Towards a verified Lustre compiler with modular reset

Timothy Bourke, Lélio Brun, Marc Pouzet

Towards a verified Lustre compiler with modular reset

Extended Abstract

Timothy Bourke
Inria Paris
École normale supérieure,
PSL University
timothy.bourke@inria.fr

Lélio Brun
École normale supérieure,
PSL University
Inria Paris
lelio.brun@ens.fr

Marc Pouzet
UPMC, Sorbonne Universités
École normale supérieure,
PSL University
Inria Paris
marc.pouzet@ens.fr

ABSTRACT

This paper presents ongoing work to add a modular reset construct to a verified Lustre compiler. We present a novel formal specification for the construct and sketch our plans to integrate it into the compiler and its correctness proof.

CCS CONCEPTS

• **Software and its engineering** → **Semantics**; **Formal software verification**; *Compilers*;

KEYWORDS

Synchronous Languages (Lustre), Verified Compilation

ACM Reference Format:
Timothy Bourke, Lélio Brun, and Marc Pouzet. 2018. Towards a verified Lustre compiler with modular reset: Extended Abstract. In *SCOPES '18: 21st International Workshop on Software and Compilers for Embedded Systems, May 28–30, 2018, Sankt Goar, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3207719.3207732

1 INTRODUCTION

Lustre is a programming language for embedded control and signal processing systems [4]. Synchronous languages like Lustre allow engineers to design and validate systems at the level of abstract block diagrams and to automatically generate executable code.

Compilation transforms sets of equations defining streams of values into imperative code. We are developing a formally verified Lustre compiler called Vélus [3] in the Coq [8] interactive theorem prover. It integrates the CompCert C compiler [2, 7] and formally guarantees that repeated execution of the generated assembly code reproduces the successive values of the dataflow streams.

In this paper, we present ongoing work to add a *modular reset* construct [6] to Vélus. In particular we show a novel extension of the dataflow semantics to include this imperative construct and describe the challenges remaining to generate efficient and provably correct code.

SCOPES '18, May 28–30, 2018, Sankt Goar, Germany

ACM ISBN 978-1-4503-5780-7/18/05…$15.00
https://doi.org/10.1145/3207719.3207732

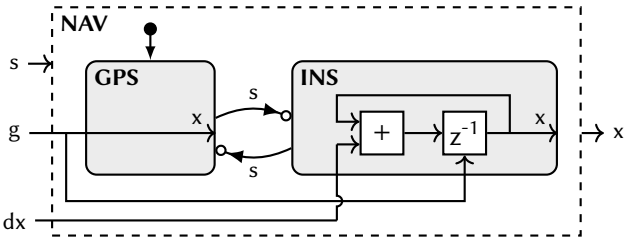

Figure 1: A graphical representation of a state machine for a simple navigation system

2 LUSTRE AND ITS VERIFIED COMPILER

The example in Figure 1 shows the logic of a simple navigation system, such as could be specified, for instance, in graphical tools like SCADE Suite[1] or Simulink.[2] The system takes three inputs: `g`, data from a GPS unit, `dx`, a local odometric estimate, and `s`, a boolean input that triggers mode changes. It produces an output `x` giving the current position. The system has two modes: `GPS` uses the external data directly and `INS` (Inertial Navigation System) is a fallback mode where the position is estimated by adding successive `dx` values to the external value at mode entry.

The state machine shown in the figure can be compiled into a purely dataflow program that uses a modular reset [5]. To show why the modular reset is necessary, we start by reprogramming the example in Lustre without it:

```
node INS(g, dx: int; rst: bool) returns (x: int);
let
  x = if (true fby false) or rst then g else (0 fby (x + dx));
tel

node NAV(g, dx: int; s: bool) returns (x: int);
  var r, c: bool;
let
  x = merge c (g when c) (INS((g, dx, r) when not c));
  c = true fby (merge c (not s when c) (s when not c));
  r = false fby (s and c);
tel
```

This program contains two nodes. A node is a function, between a list of input streams and a list of output streams, defined by a set of equations. A program associates each expression with an (infinite) stream of values. Consider, for example, the execution shown below. Variable names are given at left and their successive values are lined up alongside in columns. We fix arbitrary values for the input variables (above the line).

[1] http://www.ansys.com/products/embedded-software/ansys-scade-suite
[2] http://www.mathworks.com/products/simulink/

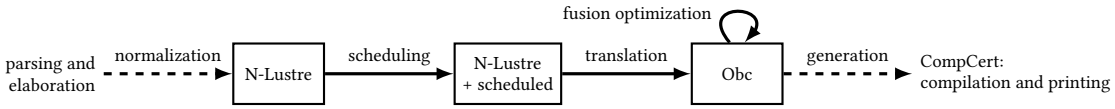

Figure 2: Compiler architecture close-up

g	6	8	2	5	8	6	7	8	10	···
dx	2	3	0	2	1	3	2	1	1	···
s	*F*	*T*	*F*	*F*	*T*	*F*	*T*	*F*	*F*	···
c	*T*	*T*	*F*	*F*	*F*	*T*	*T*	*F*	*F*	···
r	*F*	*F*	*T*	*F*	*F*	*F*	*F*	*T*	*F*	···
x_GPS	6	8				6	7			···
x_INS			2	2	4			8	9	···
x	6	8	2	2	4	6	7	8	9	···

The c variable encodes the active mode, true for GPS and false for INS. The **when** operator expresses the conditional activations implied by the state machine. In the table, the value of the expression `g when c` is labeled x_GPS and only has a value when c is true. The value of the expression `INS((g, dx, r) when not c)` is labeled x_INS and only has a value when c is false. These complementary streams are combined with the **merge** operator to give a value for x.

The local variable r is defined with the initialized delay operator **fby** ("followed by")—the z^{-1} of Digital Signal Processing. The subexpression `true fby false` defines the stream $T \cdot F \cdot F \cdot F \cdots$; for r, we have $r(0) = F$ and $\forall i > 0, r(i) = s(i-1) \wedge c(i-1)$. The initial value of c is T (GPS mode) and its next value depends on the current values of c and s. That is, (weak) transitions fire at one instant to determine the mode at the next instant; each time that s is T in the table above, the active mode alternates a column later.

The node INS implements a discrete integrator: it calculates the cumulative sum of values on dx, (re)starting from g initially and whenever rst is true. The instance of INS is thus reset using GPS data whenever r is true, that is, whenever the state machine enters the inertial mode.

In this translation of the state machine, we added an explicit reset signal to the INS interface and around the enclosed **fby** expression. In general, though, this is impractical and inefficient, as modes may contain arbitrarily many and arbitrarily nested node instantiations. The modular reset primitive solves this problem. We first describe, however, the compilation and semantics of basic Lustre programs before showing how to extend them in Section 3.

2.1 Compiler Architecture

The Vélus compiler [3] turns Lustre programs into imperative code. It implements *clock-directed modular compilation* [1] in which each node is compiled into a distinct sequential function and each equation is assigned a static clock expression that becomes a nesting of conditional statements in the generated code.

The successive source-to-source transformations of the Vélus compiler are outlined in Figure 2.

Parsing turns a source file into an abstract syntax tree with no type or clock annotations. *Elaboration* adds type and clock annotations to a program and checks that they are consistent.

Normalization rewrites a program into an abstract syntax that has two forms of expressions: *control expressions* that may contain **merge**s and **if**s (at top level) and *simple expressions* that may not. Neither form may contains **fby**s or node instantiations. Instead there are three forms of equations: those that equate a variable with a control expression, those that equate a variable with a **fby** over a simple expression, and those that equate one or more variables with a node instantiation over simple expressions.

Scheduling sorts equations by variable dependencies: variables must be written before they are read, except those defined by **fby**s which must be read before they are overwritten with a value for use in a subsequent cycle. Normalizing and scheduling the example gives the following program.

```
node INS(g, dx: int; rst: bool) returns (x: int)
  var t: bool; y: int;
let
  x = if t or rst then g else y;
  t = true fby false;
  y = 0 fby (x + dx);
tel

node NAV(g, dx: int; s: bool) returns (x: int)
  var r, c, k: bool; x_INS: int when not c;
let
  x_INS = INS((g, dx, r) when not c);
  k = merge c (not s when c) (s when not c);
  x = merge c (g when c) x_INS;
  r = false fby (s and c);
  c = true fby k;
tel
```

Translation transforms dataflow programs into an imperative intermediate language called Obc. Each equation in the original program becomes a conditionally executed assignment so that repeated execution of the imperative program generates the successive values of the streams in the dataflow program. Translation naively introduces nested **if** statements for each individual equation. A subsequent *fusion optimization* merges adjacent conditionals whenever possible to reduce branching.

Generation transforms Obc into Clight. Clight [2] is an input language of the CompCert verified C compiler, which Vélus exploits for the *compilation* to and *printing* of assembly code.

Most of the compiler passes are specified and proved correct in Coq. Coq 'extracts' them into OCaml code which can be executed. Our correctness proofs compose with those of CompCert to give the end-to-end correctness theorem presented elsewhere [3].

2.2 Dataflow Semantic Model

Defining the semantics of Lustre programs in Coq essentially means representing their executions—that is, the 'grid' shown above for the example—in formal logic. The execution of a node is encoded as an *environment* H that maps each variable to a stream of values and a *base clock* bk, a boolean stream that marks the instants when the node is active.

The semantics is centered around a pair of mutually recursive predicates, one for nodes and the other for individual equations. The formal rule for nodes is:

$$\frac{\begin{pmatrix}\text{node } f\ (\vec{\imath})\ \text{returns}\ \vec{o}\\ \text{var}\ \vec{v}\ \text{let}\ \overrightarrow{eqn}\ \text{tel}\end{pmatrix} \in G \qquad \text{same\_clock}^{\#}\ (\overrightarrow{xs}{+}{+}\overrightarrow{ys}) \qquad bk = \text{clock}^{\#}\ \overrightarrow{xs} \qquad H \vdash_{\text{var}} \vec{\imath} \Downarrow \overrightarrow{xs} \qquad H \vdash_{\text{var}} \vec{o} \Downarrow \overrightarrow{ys} \qquad G, H \vdash^{bk}_{\text{eqn}} \overrightarrow{eqn}}{G \vdash_{\text{node}} f(\overrightarrow{xs}, \overrightarrow{ys})}$$

It declares (below the line) that in a program G, a node f relates a list of input streams $\overrightarrow{xs}$ to a list of output streams $\overrightarrow{ys}$ if (above the line), (i) f is declared in G with input variables $\vec{\imath}$, output variables $\vec{o}$, local variables $\vec{v}$, and equations $\overrightarrow{eqn}$; (ii) the input and output streams are all present or absent simultaneously; (iii) a base clock bk is true only when the inputs are present; (iv) there exists an environment H that associates the input variables to the input streams; (v) it associates the output variables to the output streams, and; (vi) it also satisfies all the equations.

The predicate for individual equations has three cases, one for each of the equation forms. The rule for node instantiations is:

$$\frac{H \vdash^{bk}_{\text{e}} \vec{e} :: ck \Downarrow \overrightarrow{es} \qquad G \vdash_{\text{node}} f(\overrightarrow{es}, \overrightarrow{xs}) \qquad H \vdash_{\text{var}} \vec{x} \Downarrow \overrightarrow{xs}}{G, H \vdash^{bk}_{\text{eqn}} \vec{x} =_{ck} f(\vec{e})}$$

It declares that in a program G, an environment H satisfies a clocked equation $\vec{x} =_{ck} f(\vec{e})$, if (i) the argument expressions $\vec{e}$, with clock ck, evaluate in H to the list of streams $\overrightarrow{es}$; (ii) the node f relates the input streams $\overrightarrow{es}$ to a list of output streams $\overrightarrow{xs}$, and; (iii) H associates the variables $\vec{x}$ to these output streams.

Technicalities aside, the predicates express two important principles: externally, a node relates input streams to output streams; internally, the input and output streams are projected from an environment (the grid representing an execution) that must satisfy all the constraints imposed by the node equations.

3 THE MODULAR RESET CONSTRUCT

The modular reset was introduced [6] as a basic primitive for specifying dynamically reconfigurable systems and is used notably in the compilation of state machines [5]. A node instantiation $f(\vec{e})$ reset by a boolean expression r is written $f(\vec{e})$ every r. Using this construct, the state machine of Figure 1 can be expressed as the following dataflow program.

```
node INS(g, dx: int) returns (x: int)
let
  x = if (true fby false) then g else (0 fby (x + dx));
tel

node NAV(g, dx: int; s: bool) returns (x: int)
  var r, c: bool;
let
  x = merge c (g when c) (INS((g, dx) when not c) every r);
  c = true fby (merge c (not s when c) (s when not c));
  r = false fby (s and c);
tel
```

Compared to the previous translation, it is not necessary to pass an additional argument to `INS` nor to consider resets in the definition of `x`. The modular reset is an imperative construct: it effectively 'restarts' a dataflow node by recursively resetting all **fby**s to their initial values—and this is exactly how it is implemented.

3.1 Dataflow with Reset Semantic Model

Hamon and Pouzet [6] present a *recursive* intuition of the modular reset. If Lustre allowed recursion, an equation `y = f(x) every r` could be translated into the following program.

```
node true_until(r: bool) returns (c: bool)
let
  c = if r then false else (true fby c);
tel

node reset_f(x: int; r: bool) returns (y: int)
  var c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c)) (reset_f((x, r) when not c));
tel
```

In any instance of `reset_f`, the `c` variable is true until the next reset and then it is always false. While `c` is true, an instance of `f(x when c)` defines the values of `y`. When `c` becomes false, this instance is never reactivated and the values of `y` are defined by a fresh, recursive instantiation of `reset_f`.

Although such recursive programs are not accepted in Lustre, since compiling them for execution in bounded memory is difficult or impossible, this approach could be used to encode the modular reset semantics in Coq. But doing so engenders technicalities—like the need to mix mutually recursive inductive and coinductive predicates—that we prefer to avoid.

Another approach, derived from the original formalization [6, §4.3] and mimicking the translation presented in Section 2, is to augment the semantic predicate for nodes with a 'reset stream'. The new 'wire' is passed from node to node, combined by disjunction with local reset signals, and included in the semantics of the **fby** construct. Ideally though, the modular reset could be added without having to complicate the existing semantic definitions.

Our solution builds on key ideas from the recursive intuition.

(1) There is an instance of f for every true value in r.
(2) Each instance constrains the overall execution starting from its true value up to, but not including, the next true value.

We define an operator mask $k\ r\ xs$ that takes the value of xs from the instant of the kth true value of r to the instant just before the $(k+1)$th true value of r and is otherwise absent. The table below shows the effect of this operator—here abbreviated to xs^k_r—on the `INS` node for arbitrary values of `r`, `g`, and `dx`.

r	F	F	T	F	F	T	F	$\cdots$
g	3	5	6	9	11	15	16	$\cdots$
dx	1	2	2	1	0	3	1	$\cdots$
g^0_r	3	5						$\cdots$
dx^0_r	1	2						$\cdots$
$\text{INS}(\text{g}^0_\text{r},\text{dx}^0_\text{r})$	3	4						$\cdots$
g^1_r			6	9	11			$\cdots$
dx^1_r			2	1	0			$\cdots$
$\text{INS}(\text{g}^1_\text{r},\text{dx}^1_\text{r})$			6	8	9			$\cdots$
g^2_r						15	16	$\cdots$
dx^2_r						3	1	$\cdots$
$\text{INS}(\text{g}^2_\text{r},\text{dx}^2_\text{r})$						15	18	$\cdots$
$\vdots$								
INS(g,dx) **every** r	3	4	6	8	9	15	18	$\cdots$

The table shows a sequence of distinct instances of the INS node, with each applied to successive clippings of the input streams to give successive clippings of the overall output stream. This gives an *infinite unrolling* of the recursive intuition. Each instance constrains a different part of the output stream. Since the streams within a node are absent whenever the inputs are, any **fby** can only take its initial value after its enclosing instance becomes active.

This idea is formalized by simply using a universal quantifier to denote the unrolling of node instances:

$$\frac{\forall k,\ G \vdash_{\mathsf{node}} f(\mathsf{mask}\ k\ rk\ \overline{xs}, \mathsf{mask}\ k\ rk\ \overline{ys})}{G, rk \vdash_{\mathsf{reset}} f(\overline{xs}, \overline{ys})}$$

This predicate declares that in a program G and subject to a *reset clock* rk, a node f relates a list of input streams $\overline{xs}$ to a list of output streams $\overline{ys}$ if a sequence of node instances relates suitably clipped subsequences of the input streams to corresponding subsequences of the output streams. There is no need to explicitly 'merge' the subsequences: the relational semantics simply requires that for a given $\overline{xs}$ there exist a $\overline{ys}$ that satisfies all the constraints. The masking of inputs ensures that a node instance is 'fresh' when it becomes active. The masking of outputs ensures that an inactive node instance does not constrain $\overline{ys}$.

We can now formally state the rule for equations defined by node instantiation with reset:[3]

$$\frac{H \vdash_{\mathsf{var}} r \Downarrow rs \qquad rk = \mathsf{boolmask}^{\#}\ rs \qquad H \vdash^{bk}_{\mathsf{e}} \vec{e} :: ck \Downarrow \overline{es} \qquad G, rk \vdash_{\mathsf{reset}} f(\overline{es}, \overline{xs}) \qquad H \vdash_{\mathsf{var}} \vec{x} \Downarrow \overline{xs}}{G, H \vdash^{bk}_{\mathsf{eqn}} \vec{x} =_{ck} f(\vec{e})\ \mathsf{every}\ r}$$

It is essentially the same as the earlier rule for node instantiations without reset, except that the variable r must be associated with a stream rs and the mutual induction goes through the new predicate rather than directly through the one for nodes.

3.2 Compiling the modular reset

In clock-directed modular compilation [1], a Lustre node is translated into an Obc class with two methods: *reset* initializes the instance variables for **fby**s and *step* calculates a transition. Methods are recursively invoked for node instances. Our current prototype translates (without proof) a reset equation $\vec{x} =_{ck} f(\vec{e})\ \mathsf{every}\ r$ into a conditional call to *reset* followed directly by a call to *step*. Each call is wrapped in conditionals according to its static clock.

Consider, for example, this simple program that instantiates the INS node and a filter node, whose definition is irrelevant:

```
node main(x, dx: int; ck, r: bool) returns (y: int)
  var v, w: int when ck;
let
  v = filter(x when ck);
  w = INS(v, dx when ck) every r;
  y = merge ck w 0;
tel
```

Translation to Obc produces the following step method.

```
step(x, dx: int; ck, r: bool) returns (y: int) var v, w : int {
  if (ck) { v := filter(v).step(x) };
  if (r) { INS(w).reset() };
  if (ck) { w := INS(w).step(v, dx) };
  if (ck) { y := w } else { y := 0 } }
```

[3] $\mathsf{boolmask}^{\#}\ (\mathsf{abs} \cdot xs) = F \cdot \mathsf{boolmask}^{\#}\ xs$; $\mathsf{boolmask}^{\#}\ (F \cdot xs) = F \cdot \mathsf{boolmask}^{\#}\ xs$; $\mathsf{boolmask}^{\#}\ (T \cdot xs) = T \cdot \mathsf{boolmask}^{\#}\ xs$.

In this case, the fusion optimization will optimize the third and fourth conditionals, but the interceding reset call prevents coalescing the first and third statements. This is a shame, since the following manually optimized code calculates the same result but with less branching.

```
step(x, dx: int; ck, r: bool) returns (y: int) var v, w : int {
  if (r) { INS(w).reset() };
  if (ck) {
    v := filter(v).step(x);
    w := INS(w).step(v, dx);
    y := w
  } else { y := 0 }
}
```

The problem is that scheduling occurs before method calls are introduced, and to respect data dependencies the equation for w must come between those for v and y. Translating equations one-by-one facilitates the correctness proof, but inevitably places the reset and step calls together. Scheduling equations is easy to justify as the dataflow semantics is independent of their order; justifying reorderings of sequential programs requires more effort. Compiling hierarchical state machines produces Lustre programs with potentially many clocks and modular resets and excessive branching gives longer execution times and pessimistic worst-case estimates. Our future work thus aims to more effectively optimize this case.

4 SUMMARY AND FUTURE WORK

We have presented a novel formalization in Coq of the semantics of the modular reset construct. Our future work will focus on developing an intermediate dataflow language that exposes the details of node instances and methods. This will allow more precise scheduling and make the fusion optimization more effective. We are also working to complete the correctness proof for the compilation of the modular reset.

REFERENCES

[1] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 9th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. ACM Press, Tucson, AZ, USA, 121–130.

[2] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *Proceedings of the 14th International Symposium on Formal Methods (FM 2006) (Lecture Notes in Computer Science)*, Vol. 4085. Springer, Hamilton, Canada, 460–475.

[3] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM Press, Barcelona, Spain, 586–601.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 1987)*. ACM Press, Munich, Germany, 178–188.

[5] Jean-Louis Colaço, Bruce Pagano, and Marc Pouzet. 2005. A Conservative Extension of Synchronous Data-flow with State Machines. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT 2005)*, Wayne Wolf (Ed.). ACM Press, Jersey City, USA, 173–182.

[6] Gégoire Hamon and Marc Pouzet. 2000. Modular Resetting of Synchronous Data-Flow Programs. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, Frank Pfenning (Ed.). ACM Press, Montreal, Canada, 289–300.

[7] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[8] The Coq Development Team. 2016. *The Coq proof assistant reference manual.* Inria. Version 8.5.