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
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.
“Hello, world” example: PID controller.

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

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):
Implementing a control law

Analog electronics:
Implementing a control law

In software (today’s favorite solution):

```python
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)
```
This kind of code is rarely hand-written, but rather auto-generated from block diagrams:
In the case of Scade, this diagram is a **graphical syntax** for the Lustre reactive language:

```plaintext
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

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

Block diagram

Recursive sequences

\[ i_n = i_{n-1} + e_n \cdot dt \]
\[ d_n = (e_n - e_{n-1}) / dt \]
\[ o_n = K_p e_n + K_i i_n + K_d d_n \]

Lustre code

C code

(modeling)

(discretization)

(syntax)

(semantics)

(code generation)

(hand-coding)
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

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

A node is a set of equations \( \text{var} = \text{expr} \). It defines a function between input and output streams.

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

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th></th>
<th>y</th>
<th></th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1.5</td>
<td></td>
<td>3.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>ini</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>res</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>true fby false</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>0 fby n</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>...</td>
</tr>
</tbody>
</table>
Lustre: derived temporal operators

\( a \) at the first time step and \( b \) forever after:

\[
\begin{align*}
a & \rightarrow b \overset{\text{def}}{=} \text{if (true fby false) then } a \text{ else } b
\end{align*}
\]

The value of \( a \) at the previous time step:

\[
\text{pre}(a) \overset{\text{def}}{=} \text{nil fby } a
\]

where \text{nil} is a default value of the correct type.

\[
\text{node count(ini, inc: int; res: bool)}
\]

\[
\text{returns (n: int)}
\]

\[
\text{let}
\]

\[
\text{n = if res then ini else ini \rightarrow (pre(n) + inc)}
\]

\text{tel}
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
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

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>dist</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>...</td>
</tr>
<tr>
<td>time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>...</td>
</tr>
</tbody>
</table>
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
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 

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>dist</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>...</td>
</tr>
<tr>
<td>time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>...</td>
</tr>
<tr>
<td>(dist when sec) / time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>...</td>
</tr>
<tr>
<td>(0 fby v) when not sec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>...</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>dist</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>(dist when sec) / time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>(0 fby v) when not sec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Compilation 1: normalization

Introduce a fresh variable for each \texttt{fby} expression, and lift the \texttt{fby} expression in its own equation.

\textit{Initial code:} \\
\texttt{node count(ini, inc: int; res: bool) returns (n: int)}

\begin{verbatim}
let
  n = if (true fby false) or res then ini
  else (0 fby n) + inc;
\end{verbatim}

\texttt{tel}

\textit{Normalized code:} \\
\begin{verbatim}
var t: bool; u: int;
let
  t = true fby false;
  u = 0 fby n;
  n = if t or res then ini
  else u + inc;
\end{verbatim}

\begin{verbatim}
\end{verbatim}

\texttt{tel}

Trivia: the number of \texttt{fby} expressions is exactly the amount of memory used by the node.
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 $\text{fby}$ node, as in $x = 0 \text{fby} (x + 1)$.

**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.

```plaintext
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 t2 + inc;
tel

Not scheduled
```

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

Scheduled
```
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 $\texttt{reset}$ method to initialize the instance variables at $t = 0$.
- A $\texttt{step}$ 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.
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

```java
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;
    }
}
```

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
Compilation 3: translation to OO code

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;
    }
}

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
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;
    }
}
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

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

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;
    }
}
	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
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:
  
  ```c
  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:
  ```c
  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.
Standard encoding for an OO language without dynamic dispatch:

- Instance variables and subobjects are encoded as nested structs:
  ```c
  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.
  ```c
  void count_reset(struct count * this /*inout*/);
  void count_step (struct count * this /*inout*/,
                   int ini, int step, bool res,
                   int * n /*out*/);
  ```
Standard encoding for an OO language without dynamic dispatch:

- Instance variables and subobjects are encoded as nested structs:
  ```c
  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.
  ```c
  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

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
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.
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

Lustre \(\rightarrow\) normalization \(\rightarrow\) N-Lustre \(\rightarrow\) scheduling \(\rightarrow\) SN-Lustre

---

**declarative dataflow languages**

---

**imperative languages**

---

OBC \(\downarrow\) translation

---

fusion optimization

---

Clight \(\downarrow\) generation

---

Assembly

---

CompCert compilation
Velus languages and passes

- Lustre
- N-Lustre
- SN-Lustre
- OBC
- Clight
- Assembly

- normalization
- scheduling
- translation
- fusion optimization
- generation
- CompCert compilation

- declarative dataflow languages
- imperative languages

- denotational semantics
- operational semantics
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;
let

denotational semantics: for every node there exists a solution
\( \phi : \text{var} \to \text{stream} \) of the equations.

Substitution (of \text{var} by \text{exp} if \text{var} = \text{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 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

```plaintext
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
```

```plaintext
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
⇒ need to reason about nonaliasing
⇒ separation logic to the rescue!

\[
\{p \mapsto \_\} \ast p = v \{p \mapsto v\}
\]

\[
\frac{\{P\} \ c \ \{Q\}}{\{P \ast R\} \ c \ \{Q \ast R\}}
\]

We don’t use a full separation logic, just separation logic assertions (built from \(p \mapsto v\) and from \(\ast\) 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

```c
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

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

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

\[
S \star \left( x_f \mapsto 1 \star c_f \mapsto 2 \right)
\]

\[\text{frame}(f)\]
Pass by in-out reference, in separation logic

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

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

\[
\begin{align*}
S \star \ (x_f \mapsto 1 \star c_f \mapsto 2) \\
\quad \text{frame}(f) \\
\downarrow
\end{align*}
\]

\[
\begin{align*}
S \star \ (c_f \mapsto 2) \star \ (a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 1) \\
\quad \text{susp-frame}(f) \\
\end{align*}
\]

\[
\begin{align*}
\quad \text{frame}(g)
\end{align*}
\]
void g(int * a, int b) { *a = *a + b; }

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

\[
S \star (x_f \leftrightarrow 1 \star c_f \leftrightarrow 2)
\]

\[
\text{frame}(f)
\]

\[
\downarrow
\]

\[
S \star (c_f \leftrightarrow 2) \star (a_g \leftrightarrow \&x_f \star b_g \leftrightarrow 2 \star x_f \leftrightarrow 1)
\]

\[
\text{susp-frame}(f)
\]

\[
\downarrow
\]

\[
S \star (c_f \leftrightarrow 2) \star (a_g \leftrightarrow \&x_f \star b_g \leftrightarrow 2 \star x_f \leftrightarrow 3)
\]
void g(int * a, int b) { *a = *a + b; }

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

\[ S \star (x_f \mapsto 1 \star c_f \mapsto 2) \]

\[ \downarrow \text{frame}(f) \]

\[ S \star (c_f \mapsto 2) \star (a_g \mapsto \&x_f \star b_g \mapsto 2 \star x_f \mapsto 1) \]

\[ \downarrow \text{susp-frame}(f) \]

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

\[ \downarrow \]

\[ S \star (x_f \mapsto 3 \star c_f \mapsto 2) \]
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.
Lustre is a neat little language.

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