Open access • Proceedings Article • DOI:10.1145/3062341.3062358

A formally verified compiler for Lustre — Source link

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy ...+2 more authors

Institutions: French Institute for Research in Computer Science and Automation, Collège de France

Published on: 14 Jun 2017 - Programming Language Design and Implementation

Topics: Lustre (programming language), Imperative programming, Compiler, Dataflow and Assembly language

A Formally Verified Compiler for Lustre

Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, Lionel Rieg

A Formally Verified Compiler for Lustre

Timothy Bourke[1,2]
Timothy.Bourke@inria.fr

Lélio Brun[2,1]
Lelio.Brun@ens.fr

Pierre-Évariste Dagand[3,4,1]
Pierre-Evariste.Dagand@lip6.fr

Xavier Leroy[1]
Xavier.Leroy@inria.fr

Marc Pouzet[3,2,1]
Marc.Pouzet@ens.fr

Lionel Rieg[5,6]
Lionel.Rieg@yale.edu

1. Inria, Paris, France
2. Département d'Informatique, École normale supérieure, PSL Research University, Paris, France
3. Sorbonne Universités, UPMC Univ Paris 06, France
4. CNRS, LIP6 UMR 7606, Paris, France
5. Collège de France, Paris, France
6. Yale University, New Haven, Connecticut, USA

Abstract

The correct compilation of block diagram languages like Lustre, Scade, and a discrete subset of Simulink is important since they are used to program critical embedded control software. We describe the specification and verification in an Interactive Theorem Prover of a compilation chain that treats the key aspects of Lustre: sampling, nodes, and delays. Building on CompCert, we show that repeated execution of the generated assembly code faithfully implements the dataflow semantics of source programs.

We resolve two key technical challenges. The first is the change from a synchronous dataflow semantics, where programs manipulate streams of values, to an imperative one, where computations manipulate memory sequentially. The second is the verified compilation of an imperative language with encapsulated state to C code where the state is realized by nested records. We also treat a standard control optimization that eliminates unnecessary conditional statements.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.4 [*Programming languages*]: Processors—Compilers; D.2.4 [*Software engineering*]: Software/Program Verification—Correctness proofs, Formal methods

1. Introduction

Lustre was introduced in 1987 as a programming language for embedded control and signal processing systems [13]. It gave rise to the industrial tool SCADE Suite[1] and can serve as a target to compile a subset of Simulink/Stateflow[2] to executable code [15, 61]. SCADE Suite is used to develop safety-critical applications like fly-by-wire controllers and power plant monitoring software. Several properties make Lustre-like languages suitable for such tasks: constructs for programming reactive controllers, execution in statically-bounded time and memory, a mathematically well-defined semantics based on dataflow streams [13], traceable and modular compilation schemes [8], and the practicability of automatic program verification [17, 25, 30, 38] and industrial certification. These languages allow engineers to develop and validate systems at the level of abstract block diagrams that are compiled directly to executable code.

Compilation transforms sets of equations that define streams of values into sequences of imperative instructions that manipulate the memory of a machine. Repeatedly executing the instructions is supposed to generate the successive values of the original streams: but how is this to be ensured? Existing approaches apply translation validation [49, 52, 60] or industrial certification based on development standards and coverage criteria [50]. We take a different approach by formally specifying the source, target, and intermediate languages, their corresponding models of computation, and the compilation algorithms in the Coq Interactive Theorem Prover (ITP) [63]. We state and prove a correctness relation between the source and target semantic models, and build on the CompCert project [10, 40] to integrate the semantics

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

of machine-level types and operators and to extend the correctness relation to the level of assembly code for PowerPC, ARM and x86 processors.

In this paper, we make the following contributions:

- The Vélus compiler that turns synchronous dataflow equations into assembly instructions and a proof of its correctness in the Coq ITP (Section 2). Our compiler supports the key features of the Lustre [13] language. Initial experimental results are encouraging (Section 5).
- We identify the invariants, lemmas, and proof structures necessary to verify code generation, providing a blueprint for performing similar work in any ITP. Specifically, we (1) introduce a novel semantic model combining the infinite sequences of dataflow models with the incremental manipulation of memories that characterizes imperative models (Section 3.2), and (2) exploit separating assertions to concisely and modularly relate an idealized memory model to the lower-level model of CompCert's Clight (Section 4).
- We follow the architecture of the industrial SCADE Suite compiler, including modular code generation (one sequential function per dataflow node) and an optimization to fuse conditionals (Section 3.3). The verification of the optimization uses properties of the source language and translation function and shows the utility of our formalization.
- Our compiler front-end is implemented and verified parametrically with respect to an abstract interface of operators (Section 4.1). This approach not only facilitates proof, it also allows our tool chain to be instantiated with different backends and their semantic models.

The outcome of our work is twofold. First, our Coq development extracts to OCaml, providing us with an executable compiler. Second, we have mechanically verified that for any node f in a Lustre program G accepted by our compiler, for any well-typed input stream xs and corresponding output stream ys, the semantics of the generated assembly code is an infinite execution trace that is bisimilar to the trace that reads the stream of input values from, and writes the expected output values to, volatile variables:

$$\frac{\begin{array}{c} G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, \overrightarrow{ys}) \\ \mathsf{compile}\, G\, f = \mathsf{OK}\, asm \end{array}}{\exists T, asm \Downarrow T \wedge \langle \mathsf{VLoad}(xs_{(n)}).\mathsf{VStore}(ys_{(n)}) \rangle_{n=0}^{\infty} \sim T}$$

This proof is obtained by composing CompCert's original correctness result, from Clight to assembly, and the novel correctness result from Lustre to Clight that we present in this paper. As in standard programming practice, we combine and extend existing systems to produce new functionality. By doing so in an ITP, we guarantee that the different parts function together correctly.

2. Lustre and its Compilation

A Lustre program comprises a list of *nodes*. A node is a named function between lists of input and output streams, defined by a set of equations. Consider this example adapted from the original paper [13]:

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

This node counter takes three input streams—two of integers and one of booleans—and returns a stream of integers. It calculates the cumulative sum of values on inc taking the value of ini initially or when res is true. In graphical editors, like those of SCADE Suite or Simulink, this node would be represented by a rectangle labeled with the node name and having three input ports and one output port.

The value of the output n is defined by a single equation that freely mixes instantaneous operators (or and +), a multiplexer (**if/then/else**: both branches are active and the value of the guard selects one of the results), and the initialized delay operator '**fby**' ("followed by"). The initialized delay corresponds to the flip-flop of sequential logic and the z^{-1} of Digital Signal Processing. The subexpression true **fby** false defines the stream $T, F, F, F, \ldots$, and the subexpression 0 **fby** n defines a stream c such that $c(0) = 0$ and $\forall i > 0, c(i) = n(i-1)$.

We describe the other features of Lustre, namely node instantiations and the **when** and **merge** sampling constructs, after outlining the compiler architecture.

2.1 Compiler Architecture and Implementation

In the classical approach to compiling Lustre [13, 28], all nodes are inlined and a control automaton is constructed to minimize the code executed in a given state. In the *clock-directed modular* approach [8], each dataflow node is compiled to a distinct sequential function and each equation is assigned a static clock expression that becomes a conditional in the imperative code.[3] This approach is used in the industrially certified SCADE Suite compiler and academic compilers [23, 35]. It is the one addressed in our work.

Figure 1 outlines the successive source-to-source transformations of a clock-directed modular Lustre compiler.

The first four stages deal with dataflow programs. *Parsing* turns a source file into an Abstract Syntax Tree (AST) without type or clock annotations. *Elaboration* rejects programs that are not well-typed and well-clocked, and otherwise yields an (annotated) Lustre program that can be assigned a semantics. *Normalization* ensures that every **fby** expression and node instantiation occurs in a dedicated equation and not nested arbitrarily within an expression. *Scheduling* sorts the dataflow equations by variable dependencies in anticipation of their translation one by one into imperative assignments: variables must be written before they are read, except those defined

[3] Various trade-offs of inlining and modularity are possible [43, 55].

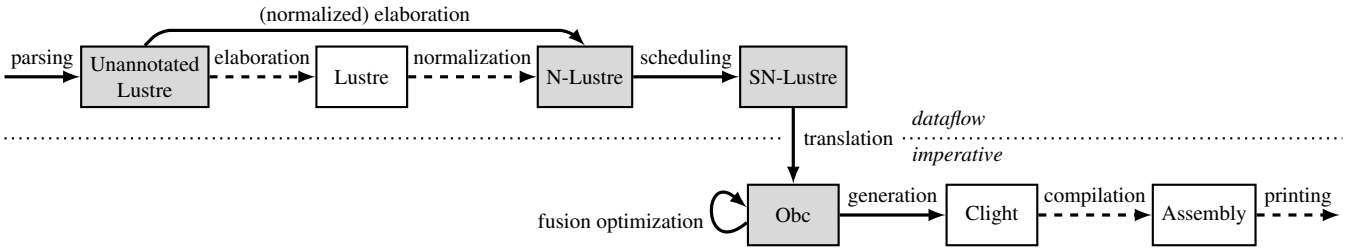

Figure 1. Clock-directed modular compiler architecture: the gray boxes and the solid arrows are addressed in this paper. The white boxes and the dashed arrows are addressed in previous work [2, 3] or by CompCert [40].

e :=	**expression**
x	(variable)
c	(constant)
$\diamond\, e$	(unary operator)
$e \oplus e$	(binary operator)
e when x	(sampling on T)
e whenot x	(sampling on F)

ce :=	**control expression**
e	(expression)
merge ck ce ce	(merge)
if e then ce else ce	(mux)

ck :=	**clock**
base	(base clock)
ck on x	(subclock on T)
ck onot x	(subclock on F)

eqn :=	**equation**
$x =_{ck} ce$	(definition)
$x =_{ck} c$ fby e	(delay)
$\overrightarrow{x} =_{ck} f(\overrightarrow{e})$	(node call)

d :=	**node declaration**
node f $(\overrightarrow{x^{ty}})$ returns $\overrightarrow{x^{ty}}$	
var $\overrightarrow{x^{ty}}$ let $\overrightarrow{eqn}$ tel	

Figure 2. SN-Lustre : abstract syntax.

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

```
node d_integrator(gamma: int)
returns (speed, position: int)
let
  speed = counter(0, gamma, false);
  position = counter(0, speed, false);
tel
```

```
node tracker(acc, limit: int) returns (p, t: int)
  var s, pt : int; x : bool; c : int when x;
let
  (s, p) = d_integrator(acc);
  x = rising(s > limit);
  c = counter(0 when x, 1 when x, false when x);
  t = merge x c (pt whenot x);
  pt = 0 fby t;
tel
```

Figure 3. SN-Lustre: example program.

by **fby**s which must be read before they are written with their next value.

Normalization and scheduling change the syntactic form of a program but preserve its dataflow semantics. Normalization is allowed because Lustre is referentially transparent: a variable can always be replaced by its defining expression and conversely. Scheduling is allowed because the meaning of a set of equations is independent of their order. The normalized and scheduled example appears at left in Figure 3.

Both normalization and scheduling were verified in Coq in prior work [2, 3]. We do not dwell on them here but rather focus on the problems that were not previously addressed. We elaborate directly to N-Lustre and apply a formally validated scheduling algorithm to produce SN-Lustre.

Returning to Figure 1, the first of these problems is the *translation* from SN-Lustre to the intermediate imperative language Obc [8, §4]. This pass changes both syntax and semantics: from dataflow equations defining functions on streams to sequences of imperative assignments defining transitions between memory states. Examples of Obc programs are given in Section 3, where we detail the translation pass along with the statement and proof of its correctness.

The translation pass produces a nesting of conditional statements for each equation in the source program. This facilitates compiler specification and proof but gives inefficient code. A *fusion optimization* is thus normally applied to merge adjacent conditionals. Its specification and verification are detailed in Section 3.3.

The second significant problem addressed in our work is the *generation* of compilable code, typically C, Java, or Ada, from Obc. In our work we generate Clight, a subset of C accepted by the CompCert compiler, whose *compilation* to assembly code has been modeled and verified in Coq [10, 40]. This requires incorporating a subset of Clight's types and operators into SN-Lustre programs. We exploit the module system of Coq to abstract over the details until the generation pass where they are instantiated with Clight-specific definitions. The generation pass makes only minor alterations to the control structure of a source Obc program, namely to support multiple return values. More significantly, it changes

the representation of program memories. The state of Obc programs is a tree of variable environments. It must be encoded as nested records in the target Clight program, and the concomitant details of alignment, padding, and aliasing must be confronted. The generation pass is described in Section 4.

Our prototype compiler implements the languages in the gray boxes and the transformations marked by solid arrows in Figure 1. The lexer is generated by ocamllex [42, §12]. The parser is generated in Coq by Menhir [53] which also produces proofs of correctness and completeness [36]. The elaborator is a mix of OCaml, mostly from CompCert for parsing constants, and Coq, it rejects programs that are not normalized. We prove using standard techniques that successful elaboration yields a well-typed and well-clocked N-Lustre program. Scheduling is implemented in OCaml and validated in Coq. The translation and generation algorithms are implemented as Coq functions and extracted to OCaml. We generate a Clight AST that CompCert's formally verified or validated algorithms compile into assembly code.

2.2 SN-Lustre, Node instantiation, and Sampling

The abstract syntax of SN-Lustre is given in Figure 2. Many of the elements have already been presented. We now describe node instantiation and the constructs for sampling and merging streams via the extended example of Figure 3.

The counter node is the normalized and scheduled version of the earlier example. The d_integrator node [28, II(C)] instantiates counter twice to calculate speed and position values from a stream of accelerometer readings. It is itself instantiated in the tracker node that returns the position and a count of the number of times that the speed exceeds a given limit. The instants of overshoot are detected as rising edges (rising(in) = ((false **fby** not in) and in) on the expression s > limit. Consider the values of the streams in this node for a given acc and with limit a constant stream of 5s:

acc	0	2	4	-2	0	3	-3	2	···
limit	5	5	5	5	5	5	5	5	···
s	0	2	6	4	4	7	4	6	···
(s_c)	0	0	2	6	4	4	7	4	···
p	0	2	8	12	16	23	27	33	···
(p_c)	0	0	2	8	12	16	23	27	···
x	F	F	T	F	F	T	F	T	···
c			1			2		3	···
(c_c)			0			1		2	···
t	0	0	1	1	1	2	2	3	···
pt	0	0	0	1	1	1	2	2	···

The values of s and p are ultimately defined by counter instances with a value at every instant; the variables defined within the instances by c = 0 **fby** n are denoted (s_c) and (p_c).

The **when** construct samples a stream and controls the activation of program elements. The effect of sampling the constant streams in the definition of c is to activate the counter node at a slower rate than its context. In contrast, counter(0, 1, false) **when** x activates counter at every instant and samples the result. The value of c is the stream $1, 2, 3, \ldots$; its synchronization with the other streams, giving the gaps in the table, is captured by the clock base on x, which is declared in the type/clock annotation ': int **when** x' in Figure 3. Similarly, the expression pt **whenot** x samples pt when x is false. Its clock is base onot x.

The **merge** construct combines complementary streams according to the clock given as the first argument. Together with the **fby** defining pt, its effect here is to sustain the value of t between excess speed events. A set of clock typing rules [19] allows clocks to be inferred by the compiler and ensures that well-clocked programs can be executed synchronously, that is, without additional buffering. These details are not central to our approach; it is only important to note that every equation is associated with a clock having the hierarchical structure shown in Figure 2.

The normalization pass ensures that **merge**s and **if**s only occur at the top level of expressions that do not contain **fby**s or node instantiations. The structural invariants given by normalization are captured in the abstract syntax by the three specific forms of equations, and the distinction between *control expressions* and *expressions*.

The simple example of Figure 3 was chosen for reasons of space and clarity. But just as the concision of the λ-calculus belies its expressiveness and utility, so too is the simplicity of Lustre misleading. It and the underlying formalism of difference equations forms the core of the Scade 6 language which is used to program complex and critical discrete controllers. It also suffices to model distributed embedded controllers [5, 14, 27], and to encode automata [20, 44, 45], temporal logic [31], regular expressions [58], and numerical simulations [6, 7]. We aim to provide a solid base to manipulate and reason about such applications in an ITP.

Differences with 'classical' Lustre. In addition to initialized registers (**fby**s) [64], Lustre [13] provides distinct initialization (`->`) and delay operators (`pre`). Initialization is readily expressed in SN-Lustre: x -> y becomes **if** (true **fby** false) **then** x **else** y. While `pre` is not fundamentally more expressive than **fby** [13, §3.2], its use sometimes gives more pleasing programs but requires a static analysis to check that it is used correctly (initialization analysis).

The **merge** operator [54] supersedes the `current` operator, whose value before the arrival of its input is undefined. It allows syntactic clock typing and is readily generalized for the compilation of case structures and automata [20]. Scade 6 uses **merge** and provides **fby**. Simulink provides counterparts for **when** ('conditionally-executed subsystems' [46, p.4-26]) and **merge** ('Merge' [47]).

For simplicity, we require all inputs and outputs of a node application to have the same clock; in Lustre the inputs and outputs may be on sub-clocks of the clock of the first input.

e :=	**expression**	s :=	**statement**	cls :=	**class declaration**
x	(local variable)	$x := e$	(update)	class $c\,\{$	
$\mathsf{state}(x)$	(state variable)	$\mathsf{state}(x) := e$	(state update)	memory $\overrightarrow{x^{ty}}$	
c	(constant)	if e then s else s	(conditional)	instances $\overrightarrow{i^{c}}$	
$\diamond\, e$	(unary operator)	$\overrightarrow{x} := c_i.m(\overrightarrow{e})$	(method call)	$(\overrightarrow{x^{ty}})\ m(\overrightarrow{x^{ty}}) =$ var $\overrightarrow{x^{ty}}$ in s ⋯	
$e \oplus e$	(binary operator)	$s; s$	(composition)	$\}$	
		skip	(do nothing)		

Figure 4. Obc: abstract syntax.

2.3 Intermediate Code: Translation and Generation

In the clock-directed modular approach, compilers typically generate imperative code in an object-based intermediate language. The idea is to encapsulate the memory required to implement a dataflow node with the code generated to manipulate it. The object style is a natural way of describing and reasoning about the translation, and the use of an intermediate language facilitates the code generation for different targets by separating generic transformations from language-specific details [8, §4].

It turns out that this approach is also advantageous for formal verification because it separates the task of reasoning about the translation between the dataflow and imperative models from that of reasoning about the packing of bytes into machine memory. The invariants needed for each task differ in purpose and form. Translating directly from SN-Lustre to Clight would eliminate the need to define and reason about the syntax, semantics, and properties of the intermediate language, but this advantage is greatly outweighed by the inevitable increase in complexity of the generation function and associated invariants and proofs.

It also turns out to be more convenient to define and verify control structure optimizations in the intermediate language.

We adopt the intermediate language Obc [8, §4], which resembles the SOL language used for the same purpose in the SCADE Suite compiler. It is a fairly conventional imperative language with a simple means of encapsulating state. The abstract syntax is presented in Figure 4. There are two noteworthy features. First, a distinction is made between variables x and memories $\mathsf{state}(x)$ in both expressions (reads) and update statements (writes). Second, a program is a list of classes and each class comprises lists of typed memories, named instances of previously declared classes, and named methods. Each method m declares—from left to right in Figure 4—lists of output, input, and local variables, and is defined by a statement s that may access and update those variables and the memories. A method call specifies a class c, a declared instance i, and a method m: a list of expressions gives its inputs and a list of distinct variable names is nominated to store the returned values.

The basic principle of the translation pass (Figure 1) is to turn each dataflow **node** into an imperative **class** where a memory is introduced for each variable defined by a **fby** and local variables are used for other equations and node inputs, and an instance is introduced for each node call. Two methods are defined for each node: *reset* to initialize memories and instances, and *step* to calculate the next single instant, that is, a 'column' of the semantic table. Calling *reset* initially and then repeatedly calling step calculates the successive values of the dataflow streams. We give examples of Obc programs, and describe the translation pass and its correctness proof in the next section.

The generation of Clight from Obc involves introducing a record type for each **class** to store memories and instances, producing a function for each class/method pair, encoding multiple return values using records and pointers, and dealing with various technicalities of the detailed machine model exposed to C programs. We describe this pass and its correctness proof in Section 4.

We obtain the full compilation chain, from N-Lustre to assembly, by composing our compiler with CompCert and, similarly, we prove the end-to-end correctness theorem by composing the respective correctness lemmas.

3. Translation to Obc

The translation pass is specified as a functional program in Coq. It maps a list of SN-Lustre nodes into a list of Obc classes. Most of the translate function is shown in Figure 5. The trnode function is not shown for want of space. It calculates a set mems of variables defined by **fby**s, uses it to partition declarations of node variables into declarations of class memories and the local variables of the *step* method, and defines the bodies of the *step* and *reset* methods using treqss and treqsr, respectively. We show that if the source program is well-typed then so is the resulting program.

Consider now the other auxiliary functions. The function var maps a dataflow variable into a local or state variable based on its membership in mems. Expressions are otherwise translated by trexp, which propagates constants and operators, and removes whens. The cases for e whenot x and $\diamond\, e$ are not shown being easily deducible. Unlike trexp which maps an SN-Lustre expression into an Obc one, trcexp maps a variable and expression into an update statement. Only the case for merge is shown, the case for if/then/else is identical.

The ctrl function embodies the principle that *clocks in the source language are transformed into control structures in the target language* [8], by following the structure of a clock ck to nest a statement s in conditional tests. It is applied for

$$\mathsf{var}\,x \triangleq \begin{cases} \mathit{state}(x) & \text{if } x \in \mathsf{mems} \\ x & \text{if } x \notin \mathsf{mems} \end{cases}$$

$$\begin{aligned}
\mathsf{trexp}\quad c &\triangleq c \\
\mathsf{trexp}\quad x &\triangleq \mathsf{var}\,x \\
\mathsf{trexp}\,(e\ \mathit{when}\ x) &\triangleq \mathsf{trexp}\,e \\
\mathsf{trexp}\ (e_1 \oplus e_2) &\triangleq (\mathsf{trexp}\,e_1) \oplus (\mathsf{trexp}\,e_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{trcexp}\,x\,(\mathit{merge}\,y\,e_t\,e_f) &\triangleq \mathit{if}\,(\mathsf{var}\,y)\ \mathit{then}\,(\mathsf{trcexp}\,x\,e_t) \\
&\qquad\qquad \mathit{else}\ (\mathsf{trcexp}\,x\,e_f) \\
\mathsf{trcexp}\,x\quad e &\triangleq x := \mathsf{trexp}\,e
\end{aligned}$$

$$\begin{aligned}
\mathsf{ctrl}\quad \mathit{base}\quad s &\triangleq s \\
\mathsf{ctrl}\ \ (ck\ \mathit{on}\ x)\ \ s &\triangleq \mathsf{ctrl}\ ck\ (\mathit{if}\,(\mathsf{var}\,x)\ \mathit{then}\ s\ \mathit{else\ skip}) \\
\mathsf{ctrl}\,(ck\ \mathit{onot}\ x)\,s &\triangleq \mathsf{ctrl}\ ck\ (\mathit{if}\,(\mathsf{var}\,x)\ \mathit{then\ skip\ else}\ s)
\end{aligned}$$

$$\begin{aligned}
\mathsf{treqs}\quad (x =_{ck} ce) &\triangleq \mathsf{ctrl}\ ck\ (\mathsf{trcexp}\,x\ ce) \\
\mathsf{treqs}\quad (x =_{ck} c\ \mathit{fby}\ e) &\triangleq \mathsf{ctrl}\ ck\ (\mathit{state}(x) := \mathsf{trexp}\,e) \\
\mathsf{treqs}\,(x : xs =_{ck} f(es)) &\triangleq \\
&\qquad \mathsf{ctrl}\ ck\ (x : xs := f_x.\mathit{step}(\mathsf{map\ trexp}\ es))
\end{aligned}$$

$$\mathsf{treqss}\ eqns \triangleq \mathsf{foldl}\ (\lambda\ acc\ eqn.\,\mathsf{treqs}\ eqn; acc)\ eqns\ \mathit{skip}$$

$$\begin{aligned}
\mathsf{treqr}\,acc\quad (x =_{ck} ce) &\triangleq acc \\
\mathsf{treqr}\,acc\quad (x =_{ck} c\ \mathit{fby}\ e) &\triangleq \mathit{state}(x) := c; acc \\
\mathsf{treqr}\,acc\,(x : xs =_{ck} f(es)) &\triangleq f_x.\mathit{reset}(); acc
\end{aligned}$$

$$\mathsf{treqsr}\ eqns \triangleq \mathsf{foldl\ treqr}\ eqns\ \mathit{skip}$$

$$\mathsf{translate}\,nodes \triangleq \mathsf{map\ trnode}\ nodes$$

Figure 5. Core of the SN-Lustre to Obc Translation.

each of the three cases in treqs: definitions become updates, delays become state updates, and node calls become calls to the (previously generated) *step* method for the associated node. The last case requires a way to identify this particular instance: we choose to use the left-most return variable, which is guaranteed to be unique within the node.

The *step* method is generated by folding treqs over the list of equations.[4] Similarly, *reset* is generated with treqr mapping delays into state updates, and node calls into invocations of the *reset* method for the associated instances.

The *step* and *reset* statements for the example tracker node are shown below at left and right:

```
s, p := d_integrator_s.step(acc);          d_integrator_s.reset();
x := rising_x.step(s > limit);             rising_x.reset();
if x then c := counter_c.step(0, 1, false) counter_c.reset();
    else skip;                             state(pt) := 0;
if x then t := c else t := state(pt);      skip
state(pt) := t;
skip
```

While the translation function is succinct, the formal justification of its correctness is not! We must define semantic models for SN-Lustre and Obc, and show a theorem describing how the translate function preserves the observable behavior of source programs. We also find it necessary to introduce a third semantic model to facilitate the proof. We outline the models and proof before describing the optimization of translated *step* methods.

[4] treqss expects equation lists in the reverse order to that used in Figure 3.

$$(c\,\mathsf{hold}^{\#}\,xs)_{(0)} = c$$

$$\frac{xs_{(n)} = \langle c' \rangle}{(c\,\mathsf{hold}^{\#}\,xs)_{(n+1)} = c'}$$

$$\frac{xs_{(n)} = \mathsf{abs}}{(c\,\mathsf{hold}^{\#}\,xs)_{(n+1)} = (c\,\mathsf{hold}^{\#}\,xs)_{(n)}}$$

$$\frac{xs_{(n)} = \mathsf{abs}}{(c\,\mathsf{fby}^{\#}\,xs)_{(n)} = \mathsf{abs}}$$

$$\frac{xs_{(n)} = \langle c' \rangle}{(c\,\mathsf{fby}^{\#}\,xs)_{(n)} = \langle (c\,\mathsf{hold}^{\#}\,xs)_{(n)} \rangle}$$

Figure 6. Definition of the $\mathsf{fby}^{\#}$ operator.

3.1 Semantic Models

Dataflow. The semantic model of SN-Lustre is a specification for the compilation chain. It must correspond with existing formal but unmechanized definitions of the language [13, §3.2][19, §3] and permit effective reasoning within an ITP.

The first important choice is the representation of streams. Earlier work [2, 3] used lists, but their extrapolation to infinite objects is not simple and they engender uninteresting proof obligations—for instance, for the empty list or to argue that two streams are of equal length. Coinductive streams are a more natural choice but their use in ITPs can involve nontrivial technicalities [18, §5]. We thus model streams as functions from natural numbers to a value domain. The nth value of a stream s is denoted by $s_{(n)}$. This choice worked well; time was treated by induction on the index argument (n) and our proofs encountered the expected obligations in the expected forms, without tedious additional technicalities.

The next choice is how to handle the sampling operators; that is, how to model the gaps in the table shown earlier. In a Kahn semantics [37, 51], these gaps are simply omitted, but in a synchronous semantics they express the timing of calculations relative to the underlying iterations of the system. We model them by explicitly encoding presence ($\langle c \rangle$) and absence (abs) in the value domain. On one hand, this complicates the definition of fby, see Figure 6, which can no longer be encoded by simply prefixing the initial value to the argument stream. Rather the argument stream must 'slide to the right' over the absent gaps which remain fixed in place; this is what the auxiliary hold operator encodes. On the other hand, it allows the explicit manipulation of presence and absence in invariants and proofs.

The dataflow semantics is standard [13] and defined in two parts: a *combinational* fragment over values at an instant and a *sequential* fragment over streams. The combinational judgments are relative to a pair (R, b) of an *instantaneous environment* R mapping variables to present or absent values, and a boolean b that indicates whether the enclosing node is active in the instant. There are judgments for variables ($R \vdash^{b}_{\mathsf{var}} x \Downarrow v$), clocks ($R \vdash^{b}_{\mathsf{ck}} ck \Downarrow b'$), expressions ($R \vdash^{b}_{\mathsf{e}} e \Downarrow v$), clocked expressions ($R \vdash^{b}_{\mathsf{e}} e :: ck \Downarrow v$), control expressions ($R \vdash^{b}_{\mathsf{ce}} ce \Downarrow v$), and clocked control expressions ($R \vdash^{b}_{\mathsf{ce}} ce :: ck \Downarrow v$).

The sequential judgments are defined relative to a *history environment* H that maps each variable to a stream of values, and a *base clock* bk that is a stream of boolean values representing the base clock of the enclosing node. History environments essentially encode the kind of semantic table shown earlier, each variable name is mapped to a row giving its values over time. Since nodes can be conditionally activated, the semantics of equations is specified with respect to a base clock that indicates when the enclosing node is active.

A mutually inductive definition simultaneously defines the semantics of equations and nodes. The semantic judgment for delay equations integrates the $\mathsf{fby}^{\#}$ of Figure 6:

$$\frac{H \vdash^{bk}_{\mathsf{e}} e :: ck \Downarrow ls \qquad H \vdash_{\mathsf{var}} x \Downarrow xs \qquad xs = c\,\mathsf{fby}^{\#}\, ls}{G, H \vdash^{bk}_{\mathsf{eqn}} x =_{ck} c\ \mathsf{fby}\, e}$$

That is, an equation $x =_{ck} c\ \mathsf{fby}\, e$ has a semantics in a program G, a history environment H, and for a base clock bk, if the semantics of e is a stream ls and the value of x in H is a stream xs that is a delayed version of ls with initial value c.

The semantics of a node f in a program G relates a stream of input values $\overrightarrow{xs}$ to a stream of output values $\overrightarrow{ys}$. The semantic judgment requires the existence of a history environment whose projections on the input and output variables correspond with $\overrightarrow{xs}$ and $\overrightarrow{ys}$, respectively, and which is satisfied by the equations in the node $\overrightarrow{eqn}$:

$$\frac{\begin{pmatrix}\mathsf{node}\ f\ (\overrightarrow{i^{t_i}})\ \mathsf{returns}\ \overrightarrow{o^{t_o}} \\ \mathsf{var}\ \overrightarrow{v^{t_v}}\ \ \mathsf{let}\ \overrightarrow{eqn}\ \ \mathsf{tel}\end{pmatrix} \in G \qquad \begin{array}{l} bk = \mathsf{clock}^{\#}\,(\mathsf{hd}\,\overrightarrow{xs}) \\ G, H \vdash^{bk}_{\mathsf{eqn}} \overrightarrow{eqn} \\ H \vdash_{\mathsf{var}} \overrightarrow{i} \Downarrow \overrightarrow{xs} \\ H \vdash_{\mathsf{var}} \overrightarrow{o} \Downarrow \overrightarrow{ys} \end{array} \qquad \forall n, \overrightarrow{xs}_{(n)} = \mathsf{abs} \Leftrightarrow \overrightarrow{ys}_{(n)} = \mathsf{abs}}{G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, \overrightarrow{ys})}$$

This judgment embodies two key properties. First, the streams of an instantiated node are only activated when the inputs $\overrightarrow{xs}$ are present since the base clock bk is derived from those inputs: $(\mathsf{clock}^{\#}\, x)_{(n)} \triangleq$ if $x_{(n)} = \mathsf{abs}$ then F else T. Second, we require that the clocks of the inputs $\overrightarrow{xs}$ and the outputs $\overrightarrow{ys}$ are synchronized.

Imperative. Obc is essentially a conventional imperative language and the formalization of its semantics in an ITP is routine. The only idiosyncrasy is in the treatment of instance memories. Expressions and statements read and update, respectively, pairs of memory environments. A *local memory* (env) models a stack frame, (partially) mapping variable names to values. A *global memory* (mem) models a static memory that contains two (partial) mappings, variable names to values and instance names to sub-memories. Its type is parameterized by a domain of values V and defined as a recursive record with two fields:

$$\mathsf{memory}\, V \triangleq \left\{ \begin{array}{ll} \mathsf{values} & : \mathsf{ident} \rightharpoonup V \\ \mathsf{instances} & : \mathsf{ident} \rightharpoonup \mathsf{memory}\, V \end{array} \right\}$$

The memory of a program compiled from SN-Lustre reflects the tree of nodes in the source code: there is an entry in values for each **fby** and one in instances for each node call. The subscript in the step and reset calls associates an instance of a node with its sub-memory.

We define a big-step semantics for Obc. The semantic judgment for a statement s in the context of a program $prog$ relates initial and updated memory pairs. For example, the rule for updates is:

$$\frac{mem, env \vdash_{\mathsf{e}} e \Downarrow v}{mem, env \vdash_{\mathsf{st}} x := e \Downarrow mem, env \cup \{x \mapsto v\}}$$

That is, after executing $x := e$ the global memory is unchanged and the x is updated in the local memory with the value v given by the expression semantics. The rule for state updates is the same but for updating $mem.\mathsf{values}$. A method call evaluates the argument expressions in the initial memory mem, looks up the class name in $prog$, and executes its step statement relative to a smaller $prog'$, a global (sub-)memory retrieved from $mem.\mathsf{instances}$, and a local memory that associates inputs to their values. The instance memory and result variables are then updated.

Executing an Obc program reads inputs from, and calculate results in, a top-level environment env, and updates internal memories mem for subsequent re-executions.

3.2 Correctness

The correctness of the translation pass is simple to state: if a node f of a dataflow program G maps the streams of inputs $\overrightarrow{xs}$ to the streams of outputs $\overrightarrow{ys}$, then, in $\mathsf{translate}\, G$, first executing the associated *reset* method before repeatedly executing the associated *step* method with the successive values of $\overrightarrow{xs}$ generates the successive values of $\overrightarrow{ys}$.

The formal statement of this property requires predicates expressing that the equations within each node of G are correctly scheduled and that nodes are not applied circularly. These properties are assured by elaboration and scheduling.

The resulting statement is too weak to prove directly. The semantic judgment $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, \overrightarrow{ys})$ only describes the input/output behavior of a node. The values of internal streams, like $\mathsf{c_c}$, $\mathsf{s_c}$ and $\mathsf{p_c}$ in the example, are hidden and, unlike Obc memories, these streams may not necessarily have a value at each instant. These facts prevent the formulation and proof of the invariant needed to show correctness. We solve this problem by introducing a new semantic model that combines aspects from the dataflow and imperative models.

$$\frac{}{\mathrm{MemCorresEqn}_n(M, mem, x =_{ck} e)}$$

$$\frac{(M.\mathsf{values}(x))_{(n)} = mem.\mathsf{values}(x)}{\mathrm{MemCorresEqn}_n(M, mem, x =_{ck} v_0 \ \mathsf{fby}\, e)}$$

$$\frac{\mathrm{MemCorres}^f_n(M.\mathsf{instances}(x), mem.\mathsf{instances}(x))}{\mathrm{MemCorresEqn}_n(M, mem, x : xs =_{ck} f(\overrightarrow{e}))}$$

Figure 7. SN-Lustre/Obc memory correspondence.

An intermediate model with exposed memory. The new semantic judgment has the form $G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, \overrightarrow{ys})$. It exposes a memory tree M whose structure is isomorphic to the tree of instances in the dataflow and translated imperative programs. Unlike in the Obc semantics, the domain V for memory is not instantiated by single constant values but rather by streams of constant values. For each variable x defined by a **fby** in the source program at any level of the call hierarchy, it specifies the sequence of values that should be taken over successive iterations by the corresponding $\mathsf{state}(x)$ in the translated code.

In defining the new model, we reuse the instantaneous semantic rules for expressions. The rules for nodes, ordinary equations, and node calls are redefined almost trivially. The new rule for **fby**s is the only really interesting one:

$$\frac{\begin{array}{cc} ms = M.\mathsf{values}(x) & H \vdash^{bk}_{\mathsf{e}} e :: ck \Downarrow ls \\ ms_{(0)} = v_0 & H \vdash_{\mathsf{var}} x \Downarrow sx \end{array} \quad \forall n, \begin{cases} ms_{(n+1)} = ms_{(n)} \wedge sx_{(n)} = \mathsf{abs} & \text{if } ls_{(n)} = \mathsf{abs} \\ ms_{(n+1)} = v \wedge sx_{(n)} = \langle ms_{(n)} \rangle & \text{if } ls_{(n)} = \langle v \rangle \end{cases}}{G, M, H \vdash^{bk}_{\mathsf{meqn}} x =_{ck} v_0 \ \mathsf{fby}\, e}$$

The stream ms corresponding to the memory for the variable x is retrieved from $M.\mathsf{values}$. Initially, it takes the value v_0. At later instants, its value is maintained if the argument stream is absent; otherwise its next value is the value present on that stream. The stream sx associated with x is absent when the argument is and otherwise present with the current value of the memory. The behavior of this model is intentionally very close to that of the translated code.

If a node has a semantics, we show that it also has a semantics with exposed memories.[5] In practice, this permits us to reason about memory values in the proofs of properties stated solely in terms of the standard dataflow semantics.

With a node memory M exposed, it becomes possible to relate it directly to a global memory mem at an instant n. This relationship is expressed in the predicate $\mathrm{MemCorres}^f_n(M, mem)$ where G is implicit. The predicate requires that each equation for the node f in G satisfy $\mathrm{MemCorresEqn}$, Figure 7, which holds trivially for basic equations, and is otherwise defined directly for delays and recursively for node applications.

[5] Formally, $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, \overrightarrow{ys})$ implies $\exists M, G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, \overrightarrow{ys})$.

Proof of translation. The correctness of the nth execution of a translated step method is stated for a well-scheduled program G in terms of the semantics that exposes memories.

LEMMA 1. *Given $G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, \overrightarrow{ys})$ for a node f in G, a global memory mem where $\mathrm{MemCorres}^f_n(M, mem)$ at an instant n, then there is a global memory mem' such that*

$$mem \vdash_{\mathsf{step}} r := f_r.step(\overrightarrow{xs}_{(n)}) \Downarrow mem', \overrightarrow{ys}_{(n)}$$
$$\textit{and} \ \ \mathrm{MemCorres}^f_{n+1}(M, mem'),$$

where we execute f's step statement in $\mathsf{translate}\, G$ using the semantics of Obc method calls.

This lemma is shown by three nested inductions: over instants n, node definitions in G, and equations $eqns$ within a node; and two case distinctions: on the three classes of equations, and whether the associated clock is true or not. Together with a lemma showing that $f_r.reset()$ establishes $\mathrm{MemCorres}^f_0(M, mem_0)$, it implies the correctness result.

The formal proof is not trivial. The most subtle case is for node applications whose clock is false: since the associated imperative function is not executed we cannot appeal to the induction hypothesis on G and must reason that the instance memory does not change in this case.

Lemma 1 involves showing that an imperative step has a semantics. While Obc programs cannot diverge by construction, we must ensure that they do not 'get stuck'. We rely on scheduling to guarantee that local variable reads succeed, on $\mathrm{MemCorres}$ to guarantee that state variable reads succeed, and on the existence of the dataflow semantics to show that operators are always defined. The last point is subtle but important: it entails an obligation to verify that source programs apply operators correctly.

3.3 Fusion Optimization

Translation generates code with too many guards, so an optimization pass is normally used to fuse adjacent conditionals.[6] For example, the *step* code for tracker becomes:

```
s, p := d_integrator_s.step(acc);
x := rising_x.step(s > limit);
if x then (c := counter_c.step(0, 1, false); t:= c)
    else t := state(pt);
state(pt) := t
```

This optimization is effective as scheduling places similarly clocked equations together. We define it over two functions. The first simply divides sequential compositions in two:

$$\begin{array}{lcl} \mathsf{fuse}\,(s_1; s_2) & \triangleq & \mathsf{zip}\ s_1\ s_2 \\ \mathsf{fuse} \quad s & \triangleq & s \end{array}$$

The zip function is shown in Figure 8.[7] It iteratively integrates statements from its second argument into the first and recursively performs the optimization.

[6] Incorporating this optimization into the translation pass would complicate both compilation and proof. In particular, the induction along the list of equations would need to track and manipulate the 'open' if/else constructs.

[7] The Coq version is broken into three pieces to make termination manifest.

$$
\begin{array}{llll}
\mathsf{zip} & (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) & (\mathsf{if}\ e\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & \\
 & & \triangleq & \mathsf{if}\ e\ \mathsf{then}\ \mathsf{zip}\ s_1\ t_1\ \mathsf{else}\ \mathsf{zip}\ s_2\ t_2 \\
\mathsf{zip} & (s_1; s_2) & t & \triangleq\ s_1; (\mathsf{zip}\ s_2\ t) \\
\mathsf{zip} & s & (t_1; t_2) & \triangleq\ \mathsf{zip}\ (\mathsf{zip}\ s\ t_1)\ t_2 \\
\mathsf{zip} & s & \mathsf{skip} & \triangleq\ s \\
\mathsf{zip} & \mathsf{skip} & t & \triangleq\ t \\
\mathsf{zip} & s & t & \triangleq\ s; t
\end{array}
$$

Figure 8. Obc optimization: loop fusion.

While the first rule of zip does not preserve the semantics of $s_1; s_2$ in general,[8] it does for the code produced by the translate function. To prove this, we must characterize an invariant that assures soundness, show that it holds of code produced by translate, and also that it is preserved by the successive transformations of fuse. We formalize a predicate Fusible over statements whose only non-trivial rule is

$$
\frac{\mathrm{Fusible}(s_1) \qquad \mathrm{Fusible}(s_2) \qquad \forall x \in \mathrm{Free}(e), \neg\mathrm{MayWrite}(x, s_1) \wedge \neg\mathrm{MayWrite}(x, s_2)}{\mathrm{Fusible}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)}
$$

where $\mathrm{MayWrite}(x, s)$ is true iff s contains an assignment to x or $\mathsf{state}(x)$. The justification that this predicate holds requires a subtle technical argument about well-formed clocks in SN-Lustre programs.

We prove that fuse preserves Fusible by showing that it is a congruence for the relation $s_1 \approx_{\mathrm{fuse}} s_2$ that relates fusible statements that transform memory states identically.

As the treatment of fusion demonstrates, proving the correctness of optimizations may require reasoning across the whole framework, from the semantics of dataflow clocks, across the details of the translation function, and finally over equivalence relations on the imperative code.

4. Generation of Clight

The generation pass takes an Obc program, for instance, one produced by translation from SN-Lustre, and generates a Clight program for compilation by CompCert.

Generation treats the Obc classes in a program one by one. A record is declared for each with fields for every memory and instance. Another record is declared for each method with fields for the output variables. This is necessary since Obc allows multiple return values but Clight does not. We optimize the special cases of zero or one output. The records generated for the example tracker class are:

```
struct tracker {
  int pt;
  struct d_integrator s;
  struct rising x;
  struct counter c;
};

struct tracker$step {
  int p;
  int t;
};
```

[8] Consider (**if** x **then** x:= false **else** x := true); **if** x **then** ⋯ **else** ⋯.

```
void tracker$step(struct tracker *self,
                  struct tracker$step *out,
                  int acc, int limit)
{
  struct d_integrator$step out$s$step;
  register int step$n, s, c;
  register bool step$edge, x;

  d_integrator$step(&(*self).s, &out$s$step, acc);
  s = out$s$step.speed;
  (*out).p = out$s$step.position;

  step$edge = rising$step(&(*self).x, s > limit);
  x = step$edge;

  if (x) { step$n = counter$step(&(*self).c, 1, 1, 0);
           c = step$n; (*out).t = c; }
  else { (*out).t = (*self).pt; }

  (*self).pt = (*out).t;
}
```

Figure 9. Clight *step* method for the tracker node.

A separate function is generated for each class/method pair. Figure 9 shows the result for tracker/step. Two additional parameters precede the method inputs: self is a pointer to an instance memory and out is a pointer to memory for returning output values. Within the function, auxiliary variables are introduced for every invoked class/method to retrieve outputs, and for every local and output variable declared in the source method. Method calls are translated into calls to previously generated functions with two new argument values: an instance memory retrieved from self and, when necessary, an output record. A sequence of assignments is added after each call to copy from the auxiliary variable into local and output variables. These new assignments aside, the control structure of the source statement is reproduced exactly in the generated code. In expressions, variables are either translated into member accesses within out or self, or directly as registers, and constants and operators are propagated directly.

We do not detail Clight's abstract syntax or the generation function. The former is described elsewhere [9, Figures 1, 2, and 3]. The latter is similar to the translate function of Section 3, although its definitions are more intricate since they must treat the low-level details inherent to Clight.

There are both big-step and small-step operational semantics for Clight. We reason about the correctness of generation in the big-step model since Obc methods always terminate and since their control structure is essentially preserved in the resulting code. We need not invoke CompCert's framework for simulation proofs; we simply proceed by induction on arbitrary Obc statements.

The big-step judgment for a Clight statement is written:

$$
ge, e \vdash_{\mathsf{stmt}} le, m, s \overset{t}{\Rightarrow} le', m', oc
$$

$\mathsf{val}, \mathsf{type}, \mathsf{const}, \mathsf{unop}, \mathsf{binop} : \mathsf{Type}$

$\mathsf{bool} : \mathsf{type} \quad \mathsf{true}, \mathsf{false} : \mathsf{val} \quad \vdash_{\mathsf{wt}} : \mathsf{val} \to \mathsf{type} \to \mathsf{Prop}$

$\mathsf{ty_c} \quad : \mathsf{const} \to \mathsf{type}$

$\mathsf{sem_c} \quad : \mathsf{const} \to \mathsf{val}$

$\mathsf{ty_{uop}} \quad : \mathsf{unop} \to \mathsf{type} \rightharpoonup \mathsf{type}$

$\mathsf{sem_{uop}} : \mathsf{unop} \to \mathsf{val} \to \mathsf{type} \rightharpoonup \mathsf{val}$

$\mathsf{ty_{bop}} \quad : \mathsf{binop} \to \mathsf{type} \to \mathsf{type} \rightharpoonup \mathsf{type}$

$\mathsf{sem_{bop}} : \mathsf{binop} \to \mathsf{val} \to \mathsf{type} \to \mathsf{val} \to \mathsf{type} \rightharpoonup \mathsf{val}$

Figure 10. Operator interface: values.

A statement s is executed in a global environment ge that tracks global variable, function, and type declarations, and a local environment e that maps (local) variable names to their addresses and types. It may read and alter values in a temporaries environment le and a memory m to produce updated versions le' and m', a trace of observable events t, and an outcome oc. Memories map addresses—that pair abstract block identifiers with integer offsets—to byte-level values and abstract permissions [41][1, §32]. In contrast to a local environment, a temporaries environment maps variable names to values directly. The address-of operator (&) is only allowed for variables in e but reasoning about variables in le is easier because there is no indirection or aliasing.

We generate code that places the output blocks in e, since we must pass their addresses in function calls, and the other variables in le—denoted by the **register** keyword in Figure 9. The semantic models can be instantiated to store function parameters in e or in le; we choose the latter.

Reasoning about the generation pass presents two main technical challenges: integrating types and operators from Clight into Obc and SN-Lustre (Section 4.1), and reasoning in the memory model to relate the tree-like memorys of Obc to nested **struct**s in the generated code and to handle multiple return values (Section 4.2). The solutions are applied to obtain the correctness result (Section 4.3).

4.1 Abstracting and Implementing Operators

The definitions and proofs about SN-Lustre, Obc, and the translation pass are defined relative to an *operator interface*. Technically, we define the operator interface as a module type and the other components as functors over that type.

Figure 10 presents most of the interface. There are types for values (val), value types (type), constants (const), unary operators (unop), and binary operators (binop). We require that type contain an element bool and that val contain two elements true and false. There is a typing judgment ($\vdash_{\mathsf{wt}}$) and (partial) functions that map constants and operators to their types and values.

The bool type and values are distinguished because they are required to define the semantics of sampling, merges, muxs, and clocks in SN-Lustre, and of conditionals in Obc.

Figure 10 omits the properties required of interface elements, namely $\mathsf{true} \neq \mathsf{false}$, $\vdash_{\mathsf{wt}} \mathsf{true} : \mathsf{bool}$, $\vdash_{\mathsf{wt}} \mathsf{false} : \mathsf{bool}$, $\vdash_{\mathsf{wt}} \mathsf{sem_c}\ c : \mathsf{ty_c}\ c$, and two type preservation properties. The type preservation property for unary operators ($\diamond$) is:

$$\frac{(\mathsf{ty_{uop}} \diamond ty) = ty' \quad \vdash_{\mathsf{wt}} v : ty \quad (\mathsf{sem_{uop}} \diamond v\ ty) = v'}{\vdash_{\mathsf{wt}} v' : ty'}$$

The property for binary operators ($\oplus$) is similar.

Our compiler from SN-Lustre to Obc can thus be instantiated to any suitable language or for different variations of a given language. Here we instantiate it with CompCert's values and semantics so as to compose both compilers.

Instantiating the interface. The operator interface must be instantiated with parts of CompCert's front-end [9, 10] for the implementation and verification of the generation pass. We instantiate val with the type of CompCert values, and type with a subset of Clight types comprising integer, boolean, and floating-point types (but not pointers, arrays or **struct**s). We reuse CompCert's dynamic semantics and type system almost directly. We do, however, impose stricter typing rules than those of C concerning Boolean values (to ensure that the only values of type bool are the integers 0 and 1) and implicit casts in assignments (we require explicit casts when assigning an expression to a variable of a different type). These stricter typing rules, applied at the level of the Obc and SN-Lustre type systems, simplify the proof of semantic preservation from Obc to Clight. They ensure correspondence between updates in Obc that store values directly and assignments in Clight that involve implicit casts.

4.2 Relating Memories

The correctness of the generation pass hinges on a lemma that shows the invariance of the assertion

$$m \models \mathsf{match\_states}\ c\ f\ mem\ env\ e\ le\ (b_{self}, sofs)\ b_{out}$$

that, for a given class c and method f, relates the Obc memories mem and env to the Clight environments e and le, and the state record at address $(b_{self}, sofs)$ and output record at address $(b_{out}, 0)$ in the Clight memory m.

In essence, this predicate describes —at a logical level— the layout of the Clight memory, following the idealized memory of the Obc program. By ensuring this invariant, our compiler is guaranteed to generate memory-safe code. We explain its most salient parts in the remainder of this section.

Separation assertions. Expressing and reasoning about the internal components of match_states involves treating the contents of m with all the concomitant details of alignment, padding, and aliasing. For example, when an Obc method updates a state element, state(pt) := t, we must show for the generated code, (*self).pt = (*out).t, that the address self + offsetof(**struct** tracker, pt) is properly aligned for the field type, that there is enough space to write a value of that type, and that other fields, or indeed, all other memory locations, are unchanged.

$$\mathsf{staterep}\ []\ f\ mem\ (b, ofs) \triangleq \mathsf{sepfalse}$$

$$\mathsf{staterep}\ (\mathsf{class}\ g\ \{\overrightarrow{m^{t_m}}\ \overrightarrow{i^c}\ \cdots\} : p)\ f\ mem\ (b, ofs) \triangleq \mathsf{staterep}\ p\ f\ mem\ (b, ofs) \qquad \text{if } g \neq f$$

$$\begin{aligned}&\mathsf{staterep}\ (\mathsf{class}\ f\ \{\overrightarrow{m^{t_m}}\ \overrightarrow{i^c}\ \cdots\} : p)\ f\ mem\ (b, ofs) \triangleq \\ &\quad \mathsf{sepall}\ \big(\lambda x^{ty}.\ \mathsf{contains}\ ty\ (b, ofs + \mathsf{field\_offset}(x, \{\overrightarrow{m^{t_m}} \cdot \overrightarrow{i^c}\}))\ \lceil mem.\mathsf{values}\rceil_x\big)\ \overrightarrow{m^{t_m}} \\ &\quad * \mathsf{sepall}\ \big(\lambda x^{c_x}.\ \mathsf{staterep}\ p\ c_x\ mem.\mathsf{instances}(x)\ (b, ofs + \mathsf{field\_offset}(x, \{\overrightarrow{m^{t_m}} \cdot \overrightarrow{i^c}\}))\big)\ \overrightarrow{i^c}\end{aligned}$$

Figure 11. The staterep function yields a memory assertion, for a class named f, comparing the Clight memory in block b at offset ofs against the contents of an Obc global memory mem. Note that $\lceil f\rceil_x \triangleq \big(\lambda v.\ f\ x \text{ is undefined} \vee f\ x = v\big)$.

Separation logic [33, 59] is designed to solve such problems and we apply it in our proof. More precisely, we exploit the basic concepts and standard connectives to express and reason about assertions on memory, but we do not use it as a variant of Hoare logic. We considered using the Verified Software Toolchain (VST) [1] in our formalization, but it focuses on proofs of particular programs whereas we prove properties about a function that produces programs. The VST also includes many sophisticated features that we do not need. Instead, we extend a small library of separation assertions that was already developed within CompCert for reasoning about stack frames.

A memory assertion is a dependent record with two main elements: mfoot, a declared memory footprint (represented as a predicate over block/offset pairs), and mpred, a predicate over memories; and two technical obligations to ensure that mpred only refers to memory elements of mfoot. Separating conjunction ($*$) is defined, for instance, as:

$$p * q \triangleq \left\{\begin{array}{l}\mathsf{mfoot} = \lambda b\ ofs.\ \mathsf{mfoot}\ p\ b\ ofs \vee \mathsf{mfoot}\ q\ b\ ofs \\ \mathsf{mpred} = \lambda m.\ \mathsf{mpred}\ p\ m \wedge \mathsf{mpred}\ q\ m \\ \qquad\qquad \wedge \mathsf{disjoint}\ (\mathsf{mfoot}\ p)\ (\mathsf{mfoot}\ q)\end{array}\right\}$$

The notation $m \models p$ is just another way to write $\mathsf{mpred}\ p\ m$. We define the standard assertions and connectives, including separating implication ($-\!\!*$) similarly. This differs from mechanizations that follow the standard approach [33, 59] of first constructing a separation algebra over 'heaplets' [1, 39]. It works well in practice for assertions that only use quantifiers in simple ways.

State representation. The trickiest part of match_states is an assertion that represents the layout of the nested state record: $m \models \mathsf{staterep}\ p\ f\ mem\ (b, ofs)$, relating a list of Obc classes p, the name of a class f, an Obc global memory mem, and a piece of Clight memory in m starting at address (b, ofs). The hierarchical Obc run-time state provides the blueprint for the nested Clight run-time state.

The assertion is defined as a Coq function by the three clauses shown in Figure 11. The function returns sepfalse for the empty program and skips over classes that are not named *f*. The last clause is the interesting one. When the class named f is found, the memory is separated into two disjoint regions: one for the memory fields $\overrightarrow{m^{t_m}}$ and one for the instance fields $\overrightarrow{i^c}$. Each region is further divided by iterating a predicate over the fields using the sepall combinator:

$$\mathsf{sepall}\ p\ xs \triangleq \mathsf{foldl}\ \big(\lambda ps\ x.\ ps * p\ x\big)\ xs\ \mathsf{emp}$$

The predicate for memories uses $\mathsf{contains}\ ty\ (b, ofs)\ spec$ to assert that *spec* holds over the memory in block b in the range $[ofs, ofs + \mathbf{sizeof}(ty))$.[9] Clight's field_offset gives the offset of a field in the generated record. The memory range must contain the value $mem.\mathsf{values}(x)$ if it is defined and is unconstrained otherwise. In either case, the predicate asserts that the memory exists, that it can be read and written, and that it is disjoint from that of other fields (and any other assertions at higher levels). The predicate for instances applies staterep recursively for each instance/class pair.

The Clight memory model requires that the ownership of local memory, like that allocated for the output records, be explicitly surrendered when a function returns. Since the sepall assertions only retain access to the field memory and not to inter-field padding, match_states includes separating implications that allow field access to be exchanged to recover ownership of the original block of memory.

We obtain the correctness result by showing that each generated operation preserves match_states.

4.3 Proof of Generation

The core of the correctness proof shows invariance of match_states between an arbitrary Obc program and the Clight program generated from it. The proof proceeds by induction on the former; it is long and contains many technicalities. The semantics of operators is the same in both languages and implicit casts are justified using the type system sketched in Section 4.1. The separation assertions relate local and state variables to their implementations in Clight, we reason about them using standard techniques (associativity and commutativity of $*$, and rules for loading and storing). The instance predicates in staterep 'detach' for access to the induction hypothesis for method calls, which includes an arbitrary separation assertion ($invariant * P$) that serves as an ersatz frame rule.

A main function is generated to call the principal *reset* method initially, and the principal *step* method repeatedly.

[9] Similarly to the standard $e \mapsto e'$ assertion [33, 59].

	Vélus	*Hept+CC*	*Hept+gcc*	*Hept+gcci*	*Lus6+CC*	*Lus6+gcc*	*Lus6+gcci*
avgvelocity	315	385 (22%)	265 (-15%)	70 (-77%)	1 150 (265%)	625 (98%)	350 (11%)
count	55	55 (0%)	25 (-54%)	25 (-54%)	300 (445%)	160 (190%)	50 (-9%)
tracker	680	790 (16%)	530 (-22%)	500 (-26%)	2 610 (283%)	1 515 (122%)	735 (8%)
pip_ex	4 415	4 065 (-7%)	2 565 (-41%)	2 040 (-53%)	10 845 (145%)	6 245 (41%)	2 905 (-34%)
mp_longitudinal [16]	5 525	6 465 (17%)	3 465 (-37%)	2 835 (-48%)	11 675 (111%)	6 785 (22%)	3 135 (-43%)
cruise [54]	1 760	1 875 (6%)	1 230 (-30%)	1 230 (-30%)	5 855 (232%)	3 595 (104%)	1 965 (11%)
risingedgeretrigger [19]	285	300 (5%)	190 (-33%)	190 (-33%)	1 440 (405%)	820 (187%)	335 (17%)
chrono [20]	410	425 (3%)	305 (-25%)	305 (-25%)	2 490 (507%)	1 500 (265%)	670 (63%)
watchdog3 [26]	610	575 (-5%)	355 (-41%)	310 (-49%)	2 015 (230%)	1 135 (86%)	530 (-13%)
functionalchain [17]	11 550	13 535 (17%)	8 545 (-26%)	7 525 (-34%)	23 085 (99%)	14 280 (23%)	8 240 (-28%)
landing_gear [11]	9 660	8 475 (-12%)	5 880 (-39%)	5 810 (-39%)	25 470 (163%)	15 055 (55%)	8 025 (-16%)
minus [57]	890	900 (1%)	580 (-34%)	580 (-34%)	2 825 (217%)	1 620 (82%)	800 (-10%)
prodcell [32]	1 020	990 (-2%)	620 (-39%)	410 (-59%)	3 615 (254%)	2 050 (100%)	1 070 (4%)
ums_verif [57]	2 590	2 285 (-11%)	1 380 (-46%)	920 (-64%)	11 725 (352%)	6 730 (159%)	3 420 (32%)

Figure 12. WCET estimates in cycles [4] for step functions compiled for an armv7-a/vfpv3-d16 target with CompCert 2.6 (CC) and GCC 4.4.8 `-O1` without inlining (gcc) and with inlining (gcci). Percentages indicate the difference relative to the first column.

It performs loads and stores of volatile variables to model, respectively, input consumption and output production. The coinductive predicate presented in Section 1 is introduced to relate the trace of these events to input and output streams.

Finally, we exploit an existing CompCert lemma to transfer our results from the big-step model to the small-step one, from whence they can be extended to the generated assembly code to give the property stated at the beginning of the paper. The transfer lemma requires showing that a program does not diverge. This is possible because the body of the main loop always produces observable events.

5. Experimental Results

Our prototype compiler, Vélus, generates code for the platforms supported by CompCert (PowerPC, ARM, and x86). The code can be executed in a 'test mode' that `scanf`s inputs and `printf`s outputs using an alternative (unverified) entry point. The *verified* integration of generated code into a complete system where it would be triggered by interrupts and interact with hardware is the subject of ongoing work.

As there is no standard benchmark suite for Lustre, we adapted examples from the literature and the Lustre v4 distribution [57]. The resulting test suite comprises 14 programs, totaling about 160 nodes and 960 equations. We compared the code generated by Vélus with that produced by the Heptagon 1.03 [23] and Lustre v6 [35, 57] academic compilers. For the example with the deepest nesting of clocks (3 levels), both Heptagon and our prototype found the same optimal schedule. Otherwise, we follow the approach of [23, §6.2] and estimate the Worst-Case Execution Time (WCET) of the generated code using the open-source OTAWA v5 framework [4] with the 'trivial' script and default parameters.[10] For the targeted domain, an over-approximation to the WCET is usually more valuable than raw performance numbers. We compiled with CompCert 2.6 and GCC 4.8.4 (`-O1`) for the `arm-none-eabi` target (armv7-a) with a hardware floating-point unit (vfpv3-d16).

The results of our experiments are presented in Figure 12. The first column shows the worst-case estimates in cycles for the step functions produced by Vélus. These estimates compare favorably with those for generation with either Heptagon or Lustre v6 and then compilation with CompCert. Both Heptagon and Lustre (automatically) re-normalize the code to have one operator per equation, which can be costly for nested conditional statements, whereas our prototype simply maintains the (manually) normalized form. This re-normalization is unsurprising: both compilers must treat a richer input language, including arrays and automata, and both expect the generated code to be post-optimized by a C compiler. Compiling the generated code with GCC but still without any inlining greatly reduces the estimated WCETs, and the Heptagon code then outperforms the Vélus code. GCC applies 'if-conversions' to exploit predicated ARM instructions which avoids branching and thereby improves WCET estimates. The estimated WCETs for the Lustre v6 generated code only become competitive when inlining is enabled because Lustre v6 implements operators, like **pre** and `->`, using separate functions. CompCert can perform inlining, but the default heuristic has not yet been adapted for this particular case. We note also that we use the modular compilation scheme of Lustre v6, while the code generator also provides more aggressive schemes like clock enumeration and automaton minimization [29, 56].

Finally, we tested our prototype on a large industrial application (≈6 000 nodes, ≈162 000 equations, ≈12 MB source file without comments). The source code was already normalized since it was generated with a graphical interface,

[10] This configuration is quite pessimistic but suffices for the present analysis.

but some other modifications were required. Most notably, we had to remove constant lookup tables and replace calls to special operators implemented in C/assembler with empty nodes. Modeling and verifying these features is ongoing work. Our prototype compiles the application to assembly code in ≈1 min 40 s demonstrating that the performance of the extracted compiler is adequate for real-world code.

These preliminary experiments show the practicability of our approach in terms of compilation time and the efficiency of the generated code.

6. Related Work

Related work falls into two interrelated categories: one focuses on modeling the semantics of languages inside an ITP and the other on the correctness of compilation. We limit our survey to work that focuses on the particularities of synchronous languages, as we do, but proving the correctness of general-purpose compilers is undeniably a related problem.

Several synchronous languages have been formalized in ITPs, including a subset of Lustre in Coq using coinductive types [21], an Esterel-like language in HOL with a focus on program proof [62], a shallow embedding of Lucid Synchrone in Coq with a focus on its higher-order features and clock calculus [12], and a denotational semantics of Kahn networks in Coq [51]. Similarly, work on synchronous compilers in ITPs has remained close to the dataflow model: an unpublished report [24] for a Scade 3 compiler focused on semantic and clocking definitions; the Gene-Auto project showed the correctness of equation scheduling for a Simulink to C code generator [34]. None of this work treats the translation of synchronous dataflow programs to imperative code, nor the generation of executables.

Translation validation is an alternative to compiler verification. It was first applied to synchronous languages two decades ago [52], more recently to a subset of Simulink and its optimizing RTW compiler [60], and in ongoing work on an existing Signal compiler [48, 49]. It is attractive because it decouples compilation from proof—even if the former must usually be adapted to provide hints for the latter—which can be an important practical advantage. While translation validation gives strong formal guarantees if the validator is verified [40], this is not the case for the work cited here.

One motivation for verifying a Lustre compiler is to ensure that properties verified on models also hold on generated code. An alternative is to also compile the properties and to reverify them at the code level [22]. This is an interesting approach, but it has two disadvantages: the compilation of properties and the re-verification must be trusted; verification may succeed on the model but fail on the code.

7. Concluding Remarks

We present a formalization and proof of correctness in Coq for a compiler that builds on CompCert to transform (normalized) Lustre into assembly code. The proof establishes that the dataflow semantics of source programs is correctly implemented by the generated code.

The definitions of the dataflow and imperative languages, and the translation function are adapted from previous work [2, 3, 8]. The correctness proof of the SN-Lustre to Obc translation is new. Several details of our formalization contribute to this success, but the introduction of the intermediate semantic model is central. It divides the proof into two more manageable parts and permits the statement and proof of the correctness lemma. It is used only in the proof and has no impact on compilation. The intermediate semantics encompasses both the dataflow and imperative ones: erasing memories gives the usual dataflow semantics; taking an instantaneous snapshot gives the usual imperative one.

The inclusion of the fuse optimization is all but obligatory in the clock-directed approach to compiling Lustre. Its verification within Obc demonstrates two advantages of the presented framework: properties of the source language and translation function can be exploited to justify optimizations on the imperative code; and the simple imperative language is a useful setting both for implementation and verification of optimizations. Expressing fuse, Fusible, and the proof of fuse $s \approx s$ in Clight is surely possible but also surely more demanding. Even as we progressively enrich our language with types, operators, and external functions from Clight, we do not expect the trade-off to change fundamentally.

In our implementation and verification of the translation of SN-Lustre to Obc and of the optimization of Obc, we abstract over the exact values, types, and operators used in programs. This separation of concerns simplifies formalization and proof, and, in principle, permits instantiations with different host languages. We integrate elements of Clight into our dataflow and intermediate languages and exploit CompCert to generate assembly code and formally relate the behavior of this code to the source dataflow model.

We describe a generation pass that transforms generic Obc programs into Clight programs. The use of a small library of separation assertions is decisive in expressing and reasoning about the invariants needed to show correctness: namely that the tree structures in the Obc semantics are correctly compiled into nested records in Clight. It quarantines the problem of reasoning about alignment and aliasing, and greatly simplifies the correctness proof of the generation pass, which is already challenging enough.

Acknowledgments

This work was partially funded by the ITEA 3 project 14014 ASSUME (Affordable Safe & Secure Mobility Evolution) and the Émergence(s) program of Paris.

We are indebted to Cédric Auger, Jean-Louis Colaço, and Grégoire Hamon for their earlier work. We thank Guillaume Baudart for his suggestions, Adrien Guatto for interesting discussions, and Adrien Gauffriau, Soukayna M'Sirdi, and Jean Souyris for their help with the industrial application.

References

[1] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, Apr. 2014.

[2] C. Auger. *Compilation certifiée de SCADE/LUSTRE*. PhD thesis, Université Paris Sud 11, Orsay, France, Apr. 2013.

[3] C. Auger, J.-L. Colaço, G. Hamon, and M. Pouzet. A formalization and proof of a modular Lustre code generator. Draft, Jan. 2013.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46, Waidhofen/Ybbs, Austria, Oct. 2010. Springer.

[5] G. Baudart, A. Benveniste, and T. Bourke. Loosely Time-Triggered Architectures: Improvements and comparisons. *ACM Transactions on Embedded Computing Systems*, 15(4): article no. 71, Aug. 2016.

[6] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. A hybrid synchronous language with hierarchical automata: Static typing and translation to synchronous code. In *Proceedings of the 11th ACM International Conference on Embedded Software (EMSOFT 2011)*, pages 137–147, Taipei, Taiwan, Oct. 2011. ACM Press.

[7] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Divide and recycle: Types and compilation for a hybrid synchronous language. In J. Vitek and B. De Sutter, editors, *Proceedings of the 12th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2011)*, pages 61–70, Chicago, USA, Apr. 2011. ACM Press.

[8] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. 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)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM Press.

[9] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43 (3):263–288, Oct. 2009.

[10] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Proceedings of the 14th International Symposium on Formal Methods (FM 2006)*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, Hamilton, Canada, Aug. 2006. Springer.

[11] F. Boniol and V. Wiels. The Landing Gear System Case Study. In *ABZ 2014: The Landing Gear Case Study—Proceedings of the Case Study Track at the 4th International Conference on Abstract State Machines*, volume 433 of *Communications in Computer Information Science*, Toulouse, France, 2014. Springer.

[12] S. Boulmé and G. Hamon. Certifying synchrony for free. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Computer Science*, pages 495–506, Havana, Cuba, Dec. 2001. Springer.

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

[14] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In U. Voges, editor, *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP'01)*, number 2187 in Lecture Notes in Computer Science, pages 215–226, Budapest, Hungary, Sept. 2001. Springer.

[15] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings of the 4th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2003)*, pages 153–162. ACM Press, 2003.

[16] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli. CoCoSpec: A mode-aware contract language for reactive systems. In R. De Nicola and E. Kühn, editors, *Proceedings of the 14th International Conference on Software Engineering and Formal Methods (SEFM 2016)*, volume 9763 of *Lecture Notes in Computer Science*, pages 347–366, Vienna, Austria, July 2016. Springer.

[17] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 model checker. In S. Chaudhuri and A. Farzan, editors, *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517, Toronto, Canada, July 2016. Springer.

[18] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[19] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155, Philadelphia, Pennsylvania, USA, Oct. 2003. Springer.

[20] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In W. Wolf, editor, *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT 2005)*, pages 173–182, Jersey City, USA, Sept. 2005. ACM Press.

[21] S. Coupet-Grimal and L. Jakubiec. Hardware verification using co-induction in Coq. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *Lecture Notes in Computer Science*, pages 91–108, Nice, France, Sept. 1999. Springer.

[22] A. Dieumegard, P.-L. Garoche, T. Kahsai, A. Taillar, and X. Thirioux. Compilation of synchronous observers as code contracts. In *Proceedings of the 30th ACM Symposium on*

Applied Computing (SAC'15), pages 1933–1939, Salamanca, Spain, Apr. 2015. ACM Press.

[23] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *Proceedings of the 13th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2012)*, pages 51–60, Beijing, China, June 2012. ACM Press.

[24] E. Gimenez and E. Ledinot. Certification de SCADE V3. Rapport final du projet GENIE II, Verilog SA, Jan. 2000.

[25] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. B. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design*, pages 15:1–15:9, Portland, OR, USA, Nov. 2008. IEEE.

[26] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[27] N. Halbwachs and L. Mandel. Simulation and verification of aysnchronous systems by means of a synchronous model. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD 2006)*, pages 3–14, Turku, Finland, June 2006. IEEE.

[28] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[29] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, volume 528 of *Lecture Notes in Computer Science*, pages 207–218, Passau, Germany, Aug. 1991. Springer.

[30] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, Sept. 1992.

[31] N. Halbwachs, J.-C. Fernandez, and A. Bouajjani. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Proceedings of the 6th International Symposium on Lucid and Intensional Programming (ISLIP'93)*, Quebec, Canada, Apr. 1993.

[32] L. Holenderski. Lustre. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems—Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*, chapter 6, pages 101–112. Springer, Berlin, 1995.

[33] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2001)*, pages 14–26, London, UK, Jan. 2001. ACM Press.

[34] N. Izerrouken, X. Thirioux, M. Pantel, and M. Strecker. Certifying an automated code generator using formal tools: Preliminary experiments in the GeneAuto project. In *Proceedings of the 4th European Congress on Embedded Real-Time Software (ERTS 2008)*. Société des Ingénieurs de l'Automobile, Jan./Feb. 2008.

[35] E. Jahier, P. Raymond, and N. Halbwachs. *The Lustre V6 Reference Manual*. Verimag, Grenoble, Dec. 2016.

[36] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In H. Seidl, editor, *21st European Symposium on Programming (ESOP 2012), held as part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416, Tallinn, Estonia, Mar./Apr. 2012. Springer.

[37] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proceedings of the International Federation for Information Processing (IFIP) Congress 1974*, pages 471–475. North-Holland, Aug. 1974.

[38] T. Kahsai and C. Tinelli. PKIND: A parallel k-induction based model checker. In J. Barnat and K. Heljanko, editors, *Proceedings of the 10th International Workshop on 2011*, number 72 in Electronic Proceedings in Theoretical Computer Science, pages 55–62, Snowbird, UT, USA, July 2011.

[39] G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In L. Beringer and A. Felty, editors, *Proceedings of the 3rd International Conference on Interactive Theorem Proving (ITP 2012)*, volume 7406 of *Lecture Notes in Computer Science*, pages 332–337, Princeton, NJ, USA, Aug. 2012. Springer.

[40] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[41] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, July 2008.

[42] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system: Documentation and user's manual*. Inria, 4.03 edition, Apr. 2016.

[43] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 2009)*, pages 78–89, Savannah, GA, USA, Jan. 2009. ACM Press.

[44] F. Maraninchi and N. Halbwachs. Compiling Argos into Boolean equations. In B. Jonsson and J. Parrow, editors, *Proceedings of the 4th International Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '96)*, volume 1135 of *Lecture Notes in Computer Science*, pages 72–89, Uppsala, Sweden, Sept. 1996. Springer.

[45] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[46] *Simulink—Using Simulink*. The Mathworks, Natick, MA, U.S.A., 5.1 edition, Sept. 2003. Release 13SP1.

[47] *Simulink® Reference*. The Mathworks, Natick, MA, U.S.A., r2016b edition, Sept. 2016. Release 2016b.

[48] V. C. Ngo, J.-P. Talpin, and T. Gautier. Translation validation for synchronous data-flow specification in the SIGNAL compiler. In S. Graf and M. Viswanathan, editors, *Proceedings of the 35th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems*

(FORTE 2015), volume 9039 of *Lecture Notes in Computer Science*, pages 66–80, Grenoble, France, June 2015. Springer.

[49] V.-C. Ngo, J.-P. Talpin, T. Gautier, L. Besnard, and P. Le Guernic. Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES'15)*, pages 109–112, St. Goar, Germany, June 2015. ACM.

[50] B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaço, T. Moniot, and P. Wang. Certified development tools implementation in Objective Caml. In P. Hudak and D. S. Warren, editors, *Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL 2008)*, number 4902 in Lecture Notes in Computer Science, pages 2–17, San Francisco, CA, USA, Jan. 2008.

[51] C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 383–413. Cambridge University Press, 2009.

[52] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1998.

[53] F. Pottier and Y. Régis-Gianas. *Menhir Reference Manual*. Inria, Aug. 2016.

[54] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, Apr. 2006.

[55] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks: An efficient symbolic representation. In *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT 2009)*, pages 215–224, Grenoble, France, Oct. 2009. ACM Press.

[56] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-V3*. PhD thesis, Grenoble INP, 1991.

[57] P. Raymond. The Lustre V4 distribution. http://www-verimag.imag.fr/The-Lustre-Toolbox.html, Sept. 1992.

[58] P. Raymond. Recognizing regular expressions by means of dataflow networks. In F. Meyer auf der Heide and B. Monien, editors, *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, number 1099 in Lecture Notes in Computer Science, pages 336–347, Paderborn, Germany, July 1996. Springer.

[59] J. C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE.

[60] M. Ryabtsev and O. Strichman. Translation validation: From Simulink to C. In A. Bouajjani and O. Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 696–701, Grenoble, France, June 2009. Springer.

[61] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo, editor, *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT 2004)*, pages 259–268, Pisa, Italy, Sept. 2004. ACM Press.

[62] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Proceedings of the 1st International Conference on Application of Concurrency to System Design (ACSD 2001)*, pages 143–154, Newcastle upon Tyne, UK, June 2001. IEEE.

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

[64] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., 1985.