Towards a verified Lustre compiler with modular reset
Timothy Bourke, Lélio Brun, Marc Pouzet

To cite this version:
Timothy Bourke, Lélio Brun, Marc Pouzet. Towards a verified Lustre compiler with modular reset. 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES 2018), May 2018, Sankt Goar, Germany. pp.4, 10.1145/3207719.3207732. hal-01817949

HAL Id: hal-01817949
https://inria.hal.science/hal-01817949
Submitted on 18 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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

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.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

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 Suite1 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).

1http://www.ansys.com/products/embedded-software/ansys-scade-suite
2http://www.mathworks.com/products/simulink/
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 $\mathcal{z}^{-1}$ of Digital Signal Processing. The subexpression true fby false defines the stream $T \cdot F \cdot F \cdot \cdots$; for r, we have $r(\emptyset) = F$ and $\forall i > 0, r(i) = s(i-1) \land c(i-1)$. 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 c 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 merges 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 eate a variable with a control expression, those that eate a variable with a fby over a simple expression, and those that eate 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);
end
```

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 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.
Towards a verified Lustre compiler with modular reset

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.

```plaintext
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 \( \text{reset}_f \), the \( c \) variable is true until the next reset and then it is always false. While \( c \) is true, an instance of \( f(x \text{ when } c) \) 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 \( \text{reset}_f \).

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 \( \text{fby} \) 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 \( \text{mask} k r x \) that takes the value of \( x \) from the instant of the \( k \)th 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 \( x_k r \)—on the INS node for arbitrary values of \( r, g, \) and \( dx \).

<table>
<thead>
<tr>
<th>( r )</th>
<th>( F )</th>
<th>( F )</th>
<th>( F )</th>
<th>( F )</th>
<th>( F )</th>
<th>( F )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g )</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>( dx )</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>( \text{INS}^{0}_{dx} )</td>
<td>( 3 )</td>
<td>4</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{INS}^{1}_{dx} )</td>
<td>( 6 )</td>
<td>9</td>
<td>11</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{INS}^{2}_{dx} )</td>
<td>( 2 )</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{INS}^{3}_{dx} )</td>
<td>( 6 )</td>
<td>8</td>
<td>9</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compared to the previous translation, it is not necessary to pass an additional argument to \( \text{INS} \) 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 \( \text{fby} \)s to their initial values—and this is exactly how it is implemented.
The table shows a sequence of distinct instances of the INS node, each with 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:

\[ \forall k. \ G \_\text{} \text{node} \ f (\text{mask } k \ r k \bar{x} \bar{s}, \text{mask } k \ r k \bar{y} \bar{s}) \]

This predicate declares that in a program \(G\) and subject to a reset clock \(r k\), a node \(f\) relates a list of input streams \(\bar{x} \bar{s}\) to a list of output streams \(\bar{y} \bar{s}\) 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 \(\bar{x} \bar{s}\) there exist a \(\bar{y} \bar{s}\) 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 \(\bar{y} \bar{s}\).

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

\[
\begin{align*}
H \_\ast \text{var} \ r \parallel r s & \quad r k = \text{boolmask}^k r s \\
H \_\ast \text{eqn} \ e : \text{ck} \parallel \bar{x} & \quad G, r k \_\text{reset} f (\bar{x}, \bar{s}) \quad H \_\ast \text{var} \ x \parallel \bar{x} \\
G, H \_\ast \text{eqn} \ x = \text{ck} \quad f (e) \text{ every } r 
\end{align*}
\]

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 \(\bar{x} = \text{ck} \ f (\bar{e})\) 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:

```plaintext
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 \emptyset;
  tel
```

Translation to Obc obeys the following step method.

```plaintext
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(v).step(v, dx); }
  if (ck) { y := w } else { y := 0 }
}
```

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.

```plaintext
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(v).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


