

```
node count(ini, inc: int;
           res: bool)
   returns (n: int)
   var t: bool; u: int;
let
  n = if t or res
      then ini
      else u + inc;
  t = true fby false;
  u = 0 fby n;
tel
```

```
class count {
  memory t: bool;
  memory u: int;

  reset() {
    this.t := true;
    this.u := 0;
  }
  step(ini:int, inc:int,
       res:bool)
  returns (n: int) {
    if (this.t | res)
      then n := ini
      else n := this.u + inc;
    this.t := false;
    this.u := n;
  }
}
```

Compilation 3: translation to OO code

```
node count(ini, inc: int;
           res: bool)
   returns (n: int)
   var t: bool; u: int;
let
  n = if t or res
      then ini
      else u + inc;
  t = true fby false;
  u = 0 fby n;
tel
```

```
class count {
  memory t: bool;
  memory u: int;

  reset() {
    this.t := true;
    this.u := 0;
  }
  step(ini:int, inc:int,
       res:bool)
  returns (n: int) {
    if (this.t | res)
      then n := ini
      else n := this.u + inc;
    this.t := false;
    this.u := n;
  }
}
```

Compilation 3: translation to OO code

```
node count(ini, inc: int;
           res: bool)
   returns (n: int)
   var t: bool; u: int;
let
  n = if t or res
      then ini
      else u + inc;
  t = true fby false;
  u = 0 fby n;
tel
```

```
class count {
  memory t: bool;
  memory u: int;

  reset() {
    this.t := true;
    this.u := 0;
  }
  step(ini:int, inc:int,
       res:bool)
  returns (n: int) {
    if (this.t | res)
      then n := ini
      else n := this.u + inc;
    this.t := false;
    this.u := n;
  }
}
```

Compilation 3: translation to OO code

```
node count(ini, inc: int;
           res: bool)
   returns (n: int)
   var t: bool; u: int;
let
  n = if t or res
      then ini
      else u + inc;
  t = true fby false;
  u = 0 fby n;
tel
```

```
class count {
  memory t: bool;
  memory u: int;

  reset() {
    this.t := true;
    this.u := 0;
  }
  step(ini:int, inc:int,
       res:bool)
  returns (n: int) {
    if (this.t | res)
      then n := ini
      else n := this.u + inc;
    this.t := false;
    this.u := n;
  }
}
```

Nesting of node instances

```
node avgvelocity (delta: int;
                  sec: bool)
  returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time =
    count((1, 1, false) when sec);
  v = ... ;
  w = 0 fby v;
tel
```

```
class avgvelocity {
  memory w: int;
  instance i1: count;
  instance i2: count;

  reset() {
    i1.reset();
    i2.reset();
    this.w := 0;
  }
  step(delta: int, sec:bool)
    returns (v: int)
  {
    dist := o1.step(0, delta, false);
    if (sec) then
      time := o2.step(1, 1, false);
    ...
    this.w := v;
  }
}
```

Nesting of node instances

```
node avgvelocity (delta: int;
                  sec: bool)
  returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time =
    count((1, 1, false) when sec);
  v = ... ;
  w = 0 fby v;
tel
```

```
class avgvelocity {
  memory w: int;
  instance i1: count;
  instance i2: count;

  reset() {
    i1.reset();
    i2.reset();
    this.w := 0;
  }
  step(delta: int, sec:bool)
    returns (v: int)
  {
    dist := o1.step(0, delta, false);
    if (sec) then
      time := o2.step(1, 1, false);
    ...
    this.w := v;
  }
}
```

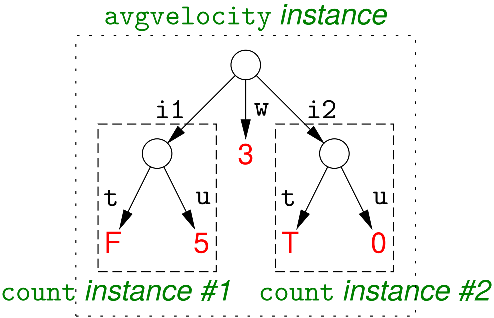
Nesting of node instances

```
node avgvelocity (delta: int;
                  sec: bool)
  returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time =
    count((1, 1, false) when sec);
  v = ... ;
  w = 0 fby v;
tel
```

```
class avgvelocity {
  memory w: int;
  instance i1: count;
  instance i2: count;

  reset() {
    i1.reset();
    i2.reset();
    this.w := 0;
  }
  step(delta: int, sec:bool)
    returns (v: int)
  {
    dist := o1.step(0, delta, false);
    if (sec) then
      time := o2.step(1, 1, false);
    ...
    this.w := v;
  }
}
```

The OBC memory model

A tree of node instances and sub-node instances, with values of instance variables at the leaves.



(Cf. objects and subobjects in C++.)

Compilation 4: production of C code

Standard encoding for an OO language without dynamic dispatch:

- Instance variables and subobjects are encoded as nested structs:

```
struct count { bool t; int u; };
struct avgvelocity { struct count i1, i2; int w; };
```

Compilation 4: production of C code

Standard encoding for an OO language without dynamic dispatch:

- Instance variables and subobjects are encoded as nested structs:

```
struct count { bool t; int u; };
struct avgvelocity { struct count i1, i2; int w; };
```

- `reset` and `step` functions take a `this` parameter by in-out reference.

```
void count_reset(struct count * this /*inout*/);
void count_step (struct count * this /*inout*/,
                 int ini, int step, bool res,
                 int * n /*out*/);
```

Compilation 4: production of C code

Standard encoding for an OO language without dynamic dispatch:

- Instance variables and subobjects are encoded as nested structs:

```
struct count { bool t; int u; };
struct avgvelocity { struct count i1, i2; int w; };
```

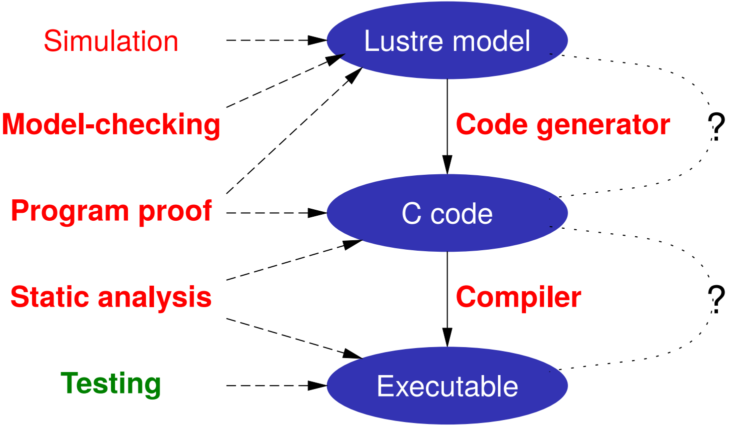
- `reset` and `step` functions take a `this` parameter by in-out reference.

```
void count_reset(struct count * this /*inout*/);
void count_step (struct count * this /*inout*/,
                 int ini, int step, bool res,
                 int * n /*out*/);
```

- Results for `step` functions are passed by out reference.

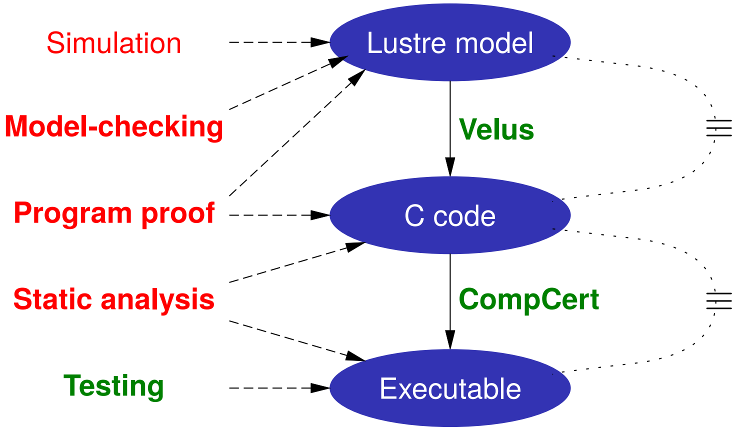
Outline

Trust in compilers and code generators



The miscompilation risk: wrong code is generated from a correct Lustre model. Casts doubts on model-level formal verification.

Trust in compilers and code generators



Formally-verified compilers and code generators rule out mis-compilation and generate trust in formal verification.

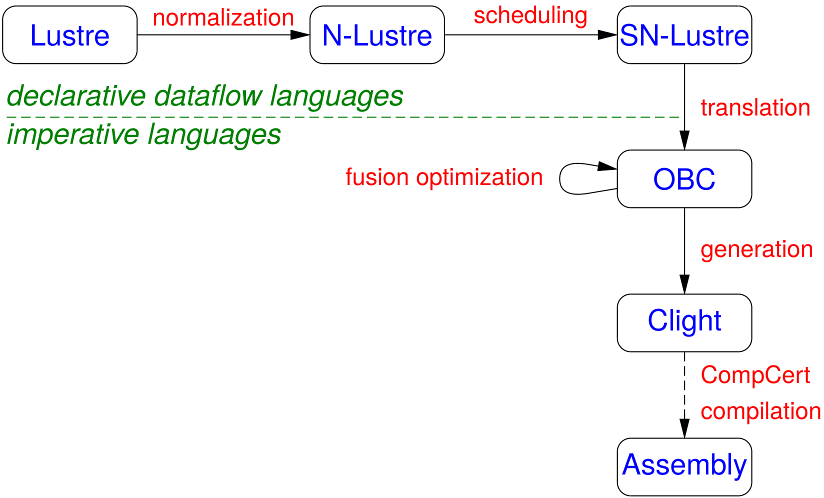
The Velus formally-verified code generator for Lustre

The Velus project, led by Timothy Bourke, develops and proves correct a code generator for the core Lustre language:

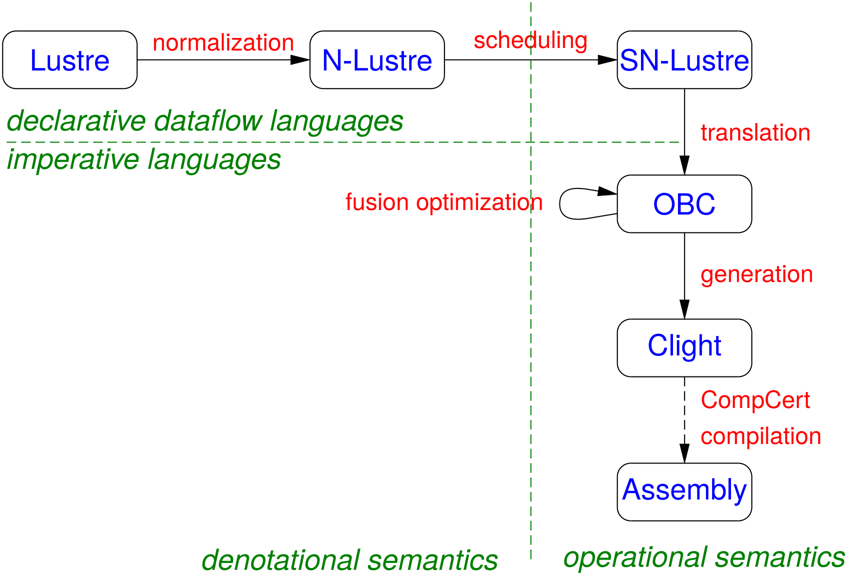
- Target language: the CompCert Clight subset of C.
- Compilation strategy: the modular approach from part 2.
- Optimizations: just one so far (`if` fusion).
- Verification: Coq proof of semantic preservation.

Same methodology as CompCert: most of the compiler is written in Coq's specification language, then extracted to OCaml for execution.

Velus languages and passes



Velus languages and passes



Proof outline 1: normalization

Initial code:

```
node count(ini, inc: int; res: bool)
   returns (n: int)

let
  n = if (true fby false) or res
      then ini
      else (0 fby n) + inc;
tel
```

Normalized code:

```
var t: bool; u: int;
let
  t = true fby false;
  u = 0 fby n;
  n = if t or res
      then ini
      else u + inc;
tel
```

Denotational semantics: for every node there exists a solution ϕ : *var* $\rightarrow$ *stream* of the equations.

Substitution (of *var* by *exp* if *var* $=$ *exp* is an equation) is valid in this semantics.

Proof outline 2: scheduling

```
node count(ini, inc: int; res: bool)
   returns (n: int)
   var t: bool; u: int;
let
  t = true fby false;
  u = 0 fby n;
  n = if t or res
      then ini
      else u + inc;
tel
```

Not scheduled

```
let
  n = if t or res
      then ini
      else u + inc;
  t = true fby false;
  u = 0 fby n;
```

Scheduled

The denotational semantics is insensitive to the order of equations.

Scheduled nodes have an operational semantics *exp* $\rightarrow$ current value $\times$ residual exp from which we can construct a solution to the equations.

Proof outline 3: translation to OO code

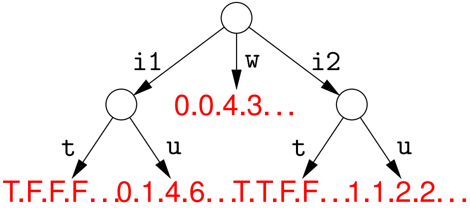
```
node avgvelocity (delta: int;
                  sec: bool)
  returns (v: int)
  var dist, time: int
let
  dist = count(0, delta, false);
  time =
    count((1, 1, false) when sec);
  ...
tel
```

```
class avgvelocity {
  memory w: int;
  instance i1: count;
  instance i2: count;

  reset() { ... }

  step(delta: int, sec:bool)
    returns (v: int)
  { ... }
}
```

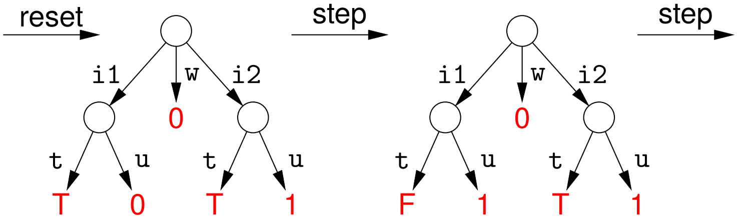
Uses an alternate denotational semantics where the solution is a tree of streams, mimicking the shape of the memory state of the OBC program.

Proof outline 3: translation to OO code

Alternate denotational semantics:



Sequence of OBC transitions:



Proof outline 4: generation of Clight code

Lots of pointers and nested structures in the generated Clight
$\Rightarrow$ need to reason about nonaliasing
$\Rightarrow$ separation logic to the rescue!

$$\{p \mapsto \_\} \; *p = v \; \{p \mapsto v\} \qquad \frac{\{P\}\; c\; \{Q\}}{\{P \star R\}\; c\; \{Q \star R\}}$$

We don't use a full separation logic, just separation logic assertions (built from $p \mapsto v$ and from $\star$ separating conjunctions) to describe the Clight memory state at each step of the Clight small-step semantics.

Pass by in-out reference, in separation logic

```
void g(int * a, int b) { *a = *a + b; }

int f(int c) { int x = 1; g(&x, c); return x; }
```

Pass by in-out reference, in separation logic

```
void g(int * a, int b) { *a = *a + b; }

int f(int c) { int x = 1; g(&x, c); return x; }
```

$$S \quad \star \quad \underbrace{(x_f \mapsto 1 \star c_f \mapsto 2)}_{\text{frame(f)}}$$

Pass by in-out reference, in separation logic

```
void g(int * a, int b) { *a = *a + b; }

int f(int c) { int x = 1; g(&x, c); return x; }
```

$$S \star \underbrace{(x_f \mapsto 1 \star c_f \mapsto 2)}_{\text{frame(f)}}$$

$$\downarrow$$

$$S \star \underbrace{(c_f \mapsto 2)}_{\text{susp-frame(f)}} \star \underbrace{(a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 1)}_{\text{frame(g)}}$$

Pass by in-out reference, in separation logic

```
void g(int * a, int b) { *a = *a + b; }

int f(int c) { int x = 1; g(&x, c); return x; }
```

$$S \star \underbrace{(x_f \mapsto 1 \star c_f \mapsto 2)}_{\text{frame(f)}}$$

$$\downarrow$$

$$S \star \underbrace{(c_f \mapsto 2)}_{\text{susp-frame(f)}} \star \underbrace{(a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 1)}_{\text{frame(g)}}$$

$$\downarrow$$

$$S \star (c_f \mapsto 2) \star (a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 3)$$

Pass by in-out reference, in separation logic

```
void g(int * a, int b) { *a = *a + b; }

int f(int c) { int x = 1; g(&x, c); return x; }
```

$$S \star \underbrace{(x_f \mapsto 1 \star c_f \mapsto 2)}_{\text{frame(f)}}$$

$$\downarrow$$

$$S \star \underbrace{(c_f \mapsto 2)}_{\text{susp-frame(f)}} \star \underbrace{(a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 1)}_{\text{frame(g)}}$$

$$\downarrow$$

$$S \star (c_f \mapsto 2) \star (a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 3)$$

$$\downarrow$$

$$S \star (x_f \mapsto 3 \star c_f \mapsto 2)$$

Outline

1. Prologue: control software and block diagrams
2. The Lustre reactive, synchronous language and its compilation
3. The Velus formally-verified Lustre compiler
4. Perspectives

What's next?

Handle the SCADE 6 extensions to Lustre
(to support mode automata)

More optimizations at the Lustre level
(e.g. node specialization on Boolean variables)

Communicate information such as "this path is unreachable"
to the C compiler (for optimization)
to the machine-code executable (for WCET analysis).

(Re-)consider formal verification at the Lustre level beyond model-checking, e.g. Astrée-style static analysis.

Does it apply to my DSL?

Some techniques here are reusable in other contexts, e.g. the use of separation logic to tame the generation of C-like code.

Prerequisite: your DSL must have fully formal semantics, preferably mechanized in Coq or Isabelle or Agda.

Watch out for DSLs that require a run-time system, e.g.

- exceptions, continuations, fibers, ...
- dynamic memory allocation: GC, refcounts
  (or: target CakeML)
- arbitrary-precision integer arithmetic
- cryptographic libraries, communication libraries, etc.

Should I verify a code generator for my DSL?

It depends. YES if

- Your DSL has a formal semantics.
- It is widely used for critical software.
- Trust in source-level verification is important to you.

NO if

- Your DSL has no other precise definition than the imperative code generated from it.
- Your DSL is a few Lisp macros or a few Haskell definitions.
- It's not used for critical software.

Take-home messages

Lustre is a neat little language.

CompCert-style compiler verification applies well to code generators for DSLs.