Verified Compilation of Synchronous Dataflow with State Machines

TIMOTHY BOURKE, BASILE PESIN, and MARC POUZET, Inria and École normale supérieure, CNRS, PSL University, France

Safety-critical embedded software is routinely programmed in block-diagram languages. Recent work in the Vélus project specifies such a language and its compiler in the Coq proof assistant. It builds on the CompCert verified C compiler to give an end-to-end proof linking the dataflow semantics of source programs to traces of the generated assembly code. We extend this work with switched blocks, shared variables, reset blocks, and state machines; define a relational semantics to integrate these block- and mode-based constructions into the existing stream-based model; adapt the standard source-to-source rewriting scheme to compile the new constructions; and reestablish the correctness theorem.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Software verification**; **Compilers**; • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: stream languages, verified compilation, interactive theorem proving

1 INTRODUCTION

Control software is specified in block-diagram languages by interconnecting blocks that represent filters and other transformations on signals. Block diagrams are compiled to imperative code for execution on embedded platforms. This approach is implemented in tools like Mathworks Simulink and Ansys Scade Suite. Its underlying principles are studied in dataflow programming languages where signals are modeled as streams of values and blocks are modeled as functions between streams. Many block-diagram languages provide activation blocks inspired by Statecharts [32] to express sequential, conditional, and mode-dependent behaviors.

A fundamental question is how to guarantee that the code generated for a block-diagram language is correct. One possibility is to encode the language semantics, compilation algorithms, and correctness proofs in an interactive theorem prover. The Vélus project [11–13] takes this approach in the Coq proof assistant [25] by building on the CompCert C compiler [34]. It treats the dataflow core of the Lustre programming language [27] extended with a reset primitive [30]. In this article, we extend Vélus with activation blocks from Lucid Synchrone [17, 40] and Scade 6 [23], namely reset blocks, switch blocks, local scopes, and hierarchical state machines. We propose new semantic definitions suitable for reasoning in a proof assistant, adapt an existing compilation scheme based on source-to-source rewriting [22], and outline a machine-checked, end-to-end correctness proof. The source code and an online demonstration are available at https://velus.inria.fr/emsoft2023.

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023.

EMSOFT23, 2023, Hamburg, Germany
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

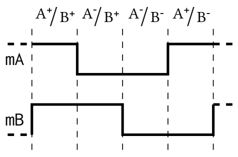
```
node count_up(inc : int) returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

Listing 1. Measuring phase duration

inc	50	50	50	50	50	50	50	...
o	50	100	150	200	250	300	350	...

Table 1. Example trace of count_up

```
node drive_sequence(step : bool)
returns (last mA : bool = true; last mB : bool = true)
let
  switch step
  | true do (mA, mB) = (not (last mB), last mA)
  | false do (mA, mB) = (last mA, last mB)
  end;
tel
```



Listing 2. Motor drive sequence

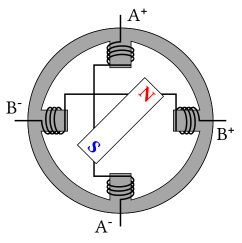


Fig. 1. Stepper motor

1.1 Programming with Activation Blocks

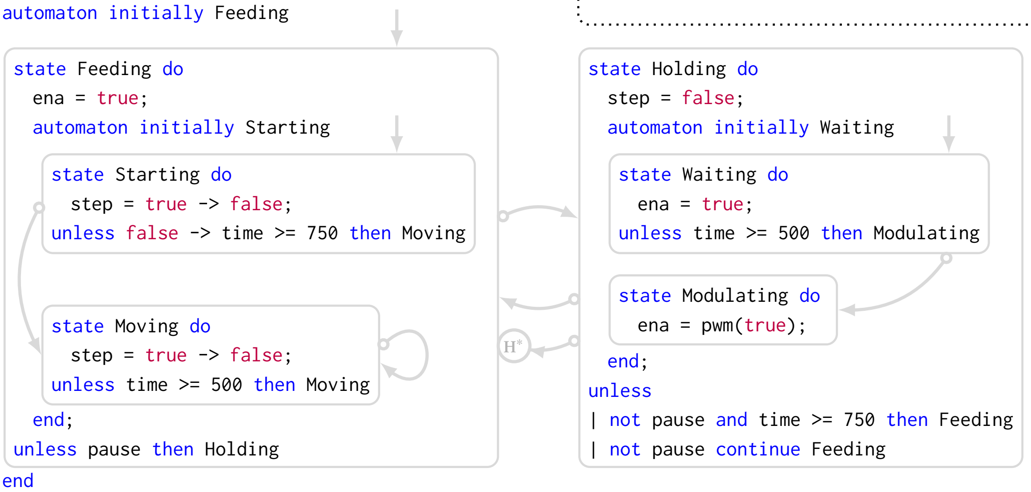
A stepper motor control program is a simple example of the use of activation blocks. Figure 1 sketches the inside of a certain stepper motor. There is a *rotor* made from a magnet turning within a fixed *stator* that has two windings named A and B. Passing current through a winding, in either direction, creates a magnetic field that acts on the rotor. Energizing the windings with the sequence of phases A^+/B^+, A^-/B^+, A^-/B^-, A^+/B^- moves the rotor clockwise from the upper-right in 90° steps.

Let's start with the function in listing 1 that counts upward from zero by a given increment. We will later instantiate it to measure the duration of a motor phase. An example trace is shown in table 1. The *node* called `count_up` maps an input stream `inc` to an output stream `o`. Its body contains a single equation that defines `o` as zero *followed by* (`fby`) the pointwise addition of itself and `inc`. Such purely dataflow nodes are accepted by previous versions of Vélus [11].

Switch blocks and shared variables. The motor windings are energized by a driver chip with three digital input signals: `mA`, `mB`, and `ena`. Assume for now that `ena = true`. To generate the sequence of phases described above, the `drive_sequence` node in listing 2 alternates the values of `mA` and `mB`. The body of this node contains a `switch` block: the value of the `step` input determines which set of equations applies at each instant to define the output signals. In the true branch, it is tempting to put `(mA, mB) = (true, false) fby (not mB, mA)`, but it would then be difficult to refer to the previous values of these streams in the false branch. Instead we use *shared variables* [28, §6; 22] by declaring `mA` and `mB` with *initial last values* and writing `last mA` and `last mB` to access their previous values. The overall effect is to advance to the next phase when `step` is true and to otherwise hold the current phase. The false branch, shown in gray, can be omitted as the semantics and compiler complete partial definitions of variables with initial last values. This example includes a `switch` on a boolean condition, but in general any enumerated type may be used.

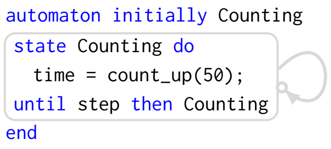
While the `count_up` node is best viewed in terms of functions on streams, the `drive_sequence` node is more naturally viewed in terms of compositions of transition systems. Both interpretations are valid and useful. The challenge is to extend the stream-oriented semantic definitions of Vélus to treat block-based features like `switch` and `last`. We also extend the existing formalized dependency analysis [13] to support activation blocks.

Reset blocks. The duration of each phase and the value of `ena` depend on state-based logic that we detail shortly. For now, consider just the first three lines in the body of `feed_pause` from listing 3.



```
node feed_pause(pause : bool) returns (ena,step : bool)
var time : int;
let
  reset
    time = count_up(50);
  every (false fby step);

  automaton initially Feeding
  state Feeding do
    ena = true;
    automaton initially Starting
    state Starting do
      step = true -> false;
    unless false -> time >= 750 then Moving

    state Moving do
      step = true -> false;
    unless time >= 500 then Moving
    end;
  unless pause then Holding

  state Holding do
    step = false;
    automaton initially Waiting
    state Waiting do
      ena = true;
    unless time >= 500 then Modulating

    state Modulating do
      ena = pwm(true);
    end;
  unless
  | not pause and time >= 750 then Feeding
  | not pause continue Feeding
  end
tel
```

Listing 3. Paper feeding control (the graphical notation [2] is superimposed in grey)



```
automaton initially Counting
state Counting do
  time = count_up(50);
until step then Counting
end
```

Listing 4. Alternative definition of time

They define a local variable time using count_up with an increment of 50. The counter is reset after the phase changes, that is, when step was true in the previous instant. The fby precludes a circular dependency since the value of time is itself needed to determine that of step.

Adding reset blocks requires extending the prior formalization of a reset operator for node instances [12]. It turns out that the intermediate language, compilation algorithms, and proofs reported in earlier work must be adapted to compile nested reset blocks.

State Machines. The controller in listing 3 holds the rotor in place when the pause input is true. When the rotor is not moving, the controller reduces energy loss by modulating the current in the windings via the ena output. This logic is specified by an automaton with two states: Feeding, at left, and Holding, at right. The automaton starts in Feeding. Transitions are listed below states after the unless keyword. Within Feeding, ena is always true and a nested automaton defines the duration of the current phase, which must be longer initially and after a long pause as the rotor (re)gains momentum. Both inner states define step with the initialization operator (->) so that it is true only in the instant a state is entered. The same operator is used in one of the transition guards to compensate for the delay in resetting count_up. Within Holding, step is always false and a nested automaton defines the value of ena. If the current phase duration exceeds 500, the inner automaton enters a state where ena is defined using pulse-width modulation (the pwm node, not shown). Two transitions leave the Holding state. The earlier one has priority. Both transitions require pause to be false and both return to Feeding. If the phase duration exceeds 750, the first transition, like all the others with the then keyword, resets the destination state. The second transition uses the continue keyword to enter the destination state 'with history' [32, p.8], that is, without resetting it.

All transitions in listing 3 are marked as *strong* by the `unless` keyword [40, §1.6.1]. A strong transition aborts a state immediately [6, §5.1] specifying a state to enter in the same instant. Listing 4 defines `time` using the `until` keyword to specify a *weak* transition [40, §1.6.2]. A weak transition exits a state after it has been active [6, §5.1] specifying a state to enter at the next instant. In our formalization, a given state machine must use either strong or weak transitions exclusively. To compensate, the initial state of a weak state machine may be specified by a list of transitions.

As the example shows, state machines are a sophisticated specification mechanism that allow programmers to freely mix dataflow equations with sequential and mode-based logic. In this article, we define their semantics with a set of novel relational predicates and use it to verify an implementation of the standard compilation scheme based on source-to-source rewriting [20].

1.2 Compiler Correctness

The Vélus compiler comprises (i) a purely functional program that transforms the abstract syntax tree of a source program into one representing a Clight program, (ii) Menhir [39] and OCaml files for parsing, calling the compiler, and scheduling, (iii) the CompCert compiler [10, 34], and (iv) machine-checked specifications and proofs. The overall correctness theorem is stated below.

Theorem 1.1 (Compiler Correctness).

$$\begin{array}{rl} \textbf{if} & \mathsf{compile}\ G\ f = \mathsf{OK}\ asm \\ \textbf{and} & G \vdash f(xs) \Downarrow ys \\ \textbf{and} & G(f) = \texttt{node}\ f(x_1 : ty_1; \ldots; x_m : ty_m)\ \texttt{returns}\ (\ldots)\ blk \quad \textbf{and} \quad \forall i \in 1 \ldots n, \vdash xs_i : ty_i \\ \textbf{then} & \exists T, asm \Downarrow T \wedge T \sim \langle \mathsf{Load}(xs(n)) \cdot \mathsf{Store}(ys(n)) \rangle_{n=0}^{\infty} \end{array}$$

In other words, **if** the compiler successfully transforms a program G with main node f into an assembly program asm **and** f has a semantics mapping a list of input streams xs to a list of output streams ys **and** the input streams are well typed **then** the generated assembly code produces an infinite trace that alternates between reading input stream values and writing output stream values. We extend the compilation function and source semantic model of previous work [11–13] to treat the activation blocks presented in the introduction. Section 2 describes the semantic model ($G \vdash f(xs) \Downarrow ys$) focusing on challenges posed by the new constructions. Section 3 describes the compiler modifications (compile) and correctness proofs. CompCert's assembly language semantics ($asm \Downarrow T$) are described elsewhere [35]. Section 4 discusses generated code quality.

2 RELATIONAL STREAM SEMANTICS

In this section, we describe a new abstract syntax for Vélus (section 2.1), recall the existing semantic model (section 2.2), and present semantic rules for activation blocks (sections 2.3 to 2.6). Figure 5 summarizes the main notations used in the semantic definitions.

2.1 Abstract Syntax

The syntax of expressions e, see figure 2, is unchanged from [13] but for the addition of `last x` and enumerated values. There are primitive and enumerated constants, variables, last variables, unary and binary operators (from Clight), `fby`s, and initializations. A `when` produces slower streams by sampling faster ones when a stream x is equal to an enumerated constant C. Inversely, a `merge` combines slower streams according to a stream x and a list of labeled expressions. As will become clear, these two operators provide intermediate constructions as activation blocks are reduced to guarded commands. While at most one branch of a `merge` is active in any instant, all branches of a `case` are active together; the value of the guard determines the overall value of the expression. Finally, an expression may instantiate a node, possibly subject to a boolean reset condition.

$$\begin{array}{rcl}
e & ::= & c \mid C \mid x \mid \texttt{last}\ x \mid \diamond\, e \mid e \oplus e \\
 & \mid & e^{+}\ \texttt{fby}\ e^{+} \mid e^{+}\ \texttt{->}\ e^{+} \mid e^{+}\ \texttt{when}\ C\ (\ x\ ) \\
 & \mid & \texttt{merge}\ x\ (\ C\ \texttt{=>}\ e^{+}\ )^{+} \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ (\ C\ \texttt{=>}\ e^{+}\ )^{+} \\
 & \mid & f\ (\ e^{+}\ ) \mid (\ \texttt{reset}\ f\ \texttt{every}\ e\ )\ (\ e^{+}\ )
\end{array}$$

Fig. 2. Expressions

$$\begin{array}{rcl}
td & ::= & \texttt{type}\ ty = (\ \mid\ C\ )^{+} \\
d & ::= & x^{ck}_{ty} \\
ck & ::= & \bullet \mid ck\ \texttt{on}\ C\ (\ x\ ) \\
n & ::= & \texttt{node}\ f\ (\ d^{+}\ )\ \ \texttt{returns}\ \ (\ d^{+}\ )\ blk \\
G & ::= & td^{*}\ n^{+}
\end{array}$$

Fig. 3. Nodes and Programs

$$\begin{array}{rcl}
blk & ::= & x^{+} = e^{+}\ ; \\
 & \mid & \texttt{var}\ loc^{*}\ \texttt{let}\ blk^{+}\ \texttt{tel} \\
 & \mid & \texttt{reset}\ blk^{+}\ \texttt{every}\ e \\
 & \mid & \texttt{switch}\ e\ (\ C\ \texttt{do}\ blk^{+}\ )^{+}\ \texttt{end} \\
 & \mid & \texttt{automaton initially}\ autinits \\
 & & \texttt{(state}\ C\ autscope\ )^{+}\ \texttt{end} \\
 & \mid & \texttt{automaton initially}\ C \\
 & & \texttt{(state}\ C\ \texttt{do}\ blk^{+}\ \texttt{unless}\ trans^{+}\ )^{+}\ \texttt{end} \\
loc & ::= & d \mid \texttt{last}\ d = e \\
autinits & ::= & C \mid \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ autinits \\
autscope & ::= & \texttt{var}\ loc^{*}\ \texttt{do}\ blk^{+}\ \texttt{until}\ trans^{+} \\
trans & ::= & \mid\ e\ \texttt{continue}\ C \mid \mid\ e\ \texttt{then}\ C
\end{array}$$

Fig. 4. Blocks and Declarations

The `bool` type is a predeclared enumerated type: `type bool = true | false` for which we provide specialized syntax for `case` (`if`/`then`/`else`), `e when true(x)` (`e when x`), and `e when false(x)` (`e when not x`). Vélus optimizes this type and its standard operations when translating to Clight.

A program G—see figure 3—comprises enumerated type definitions and node definitions. Each variable of a node is associated with a type `ty` and a clock type `ck`, which is either the base clock • or a sampled clock `ck on C(x)`. The body of a node is a *block*, see figure 4: an equation, a local scope, a reset block, a switch block, a weak state machine, or a strong state machine. The preexisting Vélus compiler [13] only permits a single local scope containing equations.

2.2 Synchronous Core

In a program G, a node f relates a list of input streams $xs_1, \ldots, xs_n$ to a list of output streams $ys_1, \ldots, ys_m$. The rule defining this relation is given at the top of figure 6. The rule asserts that f is declared in G with input variables $x_1, \ldots, x_n$, output variables $y_1, \ldots, y_m$, and body blk. It requires the existence of a *history* H that associates variables to streams. (Table 1 shows a simple concrete example of a history.) The history is constrained by the input and output streams, and also by a predicate on blocks $G, H, bs \vdash blk$. The block context includes a *base clock* bs which is a stream of booleans giving its “rhythm”. The base-of function returns the clock of the fastest stream among the node inputs. Figure 6 also shows the rule for equations—other blocks are treated one-by-one in the following—and two of the rules for expressions. As in the preexisting development [13], which follows [18, §3.2] and [24, §3], each equation constrains the history to associate a list of variables $x_1, \ldots, x_n$ to a list of corresponding streams. In expressions, a variable x is simply associated with a singleton list containing the corresponding stream from H. The rule for a constant `c` refers to the coiterative function const that maps a base clock to explicit absent $\langle\rangle$ and present $\langle v \rangle$ markers. Streams are modeled by the standard coinductive type whose only constructor ‘cons’ is written ‘·’.

The absent and present markers are used to model the effect of sampling. Figure 7 shows the three rules that define the semantic operator for `when`: (i) if both input streams are absent, so is the output stream, (ii) if the sampled stream is present and the condition stream is present and equal to the constant parameter, then the sampled value is propagated, and (iii) if the sampled and condition streams are present but the latter is not equal to the parameter, then the output stream is absent. These rules are satisfied by the example streams in table 2, that sample on an enumerated type with labels A, B, and C. As can be seen, the absent markers have the effect of encoding the synchronization of streams, which propagates through to the generated code that

Values, streams, and lists	
$\langle\rangle$ / $\langle v\rangle$	`svalue`: absence / presence with v from CompCert
$v \cdot vs$	Constructor for coinductive streams, `Stream` α
$vs_1 \equiv vs_2$	Pointwise stream equivalence
$[x_1, ..., x_n]$	`list` α of length n
$[x_i]^i$	List comprehension; lists with the same index have the same length
Naming conventions	
G	Global environment: maps a function identifier to its definition
H	Stream environment: maps a variable identifier to a `Stream svalue`
bs	Base clock of type `Stream bool`
Γ	Typing environment: maps a variable identifier to its clock type
Semantic predicates	
$G \vdash f(xs) \Downarrow ys$	A node $G(f)$ relates input streams xs to output streams ys
$G, H, bs \vdash blk$	In a program G, H satisfies the constraints of block blk
$G, H, bs \vdash e \Downarrow vss$	Associates an expression with a list of streams
$H, bs \vdash ck \Downarrow bs'$	Associates a clock type with a `Stream bool`
$G, H, bs \vdash_{\mathrm{L}}$ `last` $x = e$	H satisfies the `last` declaration
$G, H, bs \vdash_{\mathrm{I}} inits \Downarrow sts$	Associates initialization conditions to a state stream sts
$G, H, bs, C \vdash_{\mathrm{TR}} trans \Downarrow sts$	For state label C, associate transitions to a state stream sts
$G, H, bs, C \vdash_{\mathrm{W}} scope \Downarrow sts$	A weak automaton state constrains H and has state transitions sts
Clock-typing predicates	
$G, \Gamma \vdash_{\mathrm{wc}} e : cks$	An expression e is well clocked with clock types cks
$G, \Gamma \vdash_{\mathrm{wc}} blk$	A block blk is well clocked
Additional predicates	
$\vdash xs \Downarrow ty$	All present values in the stream xs have type ty
$asm \Downarrow T$	CompCert small-step assembly semantics [35, §§3.4 & 13]
$G, H, bs \vdash_{\mathrm{NL}} eq$	NLustre equation semantics [12, Figure 6]

Fig. 5. Summary of main semantic notations

$$\frac{G(f) = \texttt{node}\ f(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \qquad \forall i \in 1...n,\ H(x_i) \equiv xs_i \qquad \forall j \in 1...m,\ H(y_j) \equiv ys_j \qquad G, H, (\text{base-of}\ (xs_1, \ldots, xs_n)) \vdash blk}{G \vdash f(xs_1, \ldots, xs_n) \Downarrow (ys_1, \ldots, ys_m)}$$

$$\frac{\forall i,\ H(x_i) \equiv vs_i \qquad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash [x_i]^i = es}$$

$$\frac{H(x) \equiv vs}{G, H, bs \vdash x \Downarrow [vs]} \qquad \frac{}{G, H, bs \vdash \mathtt{c} \Downarrow [\text{const}\ bs\ \mathtt{c}]} \qquad \begin{array}{l}\text{const}\ (\mathsf{T} \cdot bs)\ \mathtt{c} \equiv \langle \mathtt{c} \rangle \cdot \text{const}\ bs\ \mathtt{c} \\ \text{const}\ (\mathsf{F} \cdot bs)\ \mathtt{c} \equiv \langle\rangle \cdot \text{const}\ bs\ \mathtt{c}\end{array}$$

Fig. 6. Semantic rules for variables, constants, equations, and nodes as in [13]

iteratively samples and calculates 'column-by-column'. They only appear in the semantic model and not as explicit values in the generated code. The two rules for the semantic operator for `merge` are also shown in figure 7. The following lemma shows a key property of the two operators.

Lemma 2.1 (Correspondence of when and merge).

$$\text{merge}\ cs\ (\text{when}^{C_1}\ vs\ cs, \ldots, \text{when}^{C_n}\ vs\ cs) \equiv vs$$

cs	$\langle A\rangle$	$\langle A\rangle$	$\langle C\rangle$	$\langle A\rangle$	$\langle B\rangle$	$\langle\rangle$	$\langle C\rangle$	$\dots$
vs	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle 5\rangle$	$\langle\rangle$	$\langle 7\rangle$	$\dots$
$\mathsf{when}^A\ vs\ cs$	$\langle 1\rangle$	$\langle 2\rangle$	$\langle\rangle$	$\langle 4\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\dots$
$\mathsf{when}^B\ vs\ cs$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 5\rangle$	$\langle\rangle$	$\langle\rangle$	$\dots$
$\mathsf{when}^C\ vs\ cs$	$\langle\rangle$	$\langle\rangle$	$\langle 3\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 7\rangle$	$\dots$

Table 2. Sample execution of when

$$\begin{aligned}
\mathsf{when}^C\ (\langle\rangle\cdot xs)\ (\langle\rangle\cdot cs) &\equiv \langle\rangle\cdot \mathsf{when}^C\ xs\ cs\\
\mathsf{when}^C\ (\langle v\rangle\cdot xs)\ (\langle C\rangle\cdot cs) &\equiv \langle v\rangle\cdot \mathsf{when}^C\ xs\ cs\\
\mathsf{when}^C\ (\langle v\rangle\cdot xs)\ (\langle C'\rangle\cdot cs) &\equiv \langle\rangle\cdot \mathsf{when}^C\ xs\ cs
\end{aligned}$$

$$\begin{aligned}
\mathsf{merge}\ (\langle\rangle\cdot xs)\ (\langle\rangle\cdot ys_1,\ \dots,\langle\rangle\cdot ys_m) &\equiv \langle\rangle\cdot \mathsf{merge}\ xs\ (ys_1,\ \dots, ys_m)\\
\mathsf{merge}\ (\langle C_i\rangle\cdot xs)\ (\langle\rangle\cdot ys_1,\ \dots,\langle v\rangle\cdot ys_i,\ \dots,\langle\rangle\cdot ys_m) &\equiv \langle v\rangle\cdot \mathsf{merge}\ xs\ (ys_1,\ \dots, ys_i,\ \dots, ys_m)
\end{aligned}$$

Fig. 7. Semantic operators for `when` and `merge`

$$\frac{\Gamma(x)=ck \qquad G,\Gamma\vdash_{\mathrm{wc}} es : [ck]^j}{G,\Gamma\vdash_{\mathrm{wc}} es\ \mathtt{when}\ C(x) : [ck\ \mathtt{on}\ C(x)]^j}
\qquad
\frac{\Gamma(x)=ck \qquad \forall i,\ G,\Gamma\vdash_{\mathrm{wc}} es_i : [ck\ \mathtt{on}\ C_i(x)]^j}{G,\Gamma\vdash_{\mathrm{wc}} \mathtt{merge}\ x\ [C_i \Rightarrow es_i]^i : [ck]^j}$$

Fig. 8. Clock typing rules for `when` and `merge`

$$\frac{\begin{array}{c}G,H,bs\vdash es_0 \Downarrow [xs_i]^i\\ G,H,bs\vdash es_1 \Downarrow [ys_i]^i\\ \forall i,\ \mathsf{fby}\ xs_i\ ys_i \equiv vs_i\end{array}}{G,H,bs\vdash es_0\ \mathtt{fby}\ es_1 \Downarrow [vs_i]^i}$$

$$\begin{aligned}
\mathsf{fby}\ (\langle\rangle\cdot xs)\ (\langle\rangle\cdot ys) &\equiv \langle\rangle\cdot \mathsf{fby}\ xs\ ys\\
\mathsf{fby}\ (\langle v_1\rangle\cdot xs)\ (\langle v_2\rangle\cdot ys) &\equiv \langle v_1\rangle\cdot \mathsf{fby1}\ v_2\ xs\ ys
\end{aligned}$$

$$\begin{aligned}
\mathsf{fby1}\ v_0\ (\langle\rangle\cdot xs)\ (\langle\rangle\cdot ys) &\equiv \langle\rangle\cdot \mathsf{fby1}\ v_0\ xs\ ys\\
\mathsf{fby1}\ v_0\ (\langle v_1\rangle\cdot xs)\ (\langle v_2\rangle\cdot ys) &\equiv \langle v_0\rangle\cdot \mathsf{fby1}\ v_2\ xs\ ys
\end{aligned}$$

Fig. 9. Semantics of `fby` operator

The rules for when and merge are defined as relations, rather than functions. A semantics is not given to programs that combine mismatched streams, for example, $\mathsf{when}^C\ (\langle v\rangle\cdot xs)\ (\langle\rangle\cdot cs)$ is not defined. Such programs are rejected by a static analysis based on clock typing rules [15] implemented as a type system [24]. The clock typing rules for `when` and `merge` are shown in figure 8. For `when`, the condition and sub-expressions must have the same clock and the overall clock is sampled on the condition. For `merge`, the sub-expression clocks must be complementary with respect to the condition variable whose clock matches that of the result.

Finally, figure 9 gives the rules for a `fby` expression and associated semantic operator. The expression rule requires that the argument expressions give rise to equal-length lists of streams to which the semantic operator is applied pointwise. The fby semantic operator is defined by a coinductive relation that skips over absences until both streams yield present values, it propagates the first value, and 'buffers' the second via an auxiliary fby1 relation which thereafter ignores the first stream but requires it remain synchronous with the second one [24, figure 2].

The semantic rules and operators are expressed directly in Coq using inductive predicates on syntax and coinductive predicates on streams. They are convenient for deductive reasoning where, as usual, rule inversion is used to decompose assumptions and forward rule application is used to define introduced elements; we return to this idea in section 3.2.

$$\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\ G, \mathrm{when}^{C_i}\ (H, bs)\ cs \vdash blks_i \qquad \star}{G, H, bs \vdash \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end}}
\qquad
\frac{G, \Gamma \vdash_{\mathrm{wc}} e : [ck] \qquad \forall i,\ G, \Gamma' \vdash_{\mathrm{wc}} blks_i \qquad \forall x\ ck',\ \Gamma'(x) = ck' \implies \Gamma(x) = ck \land ck' = \bullet}{G, \Gamma \vdash_{\mathrm{wc}} \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end}}$$

Fig. 10. Semantics and clock typing rule for switch blocks

$$\frac{\forall x,\ x \in \mathrm{dom}(H') \iff x \in locs \qquad \forall x\ e,\ (\texttt{last}\ x = e) \in locs \implies G, H + H', bs \vdash_{\mathrm{L}} \texttt{last}\ x = e \qquad G, H + H', bs \vdash blks}{G, H, bs \vdash \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel}}$$

$$\frac{G, H, bs \vdash e \Downarrow [vs_0] \qquad H(x) \equiv vs_1 \qquad H(\texttt{last}\ x) \equiv \mathrm{fby}\ vs_0\ vs_1}{G, H, bs \vdash_{\mathrm{L}} \texttt{last}\ x = e}
\qquad
\frac{H(\texttt{last}\ x) \equiv vs}{G, H, bs \vdash \texttt{last}\ x \Downarrow [vs]}$$

Fig. 11. Semantics of local scope blocks and last expressions

2.3 Switch Blocks

As described above, the semantics of a node that only contains equations is defined as an intersection of constraints on a single history. We treat activation blocks by sampling, extending, and otherwise refining histories. The switch construct is essentially a block-based version of merge with implicit sampling of variables and constants. We make the sampling explicit by lifting the definition of when to histories such that $(\mathrm{when}^C\ H\ cs)(x) = \mathrm{when}^C\ (H(x))\ cs$. In the rule of figure 10 at left, the expression evaluates to a stream cs and, unpacking the abbreviated notation, each block $blks_i$ acts under a sampled history ($\mathrm{when}^{C_i}\ H\ cs$) and base clock ($\mathrm{when}^{C_i}\ bs\ cs$). In instants when a block is inactive, the sampling ensures that the constants and variables in its expressions are absent and therefore that fbys and node instances 'stutter'. Stuttering is ensured directly for fbys by the rules in figure 9 and recursively for node instances. At any instant, only the active branch imposes its constraints on H and, recursively, only for the variables defined within it.

The implicit completion of last variables requires an additional premise, which, for every branch i missing a definition for x, adds the constraint corresponding to `x = last x` to the sampled history:

$$\star : \qquad \forall i, x \in \bigcup_j \mathrm{Def}(blks_j) \backslash \mathrm{Def}(blks_i),\ (\mathrm{when}^{C_i}\ H\ cs)(x) \equiv (\mathrm{when}^{C_i}\ H\ cs)(\texttt{last}\ x).$$

The sampling of variables in switch branches must respect the clock type rule for when from figure 8. That is, free and defined variables within switch branches must have the same clock as the switch expression. The resulting clock typing rule is shown in figure 10 at right: each block must be well typed in an environment Γ' that only contains variables from the original environment Γ that have the same clock ck as the expression. Within a block, the clock ck is renamed $\bullet$ so that it corresponds to the sampled value of bs. The clock type annotations of subexpressions are adapted accordingly. This definition adapts the original operator on clock typing environments [24, §2.1.3] which does not require renaming ck to $\bullet$ within switch branches. We found that renaming facilitates the proof and use of an alignment property that relates streams to their clock types; that is, a stream is present in an instant iff its clock type evaluates to true for a given history and base clock. The disadvantage is that elaboration and compilation must set clock type annotations appropriately.

2.4 Local Scopes and Shared Variables

In Lustre and the preexisting Vélus compiler, local variables may only be declared at node level. With the addition of block-based constructions, it is natural and useful to allow arbitrary nesting of

rs	$\langle F\rangle$	$\langle F\rangle$	$\langle T\rangle$	$\langle F\rangle$	$\langle\rangle$	$\langle F\rangle$	$\langle T\rangle$	$\langle F\rangle$	...
xs	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle\rangle$	$\langle 5\rangle$	$\langle 6\rangle$	$\langle 7\rangle$	...
$\mathsf{mask}^0\ rs\ xs$	$\langle 1\rangle$	$\langle 2\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	...
$\mathsf{mask}^1\ rs\ xs$	$\langle\rangle$	$\langle\rangle$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle\rangle$	$\langle 5\rangle$	$\langle\rangle$	$\langle\rangle$	...
$\mathsf{mask}^2\ rs\ xs$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 6\rangle$	$\langle 7\rangle$	...

Table 3. Example instances of mask

$$\begin{aligned}\mathsf{mask}^k_{k'}\ (\mathsf{F}\cdot rs)\ (sv\cdot xs) &\equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle\rangle)\cdot \mathsf{mask}^k_{k'}\ rs\ xs\\ \mathsf{mask}^k_{k'}\ (\mathsf{T}\cdot rs)\ (sv\cdot xs) &\equiv (\text{if } k'+1 = k \text{ then } sv \text{ else } \langle\rangle)\cdot \mathsf{mask}^k_{k'+1}\ rs\ xs\end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \text{bools-of } ys \equiv rs \qquad \forall k,\ G, \mathsf{mask}^k\ rs\ (H, bs) \vdash blks}{G, H, bs \vdash \texttt{reset}\ blks\ \texttt{every}\ e}$$

Fig. 12. Definition of mask and semantics of reset blocks

local declarations [17, 22]. For semantic definitions, the concatenation of environments described in [20, §3.2] works just as well in Coq with histories. Concatenation is defined as follows.

$$(H_1 + H_2)(x) = \begin{cases} H_2(x) & \text{if } x \in \mathrm{dom}(H_2)\\ H_1(x) & \text{otherwise}\end{cases}$$

For clarity, simplicity, and to avoid problems with clock types, our compiler rejects programs that redeclare local names, precluding the case where $x \in \mathrm{dom}(H_1)$ and $x \in \mathrm{dom}(H_2)$.

Shared variables can be declared with initial last values and then accessed with the `last` operator. The semantics of local scopes must provide for this possibility. We add a distinct history to associate variables with their last values and refer to its mappings using the simplified notation $H(\mathrm{last}\ x)$. This idea works just as well in Coq as it does for typing environments [4, §3.2] and compilers [14].

The local scope rule is shown at the top of figure 11. It requires an inner history H' whose domain is the set of locally declared variables. The semantics of the blocks within a local scope are relative to the concatenation of the outer and inner histories, $H + H'$. For all variables declared with `last`, the inner history must also satisfy the rule shown at bottom left. This predicate constrains the value of last x to equal the application of the fby semantic operator to the streams associated with the initial last expression, vs_0, and the variable itself, vs_1. Applying this constraint at the point that `x` is defined coincides with the expected meaning of `last x`. Finally, the stream corresponding to the `last x` expression is then simply the one associated to last x in the history.

2.5 Reset Blocks

At first glance, it may seem difficult to reconcile resetting with dataflow programming: how to model the effects of abrupt transitions on the infinite sequences produced by node instances and `fby`s? In fact, in a higher-order dataflow language, resets can be expressed with sampling and recursion [16, §4.1; 30, §3.1]. When a reset occurs, a current node instance is paused forever by filtering its inputs, a new node is instantiated recursively, and its outputs are merged with those of preceding and succeeding instances. This idea inspired the mask operation introduced in Vélus to define the semantics of the reset operation on nodes [12]. The definition of mask for a single stream is shown at the top of figure 12 and example traces are shown in table 3. The operation is parameterized by k, the instance number, and k', the number of resets that have already occurred. It takes two arguments: *rs*, the reset stream, and *xs*, the stream to reset. As the definition and example show, $\mathsf{mask}^k\ rs\ xs$ counts the number of trues encountered on *rs*, propagates values from *xs* in the window from the kth true to just before the $(k+1)$th, and is otherwise absent.

The mask operator generalizes naturally to histories, $(\mathsf{mask}^k_{k'}\ rs\ H)(x) = \mathsf{mask}^k_{k'}\ rs\ (H(x))$, and forms the basis of the rule for reset blocks shown at bottom in figure 12. The bools-of operator produces a reset stream *rs* from the stream *ys* associated with the reset expression *e*. It replaces absence with false and otherwise projects booleans from present values. The quantification on k allows each reset 'window' to constrain a slice of the history H. As far as each instance is concerned, *blks* stutters until its window begins, acts until the subsequent reset, and then stutters again forever.

2.6 State Machines

The semantics of the state machine construct were originally defined by translation [22] and later by a transition relation between instantaneous environments [20, §4.1]. We draw on this work and the semantic operators introduced above to propose new rules based on histories.

In the transition relation semantics [20, §4], a state machine is annotated with an entry state and a boolean indicating if a reset is required. In our semantics, this information is modeled as a *state stream* that is absent when a state machine is inactive, and otherwise $\langle C, r\rangle$, where C specifies the state and r indicates whether to reset it. We introduce a semantic operator called select to encode the corresponding sampling and reset mechanism. Its definition on streams is shown in figure 13a. The operator is parameterized by a specific state C, like when, and an instance number k and count k', like mask. It acts on a state stream *sts* and an argument stream *xs* by (i) stuttering on absence, (ii) propagating values when both the state and instance are active, (iii) counting resets when the state is active, and (iv) stuttering when the state is not active. The correspondence of select with mask and when can be stated precisely, using π_1 and π_2 to project the component streams from a state stream, and shown by coinduction.

Lemma 2.2 (Correspondence of select with when and mask).

$$\mathsf{select}^{C,k}_{k'}\ sts\ xs \equiv \mathsf{mask}^k_{k'}\ (\mathsf{when}^C\ \pi_2(sts)\ \pi_1(sts))\ (\mathsf{when}^C\ xs\ \pi_1(sts))$$

The select operator is lifted to environments in the expected way and applied in the rules for state machines.

Weak state machines. The two rules for weak state machines are shown in figure 13b. The state stream *sts* is defined by applying the fby semantic operator to an initial-state stream sts_0 and a next-state stream sts_1. The presence or absence of the state stream is determined by interpreting a clock-type annotation *ck* associated with the automaton. We do not know of a better way to do this as, unlike for the `switch` guard, there is no natural candidate to give the required rhythm.

The initial state stream is defined by the rules for `if` and `else` in figure 13c. They iterate along the list of potential initial states C and their guard expressions e. For each state, the guard is interpreted to give a stream of booleans bs'. The first-of function, figure 13d, then propagates the $\langle C, r\rangle$ pair of the first true guard at each instant.

Finally, the select operator combines and imposes the constraints for each state C_i and each of its instances k, onto the history H according to the state stream *sts*. Similarly, select also combines the next-state streams from each state to determine sts_1. A separate rule is introduced for the body of a single state, `var` *locs* `do` *blks* `until` *trans*, since a state's local variables may be used in weak transition guards. This rule is identical to the one for local scopes in figure 11 but for the antecedent that determines *sts* from the list of transitions. The three rules in figure 13e for transitions—ϵ, `continue`, and `then`—follow the same principle as those for initial states. The only differences are

$$\begin{aligned}
\mathrm{select}^{C,k}_{k'}\ (\langle\rangle \cdot sts)\ (\langle\rangle \cdot xs) &\equiv \langle\rangle \cdot \mathrm{select}^{C,k}_{k'}\ sts\ xs\\
\mathrm{select}^{C,k}_{k'}\ (\langle C, \mathrm{F}\rangle \cdot sts)\ (\langle v\rangle \cdot xs) &\equiv (\mathrm{if}\ k' = k\ \mathrm{then}\ \langle v\rangle\ \mathrm{else}\ \langle\rangle) \cdot \mathrm{select}^{C,k}_{k'}\ sts\ xs\\
\mathrm{select}^{C,k}_{k'}\ (\langle C, \mathrm{T}\rangle \cdot sts)\ (\langle v\rangle \cdot xs) &\equiv (\mathrm{if}\ k' + 1 = k\ \mathrm{then}\ \langle v\rangle\ \mathrm{else}\ \langle\rangle) \cdot \mathrm{select}^{C,k}_{k'+1}\ sts\ xs\\
\mathrm{select}^{C,k}_{k'}\ (\langle C', b\rangle \cdot sts)\ (\langle v\rangle \cdot xs) &\equiv \langle\rangle \cdot \mathrm{select}^{C,k}_{k'}\ sts\ xs
\end{aligned}$$

(a) select function

$$\frac{\begin{array}{c} H, bs \vdash ck \Downarrow bs' \qquad G, H, bs' \vdash_{\mathrm{I}} autinits \Downarrow sts_0 \\ \mathrm{fby}\ sts_0\ sts_1 \equiv sts \qquad \forall i,\ \forall k,\ G, (\mathrm{select}^{C_i,k}_{0}\ sts\ (H, bs)), C_i \vdash_{\mathrm{W}} autscope_i \Downarrow (\mathrm{select}^{C_i,k}_{0}\ sts\ sts_1) \qquad \star \end{array}}{G, H, bs \vdash \texttt{automaton initially}\ autinits^{ck}\ [\texttt{state}\ C_i\ autscope_i]^i\ \texttt{end}}$$

$$\frac{\begin{array}{c} \forall x,\ x \in \mathrm{dom}(H') \iff x \in locs \qquad \forall x\ e,\ (\texttt{last}\ x = e) \in locs \implies G, H + H', bs \vdash_{\mathrm{L}} \texttt{last}\ x = e \\ G, H + H', bs \vdash blks \qquad G, H + H', bs, C_i \vdash_{\mathrm{TR}} trans \Downarrow sts \end{array}}{G, H, bs, C_i \vdash_{\mathrm{W}} \texttt{var}\ locs\ \texttt{do}\ blks\ \texttt{until}\ trans \Downarrow sts}$$

$$\frac{\begin{array}{c} H, bs \vdash ck \Downarrow bs' \qquad \mathrm{fby}\ (\mathrm{const}\ bs'\ (C, \mathrm{F}))\ sts_1 \equiv sts \\ \forall i,\ \forall k,\ G, (\mathrm{select}^{C_i,k}_{0}\ sts\ (H, bs)), C_i \vdash_{\mathrm{TR}} trans_i \Downarrow (\mathrm{select}^{C_i,k}_{0}\ sts\ sts_1) \\ \forall i,\ \forall k,\ G, (\mathrm{select}^{C_i,k}_{0}\ sts_1\ (H, bs)) \vdash blks_i \qquad \star \end{array}}{G, H, bs \vdash \texttt{automaton initially}\ C^{ck}\ [\texttt{state}\ C_i\ \texttt{do}\ blks_i\ \texttt{unless}\ trans_i]^i\ \texttt{end}}$$

(b) Main rules

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \\ \text{bools-of}\ ys \equiv bs' \qquad G, H, bs \vdash_{\mathrm{I}} autinits \Downarrow sts \\ sts' \equiv \text{first-of}^{C}_{\mathrm{F}}\ bs'\ sts \end{array}}{G, H, bs \vdash_{\mathrm{I}} C\ \texttt{if}\ e; autinits \Downarrow sts'}$$

$$\frac{sts \equiv \mathrm{const}\ bs\ (C, \mathrm{F})}{G, H, bs \vdash_{\mathrm{I}} \texttt{otherwise}\ C \Downarrow sts}$$

(c) Initial state

$$\begin{aligned}
\text{first-of}^{C}_{r}\ (\mathrm{T} \cdot bs)\ (st \cdot sts) &\equiv \langle C, r\rangle \cdot \text{first-of}^{C}_{r}\ bs\ sts\\
\text{first-of}^{C}_{r}\ (\mathrm{F} \cdot bs)\ (st \cdot sts) &\equiv st \cdot \text{first-of}^{C}_{r}\ bs\ sts
\end{aligned}$$

(d) first-of function

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \\ \text{bools-of}\ ys \equiv bs' \qquad G, H, bs, C_i \vdash_{\mathrm{TR}} trans \Downarrow sts \\ sts' \equiv \text{first-of}^{C}_{\mathrm{F}}\ bs'\ sts \end{array}}{G, H, bs, C_i \vdash_{\mathrm{TR}} \texttt{if}\ e\ \texttt{continue}\ C\ trans \Downarrow sts'}$$

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \\ \text{bools-of}\ ys \equiv bs' \qquad G, H, bs, C_i \vdash_{\mathrm{TR}} trans \Downarrow sts \\ sts' \equiv \text{first-of}^{C}_{\mathrm{T}}\ bs'\ sts \end{array}}{G, H, bs, C_i \vdash_{\mathrm{TR}} \texttt{if}\ e\ \texttt{then}\ C\ trans \Downarrow sts'}$$

$$\frac{sts \equiv \mathrm{const}\ bs\ (C_i, \mathrm{F})}{G, H, bs, C_i \vdash_{\mathrm{TR}} \epsilon \Downarrow sts}$$

(e) Transitions

Fig. 13. Semantics of state machines

that they are parameterized by the current state C_i, which is the default next state, and that the transition type determines the value of the reset flag.

As for the `switch`, the ★ premise from section 2.3, with when replaced appropriately by select, is needed to complete partial definitions.

Strong state machines. The rule for strong state machines, at bottom in figure 13b, differs from those for weak state machines in three ways. First, since only a single initial state is declared, the initial-state stream is simply defined with const. Second, state-local variables cannot be used in strong transition guards so there is no need for a special treatment of local scopes. Third, and most

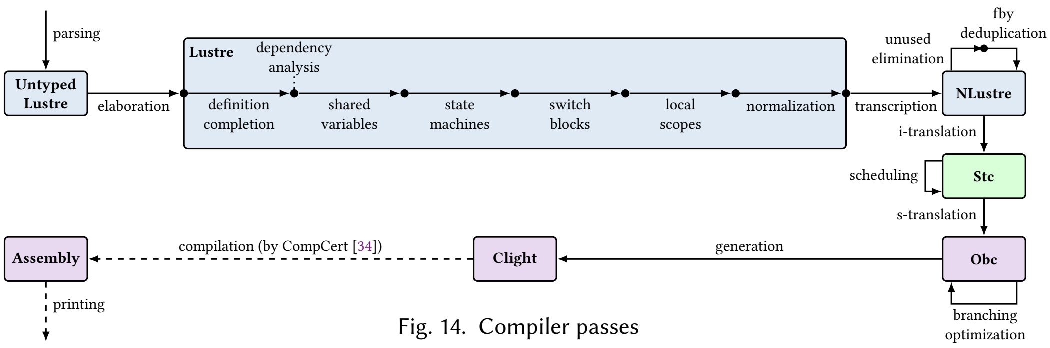


Fig. 14. Compiler passes

importantly, sts now specifies an entry-state stream whose transitions determine sts_1; and sts_1 now specifies the active-state stream which is used instead of sts in the select that combines constraints on H from the instances of $blks_i$. This models the behavior of strong transitions: they are checked first to determine the active state which subsequently becomes the entry state at the next instant.

Allowing both weak and strong transitions. The original proposition for state machines [20, 22] allows mixing weak and strong transitions in a single state machine. Weak transitions determine the next entry state and strong transitions determine the active state. It is possible to weakly enter a state only to exit it strongly the very next instant. Besides the difficulty of understanding and explaining this behavior, it is not clear whether the ephemeral state should be reset, and doing so complicates the semantic model and compilation scheme. Scade 6 avoids this problem by using a dynamic check that only allows at most one transition to fire per cycle [23, §V.C.1]. Lucid synchrone statically rejects state machines where a weak transition enters a state with strong transitions.

Rather than further complicate the semantic rules and compilation scheme, and following the example of other modern languages [14], we prefer to simply forbid mixing weak and strong transitions in the same state machine. The ability to conditionally specify the initial states of weak state machines is intended to compensate for the lost expressivity.

3 VERIFIED COMPILATION

The Vélus compiler is organized as a succession of passes, see figure 14, that transform the program, first to a normalized form (NLustre), then to transition systems (Stc) to facilitate scheduling, then to a simple imperative language (Obc) and finally to Clight, an input language of CompCert. These passes and their correctness are described elsewhere [11–13].

Activation blocks are compiled by source-to-source translations that reduce them to more primitive operations [22]. This approach works well in a proof assistant for two reasons. First, it does not require new intermediate languages; each with its own semantics, typing and clock predicates, and dependency analysis. Second, as will become clear, the syntactic transformations are mirrored in the correctness proofs by decompositions and recompositions of predicates.

In the following sections, we present the new source-to-source transformations in the order they are applied within the compiler. Our work shows that the original compilation scheme [22] is readily integrated into a verified compiler. We also note the adaptations that are required. Each transformation is implemented as a distinct recursive function, but we denote them all with the notation $\lfloor S \rfloor_p$, with S a syntactic element and p any additional parameters that are required. In most cases, a function is simply applied recursively to sub-elements. We only present the interesting cases. The complete definitions are available in the Coq source.

$$\frac{\forall x\ ck, \Gamma(x) = ck \implies H(x) \equiv xs \implies H, bs \vdash ck \Downarrow (\text{clock-of } xs)}{\Gamma, bs \vdash_{\text{ck}} H}$$

$$\frac{\begin{array}{c} G(f) = \texttt{node}\ f(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \\ \forall i \in 1 \ldots n,\ H(x_i) \equiv xs_i \qquad \forall j \in 1 \ldots m,\ H(y_j) \equiv ys_j \\ G, H, (\text{base-of } (xs_1, \ldots, xs_n)) \vdash_{\text{ck}} blk \qquad (ins + outs), (\text{base-of } (xs_1, \ldots, xs_n)) \vdash_{\text{ck}} H \end{array}}{G \vdash_{\text{ck}} f(xs_1, \ldots, xs_n) \Downarrow (ys_1, \ldots, ys_m)}$$

Fig. 15. Instrumented Semantic Model (cf. figure 6)

Each compilation pass is accompanied by proofs of type, clock, and semantics preservation. The proofs all proceed by induction on the definitions in a program, then on the syntax of blocks. We outline the invariants and core properties but omit the tedious syntactic properties that are usually also necessary, for example, typing invariants, uniqueness of declared identifiers, etcetera.

Definition completion. We do not detail the compilation pass that completes partial definitions by adding equations `x = last x`. The algorithm and proof do not pose any particular difficulties.

3.1 Dependency Analysis and Instrumented Semantic Model

The correctness of the clock types with respect to the streams in a program is a key property that is exploited in the semantic preservation proofs. If a variable x has clock type ck and, for a history H and base clock bs, a semantics xs, then the interpretation of the clock type should be true when xs is present and false otherwise.

Definition 3.1 (Clock Correctness).

$$\textbf{if} \quad \Gamma(x) = ck \quad \textbf{and} \quad H(x) \equiv xs \quad \textbf{then} \quad H, bs \vdash ck \Downarrow (\text{clock-of } xs)$$
$$\textbf{where} \quad \text{clock-of } (\langle\rangle \cdot xs) \equiv \mathsf{T} \cdot \text{clock-of } xs \quad \textbf{and} \quad \text{clock-of } (\langle x \rangle \cdot xs) \equiv \mathsf{F} \cdot \text{clock-of } xs$$

This property is an explicit constraint in the semantics of the NLustre intermediate language and shown as a property of the source language for programs without dependency cycles [13]. A recursive function is used to reject cyclic programs and otherwise provide an acyclic dependency graph as a witness. We extend this function with support for activation blocks and update the clock correctness proof. The proof follows by induction on the acyclic dependency graph. We must show that clock correctness holds after each compilation pass. Doing so by rerunning the dependency analysis would have a runtime cost and only guarantee that a cycle is detected if inadvertently introduced, but not that this never happens. Maintaining the acyclic graph as a witness is tedious and complicated. Instead, we run the dependency analysis after the definition completion pass, use the witness to show clock correctness, and define new semantic rules that add this property to the existing antecedents at node and local variable declarations. The rule for nodes is presented in figure 15. The correctness proofs show that this instrumented semantic model is preserved and thus that clock correctness is invariant. We do not present the details, they are routine once the instrumented model has been introduced.

3.2 Shared Variables

The *shared variables* transformation eliminates `last` from declarations and expressions. The case for local scopes is presented in figure 16. For every declaration `last x = e`, it introduces a fresh local variable `last$x` defined by a `fby` between the initial value expression e and the original variable x.

$$\begin{aligned}
&\lfloor \texttt{var}\ [\texttt{last}\ x_i = e_i]^i;\ locs\ \texttt{let}\ blks\ \texttt{tel} \rfloor_\sigma \triangleq \\
&\text{let } \sigma' := \{\texttt{last}\ x_i \ \mapsto\ last\$x_i\}^i \text{ in} \\
&\texttt{var}\ [x_i;\ last\$x_i]^i;\ locs\ \texttt{let}\ [last\$x_i = \lfloor e_i \rfloor_{(\sigma' \circ \sigma)}\ \texttt{fby}\ x_i]^i;\ \lfloor blks \rfloor_{(\sigma' \circ \sigma)}\ \texttt{tel}
\end{aligned}$$

Fig. 16. Compiling a last declaration

The transformation is parameterized by a substitution σ that, in the local block case, is updated to replace every `last x` expression with the new `last$x` variable. The transformation and substitution are applied simultaneously. In the original definitions, last declarations are removed together with switch blocks [22, §3.2.1.2]. We found it easier to use a separate pass.

Generation of fresh identifiers. While introducing fresh identifiers is usually trivial in a compiler, it is tedious in the purely functional language of Coq. We extend the semi-monadic, semi-axiomatized method described in [13, §4.1]. This allows us to reason about the freshness and non-duplication of identifiers in correctness proofs while respecting the constraints imposed by CompCert.

Correctness. Unsurprisingly, the treatment of histories is just as important in the correctness proofs as it is in the semantic definitions. For this compilation pass and many of the others, we will express invariants using *history extension modulo substitution*; $H_1 \sqsubseteq_\sigma H_2$ signifies that "H_2 extends H_1 after renaming by σ". When σ is the identity function, we write $H_1 \sqsubseteq H_2$.

Definition 3.2 (History extension modulo substitution).

$$H_1 \sqsubseteq_\sigma H_2 \quad \textbf{iff} \quad \forall x\ vs, H_1(x) \equiv vs \Longrightarrow H_2(\sigma(x)) \equiv vs$$

With this definition, the core of the correctness invariant for the shared variable transformation is stated as follows.

Lemma 3.3 (Correctness invariant for the compilation of `last`).

$$\textbf{if} \quad G, H_1, bs \vdash blk \quad \textbf{and} \quad H_1 \sqsubseteq_\sigma H_2 \quad \textbf{then} \quad G, H_2, bs \vdash \lfloor blk \rfloor_\sigma$$

The invariant relates the semantics of an arbitrary source block, *blk*, to the semantics of the block returned by the compilation of shared variables, $\lfloor blk \rfloor_\sigma$, for an arbitrary substitution, σ. The substitution is initially empty and is enriched whenever the compilation function traverses a local scope containing `last` declarations, as detailed in figure 16. In Coq, the invariant requires twelve additional well-formedness premises, for example, the block is well typed, well clocked, contains no duplicate local declarations, and the typing and semantic environments are coherent with one another, etcetera. These details are tedious to discover and manipulate in inductive proofs, but present no real interest. That said, efforts to manage them are worthwhile because a growing mass of technical details is certainly one of the difficulties of interactively verifying a practical system.

The Coq proof of lemma 3.3 proceeds by induction on *blk*, which generates a subgoal for each case, see figure 4, and relies on a separate lemma for expressions. The interesting case is for local declarations, which, after some manipulation, manifests as the subgoal shown in figure 17.
The goal below the line is the consequent of lemma 3.3 with *blk* instantiated as an arbitrary local scope. The compilation function, figure 16, has been unrolled to expose the new `fby` equations and an arbitrary recursive invocation on *blks*. The updated substitution function, $\sigma' \circ \sigma$, is applied to *blks* and the subexpressions e_i. Above the line, the induction hypothesis `Hind` holds for arbitrary H_1, H_2, and σ, and allows passing from the semantic predicate of the *blk* being compiled, to a semantic predicate on the recursive invocation. The next three hypotheses, `Hdom`, `Hlast`, and `Hblk`, result from the inversion of the first antecedent of lemma 3.3 according to the semantic rule of figure 11.

$$\dfrac{\begin{array}{rl} \texttt{Hind:} & \forall H_1\, H_2\, \sigma,\ G, H_1, bs \vdash blk \implies H_1 \sqsubseteq_\sigma H_2 \implies G, H_2, bs \vdash \lfloor blks \rfloor_\sigma \\ \texttt{Hdom:} & \forall x, x \in \mathrm{dom}(H_1') \iff x \in [\texttt{last}\ x_i = e_i]^i \cup locs \\ \texttt{Hlast:} & \forall x\, e, (\texttt{last}\ x = e) \in [\texttt{last}\ x_i = e_i]^i \cup locs \implies G, H_1 + H_1', bs \vdash_{\texttt{L}} \texttt{last}\ x = e \\ \texttt{Hblk:} & G, H_1 + H_1', bs \vdash blks \\ \texttt{Href:} & H_1 \sqsubseteq_\sigma H_2 \\ \sigma' := & \{\texttt{last}\ x_i \ \mapsto\ last\$x_i\}^i \end{array}}{G, H_2, bs \vdash \texttt{var}\ [x_i;\ last\$x_i]^i;\ locs\ \texttt{let}\ [last\$x_i = \lfloor e_i \rfloor_{(\sigma' \circ \sigma)}\ \texttt{fby}\ x_i]^i;\ \lfloor blks \rfloor_{(\sigma' \circ \sigma)}\ \texttt{tel}}$$

Fig. 17. Subgoal for the correctness of shared variable compilation

$$\dfrac{\dfrac{\text{... apply Hdom.}}{\begin{array}{l}\forall x, x \in \sigma'(H_1') \iff \\ \quad x \in [x_i;\ last\$x_i]^i \cup locs\end{array}} \quad \dfrac{\text{intros * Hin. exfalso. ...}}{\forall x\, e, (\texttt{last}\ x = e) \in [x_i;\ last\$x_i]^i \cup locs \implies \cdots} \quad \dfrac{\dfrac{\begin{array}{c}\text{inversion Hlast.}\\ \text{constructor; auto.}\end{array}}{G, H_2 + \sigma'(H_1'), bs \vdash last\$x_i = \lfloor e_i \rfloor_{(\sigma' \circ \sigma)}\ \texttt{fby}\ x_i} \quad \dfrac{\dfrac{\text{apply Hblk.}}{G, H_1 + H_1', bs \vdash blks} \quad \dfrac{\text{... apply Href.}}{H_1 + H_1' \sqsubseteq_{(\sigma' \circ \sigma)} H_2 + \sigma'(H_2')}\ \ \text{apply Hind.}}{G, H_2 + \sigma'(H_1'), bs \vdash \lfloor blks \rfloor_{(\sigma' \circ \sigma)}}}{G, H_2 + \sigma'(H_1'), bs \vdash [last\$x_i = \lfloor e_i \rfloor_{(\sigma' \circ \sigma)}\ \texttt{fby}\ x_i]^i;\ \lfloor blks \rfloor_{(\sigma' \circ \sigma)}}}{G, H_2, bs \vdash \texttt{var}\ [x_i;\ last\$x_i]^i;\ locs\ \texttt{let}\ [last\$x_i = \lfloor e_i \rfloor_{(\sigma' \circ \sigma)}\ \texttt{fby}\ x_i]^i;\ \lfloor blks \rfloor_{(\sigma' \circ \sigma)}\ \texttt{tel}}$$

Fig. 18. Proof tree for the correctness of shared variable compilation.

This inversion introduces H_1' to map the exposed local variables to their stream values. Finally, the second antecedent of lemma 3.3 is named `Href`.

The structure of the proof is shown as a tree in figure 18. As usual, the proof is built from the goal at the bottom, upward through several branches, each of which ends in a hypothesis or a contradiction. The horizontal lines represent the application of a rule. The first rule applied to the goal is the semantic rule for local scopes, see figure 11, with H' instantiated by $\sigma'(H_1')$ to express the substitution applied by the compilation function. This gives three new goals. The first requires justifying the instantiation of H' with respect to the declared local variables and follows from `Hdom`. The second is a requirement on last declarations and is readily shown by contradiction since the compilation function eliminates these declarations. The third goal is more involved. It requires proving that the new `fby` equations and recursive invocation on *blks* have a semantics under $H_2 + \sigma'(H_1')$. The proof continues over two branches: one for the `fby` equations, the other for the recursive invocation. For the `fby` equations, `Hlast` gives a semantics $G, H_1 + H_1', bs \vdash_{\texttt{L}} \texttt{last}\ x_i = e_i$, and inversion of the corresponding rule from figure 11 gives, in turn, enough predicates to apply the constructor for `fby` from figure 9 and close the first branch. The recursive invocation follows from the induction hypothesis, `Hind`, together with `Hblk` and `Href`.

The proof described here is simpler than those for later passes, but it exhibits features common to all of them. The semantic rules from section 2 are applied forward from the source semantics (`Hdom`, `Hlast`, `Hblk`), manipulated using other lemmas, passed through an induction hypothesis

$$\lfloor \texttt{automaton initially inits } [\texttt{state } C_i \texttt{ var } \mathit{locs} \texttt{ do } \mathit{blks} \texttt{ until } \mathit{trans}]^i \texttt{ end} \rfloor \triangleq$$
$$\begin{array}{l}\texttt{var } x_{st}, x_{res}, x_{nst}, x_{nres} \texttt{ let} \\ \quad (x_{st}, x_{res}) = \lfloor \mathit{inits} \rfloor \texttt{ fby } (x_{nst}, x_{nres}); \\ \quad \texttt{switch } x_{st} \ [C_i \texttt{ do reset var } \mathit{locs} \texttt{ let } \lfloor \mathit{blks} \rfloor; (x_{nst}, x_{nres}) = \lfloor \mathit{trans} \rfloor_{C_i} \texttt{ tel every } x_{res}]^i \texttt{ end} \\ \texttt{tel}\end{array}$$

Fig. 19. Compiling weak state machines

$$\lfloor \texttt{automaton initially C } [\texttt{state } C_i \texttt{ do } \mathit{blks}_i \texttt{ unless } \mathit{trans}_i]^i \texttt{ end} \rfloor \triangleq$$
$$\begin{array}{l}\texttt{var } x_{st}, x_{res}, x_{nst}, x_{nres} \texttt{ let} \\ \quad (x_{nst}, x_{nres}) = (C, \texttt{false}) \texttt{ fby } (x_{st}, x_{res}); \\ \quad \texttt{switch } x_{nst} \ [C_i \texttt{ do reset } (x_{st}, x_{res}) = \lfloor \mathit{trans}_i \rfloor_{C_i} \texttt{ every } x_{nres}]^i \texttt{ end}; \\ \quad \texttt{switch } x_{st} \ [C_i \texttt{ do reset } \lfloor \mathit{blks}_i \rfloor \texttt{ every } x_{res}]^i \texttt{ end} \\ \texttt{tel}\end{array}$$

Fig. 20. Compiling strong state machines

$$\begin{aligned} \lfloor \epsilon \rfloor_{C_d} &\triangleq (C_d, \texttt{false}) \\ \lfloor \texttt{e then C; } \mathit{trans} \rfloor_{C_d} &\triangleq \texttt{if e then (C, true) else } \lfloor \mathit{trans} \rfloor_{C_d} \\ \lfloor \texttt{e continue C; } \mathit{trans} \rfloor_{C_d} &\triangleq \texttt{if e then (C, false) else } \lfloor \mathit{trans} \rfloor_{C_d} \end{aligned}$$

Fig. 21. Compiling transitions

(`Hind`) to justify recursive invocations, and ultimately used to construct the semantics of a rewritten program. The relational form of the semantic rules facilitates this kind of reasoning. In the proof, the compilation function ($\lfloor \cdot \rfloor$) is applied to variables representing arbitrary program fragments (*locs*, *blks*, etcetera) and data structures (σ).

3.3 State Machines

Since we know that each state machine must contain either weak or strong transitions exclusively, we reimplement the original compilation scheme [22] as two simpler cases. Compiling a state machine requires augmenting the global context with a new enumerated type that has a constructor for each state label. This is assumed by the functions presented below.

The compilation of weak state machines is presented in figure 19. Fresh local variables are introduced to represent the state stream, x_{st} and x_{res}, and the next-state stream, x_{nst} and x_{nres}. An equation is introduced to define these variables using `fby` and the result of compiling the list of potential initial states, the definition of which is not shown. The sampling and resetting implicit in state machines is realized by introducing explicit `switch` and `reset` blocks. The `switch` is guarded by x_{st} and has a case for each state. The body of each case is wrapped in a `reset` guarded by x_{res}. It contains the result of recursively compiling the body of the original state and a new equation defining x_{nst} and x_{nres} in terms of a chained if/then/else generated from the transitions by the function shown in figure 21.

The compilation of strong state machines is presented in figure 20. It must generate two `switch` blocks per state machine. The first one reproduces the effect of the strong transitions to determine x_{st} and x_{res}, which are subsequently used in the second one over the translated state bodies.

$$
\begin{array}{l}
\lfloor \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end} \rfloor_{\sigma,ck} \triangleq \\
\texttt{var}\ x_{cond}, \ldots;\ \texttt{let} \\
\quad x_{cond} = \lfloor e \rfloor_{\sigma,ck}; \\
\quad x_j^{C_i} = \sigma(x_j)\ \texttt{when}\ C_i(x_{cond}); \qquad \forall x_j \in \mathsf{Free}(blks_i), \forall C_i \\
\quad \sigma(y_j) = \texttt{merge}\ x_{cond}\ [C_i \Rightarrow y_j^{C_i}]^i;\ \forall y_j \in \mathsf{Def}(blks_i) \\
\quad \lfloor blks_i \rfloor_{\{x_j \mapsto x_j^{C_i}\} \circ \{y_j \mapsto y_j^{C_i}\} \circ \sigma,\ ck\ \texttt{on}\ C_i(x_{cond})} \\
\texttt{tel}
\end{array}
$$

Fig. 22. Compiling switch blocks

$$
\begin{array}{r}
\vdots \\
\lfloor c \rfloor_{\sigma,ck} \triangleq c\ \texttt{when}\ ck \\
\lfloor x \rfloor_{\sigma,ck} \triangleq \sigma(x) \\
\lfloor e\ \texttt{when}\ C(x) \rfloor_{\sigma,ck} \triangleq \lfloor e \rfloor_{\sigma,ck}\ \texttt{when}\ C(\sigma(x))
\end{array}
$$

Fig. 23. Renaming/resampling expressions

Correctness. The significant feature of this compilation pass is the transformation of state machines into switch and reset blocks. The correctness of this transformation depends on the relation between select, when and mask that is stated in lemma 2.2. Essentially, inverting either of the semantic predicates for state machines (figure 13) gives several instances of select, lemma 2.2 transforms them into nestings of mask and when, which are teased apart and used to introduce the semantic predicates for switch (figure 10) and reset (figure 12).

The invariant for the compilation of state machines is presented below; it is agreeably straightforward, as is the proof of correctness for this pass. This is because the semantic definitions are directly mirrored in the source-to-source transformation, and because they are easily decomposed and recomposed according to the recursive definition of the translation function. A global inductive invariant is unnecessary as all reasoning is local. Still, the relative complexity of state machines is only partially resolved by this pass, the switch, reset, and local scopes remain!

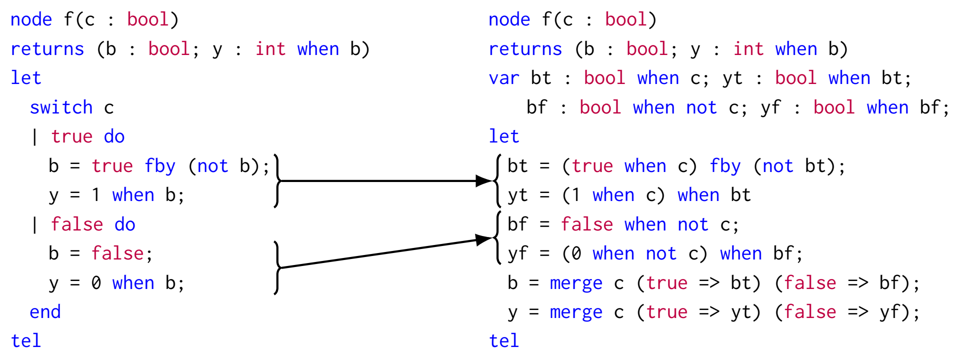
Lemma 3.4 (Correctness invariant for state machine compilation).

$$\textbf{if} \quad G, H, bs \vdash blk \quad \textbf{then} \quad G, H, bs \vdash \lfloor blk \rfloor$$

3.4 Switch Blocks

A switch block is translated using the core sampling operators when and merge [22, §3.2.1.2]. The compilation function is parameterized by a substitution σ for renaming variables and a clock type ck for resampling constants. Compilation, renaming, and resampling thus occur together in a single pass. This complicates certain invariants but satisfies the Coq syntactic criterion for checking function termination. Figure 22 presents the only interesting case of the block compilation function. The incoming σ and ck parameters account for any enclosing switch blocks. For the current switch, we introduce new local variables (i) x_{cond}, for the condition stream, (ii) $x_j^{C_i}$, for any variable x_j that is free (Free) in any block, for each branch label C_i, and (iii) $y_j^{C_i}$, for any variable y_j that is defined (Def) in any block, for each branch label C_i. Each of the new free variables $x_j^{C_i}$ samples the original variable x_j with when according to the condition variable and the appropriate branch label. The defined variables y_j are reconstituted by a merge over their branch-specific versions $y_j^{C_i}$. Finally, the compilation function is applied recursively on each branch, adding substitutions for branch-specific free and defined variables, and a deeper, branch-specific clock type. Figure 23 presents three important cases from the expression compilation function: constants are resampled on the given clock, variables are renamed, and recursively so through when (and other) expressions.

Clock Typing. The correctness of the compilation function exploits the clock-typing rule for switch, figure 10, requiring that all free variables within a branch have the same clock as the condition expression. To see what goes wrong if this rule is not respected, consider the program at left in figure 24. This program violates the clock typing rule because the clock of y (● on true(b))



```
node f(c : bool)
returns (b : bool; y : int when b)
let
  switch c
  | true do
    b = true fby (not b);
    y = 1 when b;
  | false do
    b = false;
    y = 0 when b;
  end
tel
```

```
node f(c : bool)
returns (b : bool; y : int when b)
var bt : bool when c; yt : bool when bt;
    bf : bool when not c; yf : bool when bf;
let
 bt = (true when c) fby (not bt);
 yt = (1 when c) when bt
 bf = false when not c;
 yf = (0 when not c) when bf;
 b = merge c (true => bt) (false => bf);
 y = merge c (true => yt) (false => yf);
tel
```

Fig. 24. Compilation of a malformed program

is slower than that of the condition `c` (•). Compiling this program gives the one shown at right. This program is semantically correct but only because of an invariant involving `b`, `bt`, and `bf` that cannot be expressed by the standard clock typing rules. As it stands, the generated program is not correctly typed because `yt` and `yf` are resampled respectively on `bt = true` and `bf = true` which breaks the clock typing rule for `merge`, figure 8, that requires complementary conditions on the same variable. We reject such programs, as do other compilers [22, 23], and, anyway, it is unusual to mix activation blocks with explicit sampling in real programs.

Simplification and Optimization. The Coq implementation does not actually compute $\mathsf{Free}(blks_i)$. It instead simply samples all variables that are on the same clock as the condition and not defined within the `switch`, whether they are used or not. The `switch` clock typing rule guarantees that this gives a superset of the free variables. This approach simplifies the proof since we do not have to reason about the Free function. The later *unused elimination* pass eliminates variables and equations that are not required to calculate an output variable, whether they are introduced by the compiler or present in the original source program.

Correctness. The inductive proof is structured around the invariant below.

Lemma 3.5 (Correctness invariant for the compilation of `switch` blocks).

$$\textbf{if}\quad G, H_1, bs \vdash blk \quad\textbf{and}\quad H_1, bs' \vdash ck \Downarrow bs \quad\textbf{and}\quad H_1 \sqsubseteq_\sigma H_2 \quad\textbf{then}\quad G, H_2, bs' \vdash \lfloor blk \rfloor_{\sigma,ck}$$

Given (i) a semantics for the source block blk for a history H_1 and base clock bs, (ii) a faster base clock bs' under which a clock ck evaluates to bs, and (iii) a second history H_2 that extends H_1 modulo the substitution σ, then the compiled block has a valid semantics under H_2 and bs'.

For the case of a switch, we know that the source block has a semantics under a history H. To prove that the generated local scope has a semantics, each introduced variable is associated to a stream satisfying the corresponding equation. The variable x_{cond} is associated with a condition stream cs whose correctness follows from a lemma on expressions. For each branch, a sampled free variable $x_j^{C_i}$ is associated with the stream $\mathsf{when}^{C_i}\ cs\ H(x_j)$. The semantics of the new equations for defined variables, the $\sigma(y_j)$, follows from the relation between when and merge stated in lemma 2.1. The semantics for the blocks in branches follows inductively from the invariant, setting H_1 to H, H_2 to H extended with the new variables, σ to the substitution introduced by the compilation function, and ck to the clock for the corresponding branch. Compared to a source block within a branch with base clock bs, the compiled version is activated on the faster base clock of the new context bs'. The slower clock bs corresponds to the clock type given as a parameter of the compilation function.

$$\begin{array}{l} \lfloor \texttt{var}\ \mathit{locs}\ \texttt{let}\ \mathit{blks}\ \texttt{tel}\ \rfloor_{\sigma} \triangleq \\ \textbf{let}\ \sigma' = \{x \mapsto \mathit{local\$x} \mid x \in \mathit{locs}\} \circ \sigma\ \textbf{in} \\ \textbf{let}\ (\mathit{blks}', \mathit{xs}') = \lfloor \mathit{blks} \rfloor_{\sigma'}\ \textbf{in} \\ (\mathit{blks}', \mathit{xs}' \cup \{(\mathit{local\$x} : \lfloor \mathit{ck} \rfloor_{\sigma'}) \mid (x : \mathit{ck}) \in \mathit{locs}\}) \end{array}$$

$$\begin{array}{l} \lfloor \texttt{node ins returns outs blk} \rfloor \triangleq \\ \textbf{let}\ (\mathit{blks}', \mathit{xs}') = \lfloor \mathit{blk} \rfloor_{\mathrm{id}}\ \textbf{in} \\ \texttt{node ins returns outs var}\ \mathit{xs}'\ \texttt{let}\ \mathit{blks}'\ \texttt{tel} \end{array}$$

Fig. 25. Inlining local scopes

3.5 Local Scopes

The compilation function for local scopes must inline arbitrary nestings, whether present in the source program or introduced by the preceding compilation passes, and produce a node that has a single local scope as its root block. Figure 25 presents the main case for blocks, at left, and the case for a node, at right. The function on blocks returns a list of updated blocks and a list of variable declarations, which the function on nodes uses to introduce the single local scope. The function on blocks is parameterized by a substitution σ, which is set initially to the identity function and then updated at each local scope. All variables must be renamed to fresh identifiers to ensure uniqueness, since a given name may be used in different local scopes. The substitution is applied to all expressions and also to clock types in declarations and annotations. The Coq implementation uses the monadic approach described earlier to track the generated variables.

Correctness. The correctness reasoning is similar to that used for the removal of `last` declarations. The proof is based on the following invariant.

Lemma 3.6 (Correctness invariant for the inlining of local scopes).

$$\textbf{if}\quad G, H_1, bs \vdash blk \quad \textbf{and}\quad H_1 \sqsubseteq_{\sigma} H_2 \quad \textbf{then}\quad \exists H_3, H_2 \sqsubseteq H_3 \wedge G, H_3, bs \vdash \lfloor blk \rfloor_{\sigma}$$

The invariant relates three histories. H_1 is the one used to give a semantics to the source block. H_2 gives the effect of applying the substitution σ, it is enriched as each local scope is inlined. The proof must show the existence of a new history, H_3, which may extend H_2, to take account of inlining that happens recursively in *blk*, and under which the generated block has a semantics.

3.6 Reset Blocks

Reset blocks are treated by extending the existing [13] *normalization*, *transcription*, and *translation* passes; see figure 14. At this stage, apart from the node-level local scope, the only cases remaining from figure 4 are those for equations and nested reset blocks.

Normalization. We extend the normalization pass with a new case, see figure 26, that replaces condition expressions with variables and distributes reset blocks over equations. The result is a node containing a single local scope that, in turn, contains a list of normalized equations, each possibly wrapped in `reset` blocks: `reset` $(\cdots($`reset` $(x^+ = e)$ `every` $r_0)\cdots)$ `every` r_n. The expression e is a `fby` expression, a node instance, or an expression containing neither.

It is not possible, in general, to replace the nesting of `reset` blocks with a single one triggered by the disjunction $r_0 \vee \cdots \vee r_n$ since the variables do not necessarily have the same clock. Furthermore, the nested equation may also have a different clock to the variables. This situation arises naturally during the compilation of hierarchical state machines. The key principle is that an equation must be reset even when its clock is false. In terms of compilation, two schemes are commonly used. The first uses an initialization flag to remember the occurrence of a reset and apply it when the equation is next active [30, §4.3]. This delays the work of resetting but requires extra variables and,

$$
\begin{array}{l}
\lfloor \texttt{reset}\ blks\ \texttt{every}\ er \rfloor \triangleq \\
\textbf{let}\ blk_1, \ldots, blk_n = \lfloor blks \rfloor\ \textbf{in} \\
xr = \lfloor er \rfloor; \\
\texttt{reset}\ blk_1\ \texttt{every}\ xr; \\
\ldots \\
\texttt{reset}\ blk_n\ \texttt{every}\ xr;
\end{array}
$$

Fig. 26. Normalization of reset blocks

$$
\begin{aligned}
\lfloor \texttt{reset}\ blk\ \texttt{every}\ xr \rfloor_{xrs} &\triangleq \lfloor blk \rfloor_{(xr::xrs)} \\
\lfloor (xs = rhs) \rfloor_{xrs} &\triangleq (xs = \lfloor rhs \rfloor_{xrs}) \\
\lfloor ce \rfloor_{xrs} &\triangleq \lfloor ce \rfloor \\
\lfloor c\ \texttt{fby}\ e1 \rfloor_{xrs} &\triangleq \texttt{reset}\ c\ \texttt{fby}\ \lfloor e1 \rfloor\ \texttt{every}\ xrs \\
\lfloor f(es) \rfloor_{xrs} &\triangleq (\texttt{reset}\ f\ \texttt{every}\ xrs)(\lfloor es \rfloor) \\
\lfloor (\texttt{reset}\ f\ \texttt{every}\ xr)(es) \rfloor_{xrs} &\triangleq (\texttt{reset}\ f\ \texttt{every}\ (xr :: xrs))(\lfloor es \rfloor)
\end{aligned}
$$

Fig. 27. Transcription with reset blocks

$$
\begin{aligned}
\mathsf{fby}_{\mathsf{NL}}\ v_0\ (\langle\rangle \cdot xs) &\equiv \langle\rangle \cdot \mathsf{fby}_{\mathsf{NL}}\ v_0\ xs \\
\mathsf{fby}_{\mathsf{NL}}\ v_0\ (\langle v\rangle \cdot xs) &\equiv \langle v_0\rangle \cdot \mathsf{fby}_{\mathsf{NL}}\ v\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{reset}^{v_0}_{r_0}\ (\langle\rangle \cdot xs)\ (r \cdot rs) &\equiv \langle\rangle \cdot \mathsf{reset}^{v_0}_{(r_0 \vee r)}\ xs\ rs \\
\mathsf{reset}^{v_0}_{r_0}\ (\langle v\rangle \cdot xs)\ (r \cdot rs) &\equiv \langle \mathsf{if}\ (r_0 \vee r)\ \mathsf{then}\ v_0\ \mathsf{else}\ v\rangle \cdot \mathsf{reset}^{v_0}_{\mathsf{F}}\ xs\ rs
\end{aligned}
$$

Fig. 28. The semantics of fby and reset in NLustre

in our case, extra invariants. The second scheme simply applies the reset immediately. This is the approach taken in the preexisting Vélus compiler and which we generalize in our work.

Transcription. This pass transforms a normalized Lustre program into the NLustre intermediate language, which has three equation forms:

eq ::= x = ce ; | x^+ = reset f (e^+) every [x , ..., x] ; | x = reset c fby e every [x , ..., x] ;

Compared to prior work [12, 13], we generalize the *resettable instance* from a single reset variable to a list of reset variables, and add a *resettable* `fby` that also has a list of reset variables. Figure 27 shows the main cases of the transcription pass. It descends through nested `reset` blocks, accumulating their condition variables in a list. At equations, it transforms the defining expression: for those that do not contain a node instance or `fby`, the reset disjunction is simply dropped; a `fby` expression is replaced by a resettable `fby`; a plain node instance is replaced by a resettable instance; and a resettable instance is extended with the accumulated reset conditions.

In Lustre, the semantics of `e0 fby e1`, figure 9, is given by a partial function since it has two argument streams. In NLustre, however, the semantics of `c fby e` is a total (coinductive) function since the first argument is necessarily a constant. The definition is recalled at the top of figure 28. We make it resettable by composing it with the $\mathsf{reset}^{v_0}_{r_0}\ xs\ rs$ function shown in the same figure. This function encodes the key principle described above. It takes an initial value v_0, a value stream xs, a reset stream rs, and a pending reset r_0 that, like an initialization flag, waits for a subsequent present value on xs. The reset stream rs is derived from the streams of the individual reset variables rs_i by first applying bools-of to each of them, see section 2.5, and then taking the disjunction. That is, if $rs'_i \equiv$ bools-of rs_i, then $rs = \bigvee_i rs'_i$.

Correctness. The transcription correctness invariant is shown below. Given the semantics of a source block relative to a history masked by the disjunction of the boolean streams associated with reset conditions in xrs, the transformed block has the same semantics under the unmasked history.

Lemma 3.7 (Transcription invariant).

$$\textbf{if}\quad (\forall k, G, (\mathsf{mask}^k_0\ (\bigvee_i\ H(xrs_i))\ (H, bs)) \vdash blk) \quad \textbf{then} \quad \lfloor G \rfloor, H, bs \vdash_{\mathsf{NL}} \lfloor blk \rfloor_{xrs}$$

Translation. Our generalizations to NLustre require us to update the *i-translation* pass, the Stc intermediate language, and the *s-translation* pass. Stc is a language based on transition systems. It was introduced [12, §3.1] to allow separate scheduling of the reset and step actions associated with node instances, and thus to permit better optimization of branching in the generated code. Without going into too much detail, we generalize Stc so as to translate a resettable instance into a single *step constraint* and multiple *reset constraints*, one for each reset variable, and similarly for resettable `fby`s. The basic idea continues to work well. The correctness proof of *i-translation* relates the semantics of a single NLustre equation to that of a set of Stc constraints. *Scheduling* places the reset constraints independently according to their clock guards with the aim of increasing the efficacy of the later *branching optimization* on the generated code.

4 EFFICIENCY OF THE GENERATED CODE

An executable compiler is produced from the compilation functions formalized in Coq by extracting them to an OCaml program. Theorem 1.1 assures that the code generated by this compiler is correct, but is it also efficient? For safety-critical embedded software, the relevant measures are the required memory and the estimated Worst Case Execution Time (WCET). The results depend on the compilation scheme, the implementation details, and the backend compiler. Like the Scade KCG [23, figure 4] and academic Heptagon [26, §6] compilers, Vélus implements *clock-directed modular code generation* [9] extended with source-to-source transformations for state machines [22]. The challenge is to implement this scheme in an interactive theorem prover without compromising to simplify formalization and proof. Certain optimizations are best implemented in the Vélus frontend because they require or exploit properties of the intermediate languages. Optimizations that rely on machine-level details are provided by the CompCert backend.

Memory. The compilation scheme guarantees compile-time bounds on the memory required by the generated code: state variables are allocated statically, recursion is not permitted, and a heap is not used. To minimize memory use, a frontend should concentrate on the state variables, since a backend cannot normally optimize them. Local variables will, however, typically be optimized during register allocation in the backend. The interaction between scheduling in the Vélus frontend and register allocation in the CompCert backend has not been optimized, but is anyway independent of the constructions treated in this article. We introduce the *unused elimination* and *fby deduplication* passes, see figure 14, to simplify the compilation and related proofs of, respectively, switch blocks and reset blocks. These passes also remove any redundancies in the source program.

The states in an automaton that are always reset on entry could be allocated overlapping blocks of memory, for example, with a `union` in the generated Clight. This is not possible for states that may be entered with history, that is, via a `continue` transition. We do not implement this optimization. For verified compilation, a mutual-exclusion invariant would need to be expressed on the source language, where it is obvious from the structure, and carried through intermediate passes, where the structure is obscured. The Obc language would need adaptations to express mutually exclusive states, as would the separation logic predicates in the correctness proof for Clight generation.

The size of the generated code mostly depends on the backend, though minimizing the estimated WCET by minimizing branching also reduces the number of jump operations.

WCET. The *scheduling* of guarded equations prior to their translation into Obc, see figure 14, plays a central role in minimizing the estimated WCET of generated code. This pass aims to place equations with similar guards close to one another so that the subsequent *branching optimization* can merge the guards and thereby reduce the number of conditionals [9, §5], and presumably also the estimated WCET. Figure 29 demonstrates this mechanism. It shows extracts of two Obc step functions generated from listing 3. They both calculate the same result, but the one at left

```
class feed_pause {
 method step(pause: bool) returns (enable, step: bool)
 var · · · ; {
  switch state(auto$2) {
   | Feeding => swi$pause$1 := pause
   | Holding => _
   | default => swi$pause$9 := pause
  };
  switch state(norm1$1) {
   | False => skip
   | True => _
   | default => count_up(time).reset()
  };
  time := count_up(time).step(50);
  switch state(auto$2) {
   | Feeding => · · ·
   | Holding => _
   | default => swi$time$12 := time; · · ·
  }; · · ·
```

```
class feed_pause {
 method step(pause: bool) returns (enable, step: bool)
 var · · · ; {
  switch state(norm1$1) {
   | False => skip
   | True => _
   | default => count_up(time).reset()
  };
  time := count_up(time).step(50);
  switch state(auto$2) {
   | Feeding => swi$pause$1 := pause; · · ·
   | Holding => _
   | default =>
      swi$pause$9 := pause;
      swi$time$12 := time;
      · · ·
  }; · · ·
```

Fig. 29. Example of bad (left) and good (right) fusion of conditionals for Obc generated for listing 3

	Vélus	*Hept+CC*	*Hept+gcc*	*Hept+gcci*
stepper_motor (listing 3)	930	1185 (+27 %)	610 (−34 %)	535 (−42 %)
chrono [22, Figure 2]	505	970 (+92 %)	570 (+12 %)	570 (+12 %)
cruisecontrol [40]	1405	1745 (+24 %)	960 (−31 %)	895 (−36 %)
heater [40, §2.2]	2415	3125 (+29 %)	730 (−69 %)	515 (−78 %)
buttons [23, §VI.B]	1015	1430 (+40 %)	625 (−38 %)	625 (−38 %)
stopwatch [23, §V.C]	1305	1970 (+50 %)	1290 (−1 %)	1290 (−1 %)

Fig. 30. Estimated WCET in cycles by OTAWA 2.3.0 [3] for armv7-a using CompCert 3.11 (CC) and GCC 12.2.1 at -O1 without inlining (gcc) and with inlining (gcci); percentages are relative to the *Vélus* column

has more `switch` statements: the schedule places the reset and step calls for the count_up instance between three equations guarded by `state(auto$2)`. At right, the step and reset calls are placed first—no dependencies prevent it,—and the branching optimization is more effective. Incidentally, the underscores next to `switch` labels in figure 29 indicate that the default branch applies. This encoding facilitates the fusion of conditionals and avoids a jump in the generated assembler which may otherwise be required to implement a C switch statement.

The Vélus scheduling pass is implemented in OCaml and its results are validated in Coq. It combines two greedy heuristics: the first uses a tree of queues, based on clocks and `merge`/`case` guards, to find a candidate schedule; the second, taken from Heptagon [26], tries to improve the candidate by pushing similarly clocked equations together while respecting their dependencies.

The details described above are not specific to the verified compilation of state machines. However, it is even more important to treat them thoroughly, even at the cost of more complex invariants and proofs, as block-based constructs in source programs induce branching in generated code. For instance, a 'two-phase' generation scheme, which strictly separates the calculation of current values from state updates, may facilitate some proofs and give reasonable results on basic dataflow programs, but will also tend to duplicate nested control structures across both phases.

We validated our work by compiling (i) an existing set of programs [11, §5], and (ii) the set of programs with activation blocks shown in figure 30. In both cases, we estimated the WCET using OTAWA v2.3.0 [3] and compared with that of code generated using Heptagon 1.05 and different backend C compilers. The estimates for the first set of benchmarks are not shown; they are essentially the same as those reported for the original compiler. The results confirm earlier observations [11, §5] on the importance of backend optimizations, notably that CompCert's inlining heuristic is not fine-tuned to handle the many small functions generated by the frontends. Note, however, that heavily optimizing backends are often avoided in safety-critical applications and that one of CompCert's advantages is to provide optimizations with formal correctness guarantees. For the small set of benchmarks in figure 30, Vélus produces code with a lower estimated WCET than does Heptagon/CompCert. This seems to be due to the scheduling heuristic, the scheduling of reset calls separately from step calls, and the treatment of default switch branches. As the other columns show, however, these features are ultimately outdone by gcc.

At a first approximation, `last` variables provide a convenient but dispensable macro. Their utility is increased by the completion of partial definitions. Here, we eliminate them early by completing with `x = last x` and then replacing `last x` by a fresh variable `last$x`. This simplifies verification, but hinders the elimination of unnecessary writes, `x := last$x`, in the translation to Obc, which would require carrying an additional invariant through the intervening passes. A better approach would be to maintain `last x` expressions down to the Stc language and then to compile `x = last x` directly into `skip`. As usual, semantic preservation proofs would provide a simpler substitute to reasoning about awkward invariants between pairs of variables, for example, between `x` and `last$x`.

5 RELATED WORK

We contribute to a growing body of work on using proof assistants to specify and verify compilers. The CompCert project [10, 34] formalizes a subset of the C language and algorithms to transform it into assembly. CakeML [33] is a verified compiler for an ML-like language in the HOL proof assistant. Vellvm [45] provides a framework for verifying program transformations of the LLVM intermediate representation. This representation respects the Static Single Assignment (SSA) property, as do Lustre programs, but does not offer activation blocks like those presented in this article.

CompCert, CakeML, and Vellvm provide practical compilers for languages with sequential semantics. The theory for such languages, with and without proof assistants, is well developed compared to specialized languages like Lustre and Scade. Research on synchronous languages has always considered mathematical models and their formalization and verification, as in, for instance, Berry's What You Prove Is What You Execute (WYPIWYE) principle for Esterel programs [5, p.10]. We cite the most direct influences on our work throughout the article. That said, the treatment of such languages within a proof assistant is less well studied. Shi *et al.* have also developed a verified Lustre compiler in Coq [42, 43]. Their approach to the semantics differs from that of Vélus and they do not treat activation blocks. Berry and Rieg have developed Coq models for the semantics of Esterel [41]. They focus on the translation to circuits rather than compilation to software.

The state machines of Scade 6 [23], whose key features are treated in this article, are based on a long history. It begins with Statecharts [32], where Harel extends state transition diagrams with hierarchy, concurrency, and communication to facilitate the design and specification of reactive systems. The Esterel [7, 8] and Argos [36, 37] languages provide precise, synchronous semantics for hierarchical state machines. SyncCharts [1] and Mode-Automata [38] also provide synchronous interpretations of extended state machines. Lucid synchrone [17, 28, 40] builds on the principles of Lustre and features from functional programming, namely higher order functions and type inference. Compared to earlier synchronous languages, it introduces clocks expressed as types with inference, and the possibility of freely mixing dataflow equations and hierarchical state machines.

Stateflow [44] provides a Simulink block whose behavior is specified in a visual notation inspired by Statecharts. In contrast to the work described in the previous paragraph, it is most accurately described as a sequential language with a graphical syntax. Hamon formalizes the Stateflow language, initially with Rushby in an operational semantics based on reactions [31], and later in a denotational semantics using continuations [29]. The latter work also formalizes a compilation scheme from Stateflow to imperative code. This work has not been mechanized in a proof assistant.

Our work is based on earlier formal approaches to compiling [22] and specifying [20] state machines in a synchronous dataflow language, and especially on their implementation in Lucid synchrone [40]. Some elements of the idea of defining the semantics 'by translation' [22] are visible in our formalization. For example, in lemma 2.2, the select operator is expressed in terms of mask and when. The correspondence between syntactic and semantic changes facilitates formal reasoning, but there are limits to this approach. Ultimately, the semantic definitions must also provide a reference specification and facilitate proofs about the language and programs. State machines are naturally specified by rules defining how the state evolves from one reaction to the next [20]. We considered such a formalization but it was not obvious how to adapt it to the stream-based style of Vélus, nor clear that it would be a better base for compiler correctness proofs. Another approach is to define the semantics of state machines as a coiterative interpreter [19, 21]. Such models are executable, unlike our relational model, which allows for testing. Conversely, it is not yet clear whether this style is suitable for proving language properties or compiler correctness. A compromise could be to prove that such an interpreter satisfies the semantic predicates presented in this article. One could then use whichever model is most convenient for a given problem.

6 CONCLUSION

This article presents a new relational stream-based semantics for the shared variables, local declarations, switch blocks, reset blocks, and state machines provided by modern synchronous dataflow languages. We have implemented the compilation algorithms for these constructions, with some adaptations, in the Coq proof assistant and developed machine-checked proofs of semantics preservation. The resulting extension of the Vélus compiler gives a working prototype that transforms an expressive programming language designed for embedded control applications to executable assembly code, accompanied by an end-to-end correctness proof.

ACKNOWLEDGMENTS

We thank L. Brun for adding enumerated types to Vélus. This work was supported by the ANR JCJC FidelR 19-CE25-0014 project "Fidelity in Reactive Systems Design and Compilation".

REFERENCES

[1] C. André. 1995. *SyncCharts: A Visual Representation of Reactive Behaviors.* Technical Report. I3S. RR 95-52.

[2] C. André. 1996. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Computational Engineering in Systems Applications.* 19–29.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *SEUS (LNCS, Vol. 6399).* 35–46.

[4] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. 2011. Divide and Recycle: Types and Compilation for a Hybrid Synchronous Language. In *LCTES.* 61–70.

[5] G. Berry. 1989. Real Time Programming: Special Purpose or General Purpose Languages. In *IFIP.* 11–17.

[6] G. Berry. 1993. Preemption in Concurrent Systems. In *Foundations of Software Technology and Theoretical Computer Science (LNCS, Vol. 761).* 72–93.

[7] G. Berry and G. Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152.

[8] G. Berry, S. Moisan, and J.-P. Rigault. 1983. ESTEREL: Towards a synchronous and semantically sound high level language for Real Time Applications. In *RTSS*. 30–37.
[9] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES*. 121–130.
[10] S. Blazy and X. Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *JAR* 43, 3 (2009), 263–288.
[11] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. 2017. A Formally Verified Compiler for Lustre. In *PLDI*. 586–601.
[12] T. Bourke, L. Brun, and M. Pouzet. 2020. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *PACMPL* 4, POPL (2020), 1–29.
[13] T. Bourke, P. Jeanmaire, B. Pesin, and M. Pouzet. 2021. Verified Lustre Normalization with Node Subsampling. *ACM TECS* 20, 5s (2021), Article 98.
[14] T. Bourke and M. Pouzet. 2013. Zélus: A Synchronous Language with ODEs. In *HSCC*. 113–118.
[15] P. Caspi. 1992. Clocks in dataflow languages. *TCS* 94, 1 (1992), 125–140.
[16] P. Caspi. 1994. Towards recursive block diagrams. *Annual Review in Automatic Programming* 18 (1994), 81–85.
[17] P. Caspi, G. Hamon, and M. Pouzet. 2008. Synchronous Functional Programming : The Lucid Synchrone Experiment.
[18] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *POPL*. 178–188.
[19] P. Caspi and M. Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *CMCS (ENTCS, Vol. 11)*.
[20] J.-L. Colaço, G. Hamon, and M. Pouzet. 2006. Mixing Signals and Modes in Synchronous Data-flow Systems. In *EMSOFT*. 73–82.
[21] J.-L. Colaço, M. Mendler, B. Pauget, and M. Pouzet. 2023. A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State Machines. *ACM TECS* same issue (2023).
[22] J.-L. Colaço, B. Pagano, and M. Pouzet. 2005. A Conservative Extension of Synchronous Data-flow with State Machines. In *EMSOFT*. 173–182.
[23] J.-L. Colaço, B. Pagano, and M. Pouzet. 2017. Scade 6: A Formal Language for Embedded Critical Software Development. In *TASE*. 4–15.
[24] J.-L. Colaço and M. Pouzet. 2003. Clocks as First Class Abstract Types. In *EMSOFT (LNCS, Vol. 2855)*. 134–155.
[25] Coq Development Team. 2020. *The Coq proof assistant reference manual*. Inria.
[26] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. 2012. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *LCTES*. 51–60.
[27] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous dataflow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.
[28] G. Hamon. 2002. *Calcul d'horloges et structures de contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML*. Ph.D. Dissertation. Université Pierre et Marie Curie.
[29] G. Hamon. 2005. A Denotational Semantics for Stateflow. In *EMSOFT*. 164–172.
[30] G. Hamon and M. Pouzet. 2000. Modular Resetting of Synchronous Data-Flow Programs. In *PPDP*. 289–300.
[31] G. Hamon and J. Rushby. 2004. An Operational Semantics for Stateflow. In *FASE (LNCS, Vol. 2984)*. 229–243.
[32] D. Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Computer Programming* 8, 3 (1987), 231–274.
[33] R. Kumar, M.O. Myreen, M. Norrish, and S. Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL*. 179–191.
[34] X. Leroy. 2009. Formal verification of a realistic compiler. *Comms. ACM* 52, 7 (2009), 107–115.
[35] X. Leroy. 2009. A formally verified compiler back-end. *JAR* 43, 4 (2009), 363–446.
[36] F. Maraninchi. 1991. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In *Proc. IEEE Workshop on Visual Languages*. 254–259.
[37] F. Maraninchi and N. Halbwachs. 1996. Compiling Argos into Boolean equations. In *FTRTFT (LNCS, Vol. 1135)*. 72–89.
[38] F. Maraninchi and Y. Rémond. 1998. Mode-Automata: About Modes and States for Reactive Systems. In *ESOP (LNCS, Vol. 1381)*. 185–189.
[39] F. Pottier and Y. Régis-Gianas. 2016. *Menhir Reference Manual*. Inria.
[40] M. Pouzet. 2006. *Lucid Synchrone, v. 3. Tutorial and reference manual*. Université Paris-Sud.
[41] L. Rieg and G. Berry. 2022. Towards Coq-verified Esterel Semantics and Compiling. arXiv.
[42] G. Shi, Y. Gan, S. Shang, S. Wang, Y. Dong, and P.-C. Yew. 2017. A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs. In *ICSE-C*. 109–111.
[43] G. Shi, Y. Zhang, S. Shang, S. Wang, Y. Dong, and P.-C. Yew. 2019. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. *Science China Information Sciences* 62, 1 (2019), no. 12801.
[44] The Mathworks 2022. *Stateflow® User's Guide* (10.7 ed.). The Mathworks. Matlab & Simulink R2022b.
[45] J. Zhao, S. Nagarakatte, M.M.K. Martin, and S. Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.* 47, 1 (2012), 427–440.