Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset

TIMOTHY BOURKE, Inria and École normale supérieure – PSL University, France
LÉLIO BRUN, École normale supérieure – PSL University and Inria, France
MARC POUZET, Sorbonne University, École normale supérieure – PSL University, and Inria, France

Specifications based on block diagrams and state machines are used to design control software, especially in the certified development of safety-critical applications. Tools like SCADE Suite and Simulink/Stateflow are equipped with compilers that translate such specifications into executable code. They provide programming languages for composing functions over streams as typified by Dataflow Synchronous Languages like Lustre.

Recent work builds on CompCert to specify and verify a compiler for the core of Lustre in the Coq Interactive Theorem Prover. It formally links the stream-based semantics of the source language to the sequential memory manipulations of generated assembly code. We extend this work to treat a primitive for resetting subsystems. Our contributions include new semantic rules that are suitable for mechanized reasoning, a novel intermediate language for generating optimized code, and proofs of correctness for the associated compilation passes.

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

ACM Reference Format:
Timothy Bourke, Lélio Brun, and Marc Pouzet. 2020. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *Proc. ACM Program. Lang.* 4, POPL, Article 44 (January 2020), 29 pages. https://doi.org/10.1145/3371112

1 INTRODUCTION

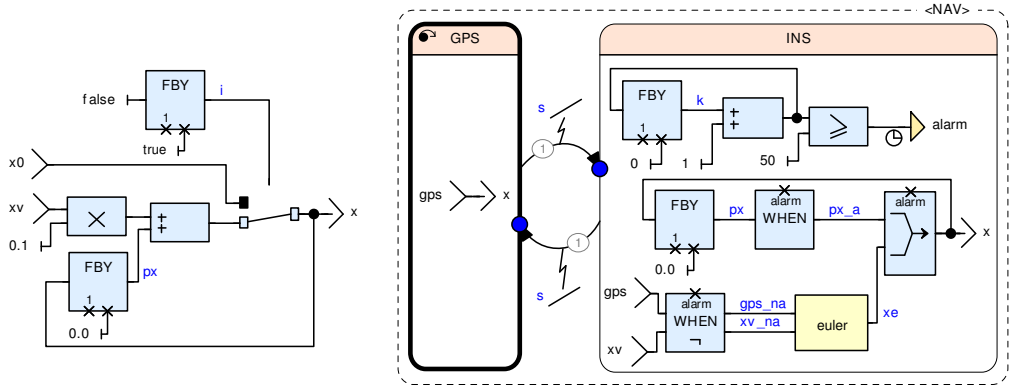
Block-diagram tools like SCADE Suite[1] and Simulink[2] are used to design control software. At their core are dataflow languages: operators apply point-wise to streams, state is encoded by unit delays, and subsystems are abstracted as stream functions. The Lustre synchronous language [Caspi et al. 1987] epitomizes these ideas, but more sophisticated applications require more sophisticated constructs like state machines. State machines can be compiled into primitive constructs [Colaço et al. 2005; Maraninchi and Rémond 2003], foremost among which is the *reset operator* [Hamon and Pouzet 2000], which is a useful primitive in its own right and the subject of this article.

In earlier work, Bourke et al. 2017 describe the specification and verification in the Coq [Coq Development Team 2019] Interactive Theorem Prover (ITP) of a compiler for the dataflow kernel of Lustre. We present its non-trivial extension to treat the reset operator, which permits a subsystem

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

Authors' addresses: Timothy Bourke, Inria, École normale supérieure – PSL University, France, Timothy.Bourke@inria.fr; Lélio Brun, École normale supérieure – PSL University, Inria, France, Lelio.Brun@ens.fr; Marc Pouzet, Sorbonne University, École normale supérieure – PSL University, Inria, France, Marc.Pouzet@ens.fr.

2475-1421/2020/1-ART44
https://doi.org/10.1145/3371112



(a) Euler node (b) State machine

Fig. 1. Graphical representation of the example (screenshot from ANSYS SCADE Suite)

to be restarted in its initial state. We solve two main problems. The first is to reconcile the dataflow semantics of the core constructions with the imperative nature of the reset operator. Building on an earlier idea [Hamon and Pouzet 2000] and preliminary work [Bourke et al. 2018], we show that the existing relational semantics can be extended by adding a single extra rule. The second challenge is to prove that this construct is compiled correctly. To do so we introduce a new intermediate language and associated semantics, adapt a model that specifies the manipulation of internal state, and combine both to show that a new compilation pass is semantics preserving. The overall result is an end-to-end proof showing that the assembly code generated by our compilation passes coupled with those of CompCert [Leroy 2009] respects the extended dataflow semantics. We can thus have the same confidence in the extended language as in the dataflow core. Our work is an important step toward the verification of a code generator for a real language like Scade 6 [Colaço et al. 2017].

The source code and proofs presented in this article have been implemented as extensions to the Vélus compiler. They are available online via https://velus.inria.fr.

1.1 Lustre: Streams and (Synchronous) Stream Functions

Before presenting formal details, we explain and motivate the source language through a toy example inspired by a navigation system. We first program a function, or *node*, that receives two input streams `x0` and `xv`, respectively, an initial position and an instantaneous velocity, and calculates an output stream `x`, representing an approximate position. For simplicity, we calculate with floating-point values whereas a real implementation may use pairs or triples of floating-point or fixed-point values. The node is depicted graphically in fig. 1a and defined as follows.

```
node euler(x0, xv: float64) returns (x: float64);
var i: bool; px: float64;
let
  x = if i then x0 else (px + 0.1 * xv);
  i = true fby false;
  px = 0. fby x;
tel
```

The first line declares the node name and interface. The second declares local variables. The outputs and local variables are defined by equations between `let` and `tel`. The order of equations is inconsequential. The variable `x` is defined by a multiplexer applied pointwise across three argument

streams. Arithmetic operators like + and * are also applied pointwise to their stream arguments. In the equation for i, a fby ("followed-by") operator defines the stream $T, F, F, F, \ldots$, and, similarly, px is defined by a fby operator that tracks the previous value of x. In fact, this node could normally be defined by the single equation `x = x0 -> (pre(x) + 0.01 * xv)`, where -> initializes the first value of the stream and pre is an uninitialized delay,[3] but we have not yet verified the static initialization analysis [Colaço and Pouzet 2004] and normalization pass that are required for these features.[4]

A well-defined node specifies a function from input streams to output streams. Given values for the input variables, the equations determine the values of local and output variables. An example application of euler to two arbitrary input streams is shown in the following table.

x_0	10.00	10.15	10.17	10.16	···
xv	0.50	1.00	0.50	0.40	···
x	10.00	10.10	10.15	10.19	···
i	T	F	F	F	···
px	0.00	10.00	10.10	10.15	···

Once a node is declared, it can be instantiated in other nodes. The following node composes an instance of euler, initialized from a gps input and fed a stream of displacement values, with a counter used to signal an alarm when the resulting odometric approximation is judged outdated.

```
node ins(gps, xv: float64) returns (x: float64; alarm: bool);
var k: int; px: float64; xe: float64 when not alarm;
let
  k = 0 fby (k + 1);
  alarm = (k >= 50);
  xe = euler(gps when not alarm, xv when not alarm);
  x = merge alarm (px when alarm) xe;
  px = 0. fby x;
tel
```

The when operator samples a stream. Here, the arguments of the euler instance are sampled when alarm is F, consequently xe is no longer calculated once the alarm condition occurs. The variable xe is declared with a clock type. Clock types are exploited by code generation [Biernacki et al. 2008], but the details have little bearing on resetting. The x output is defined by a merge operator that combines complementary streams. The effect here is to freeze the value of x when an alarm occurs. The graphical form of these equations can be seen within the INS state of fig. 1b.

Some example values of streams in an instance of ins are shown in the following table.

gps	10.00	10.15	10.17	10.16	···	12.10	12.12	12.18	···
xv	0.50	1.00	0.50	0.40	···	0.80	0.35	0.50	···
k	0	1	2	3	···	49	50	51	···
alarm	F	F	F	F	···	F	T	T	···
xe	10.00	10.10	10.15	10.19	···	11.49			···
x	10.00	10.10	10.15	10.19	···	11.49	11.49	11.49	···
px	0.00	10.00	10.10	10.15	···	11.41	11.49	11.49	···
px when *alarm*					···		11.49	11.49	···

The last line shows the value of `px when alarm`. The gaps in the table show the effect of filtering. They are a distinguishing feature of synchronous languages where streams advance simultaneously

[3] The fby combines initialization and delay: `e0 fby e` defines the same stream as `e0 -> pre e`.

[4] Normalization facilitates imperative code generation by limiting the form of expressions. Section 2.1 defines the normalized form. Graphical tools produce normalized programs. Our semantic models generalize readily for non-normalized programs.

in rounds (the columns of the table). Gaps are represented in semantic definitions by an explicit 'absent' value which cannot be detected or manipulated within the language. Consequently, ignoring the gaps, that is, considering that *px* when *alarm* = 11.49, 11.49, . . ., gives a model consistent with those of Kahn networks [Kahn 1974] or Haskell's lazy streams [Peyton Jones 2003], while taking the gaps into account, that is, considering that *px* when *alarm* = ‹›, . . . , ‹›, ‹11.49›, ‹11.49›, . . ., allows each value, and the corresponding computation, to be associated with a precise instant on a discrete time base. Moreover, the language is restricted to guarantee execution in bounded memory by forbidding recursion and limiting how operators may combine streams [Caspi 1992]: for example, one of the two streams combined by a `merge` must be absent when the other is present. The result is a stream language suitable for programming real-time embedded systems.

Another useful perspective is that Lustre is a language for composing transition systems that communicate by sampling [Caspi and Pouzet 1998]. The `when` and `merge` operators allow for guarded activation with precise control over sampling, and restrictions on causality and clock types ensure determinism. A program denotes a transition system to be executed cyclically to read inputs, update internal state, and write outputs. For each node, compilers produce a *step* function to calculate a transition in bounded time. The triggering of a top-level step function determines the relationship between the logical time of a program and the real time of an implementation.

1.2 A Primitive for Resetting Subsystems

In addition to the primitive dataflow operators, tools like SCADE Suite and Simulink provide control structures for modular programming. For example, adapting the `euler` node so that it can be reset to `x0` using just the primitive operators requires adding a boolean input `r` and redefining `x` as

```
x = if i or r then x0 else (px + 0.1 * xv);
```

This idea does not scale because such inputs must be added to all nodes that may need to be reset, and reset signals must be propagated throughout a program. The problem is exacerbated by state machines because the equations within a state may be reset when a transition occurs. Consider, for example, the node shown in fig. 1b. An `s` input toggles between two modes: *GPS*, where the `x` output is defined directly by the `gps` signal; and *INS*, which replicates the contents of the `ins` node to approximate `x`. When entering a mode, the `fbys` and node instances within must be reinitialized. State machines are compiled by reducing them to more primitive constructs: `fbys` are reinitialized by adding conditionals, but treating node instances in a modular way requires a specific construct.

The reset operator [Hamon and Pouzet 2000] is a primitive for reinitializing node instances. For example, the following equation instantiates the `ins` node to be reinitialized whenever `r` is *T*.

```
x, alarm = (restart ins every r)(gps, xv);
```

The syntax is from Scade 6 [Colaço et al. 2017] and emphasizes the fact that only the node state, and not that of its arguments, is reset. To see this operator's effect, consider the following node.

```
node nat(i : unit) returns (m);
let m = 0 fby (m + 1); tel
```

The input serves only to set the tempo (our formalization currently requires nodes to have at least one input). The node is instantiated with reset on a stream `r` by the following equation.

```
y = (restart nat every r)()
```

An arbitrary value for `r` and the resulting value of `y` are shown in the following table.

r	F	F	F	T	F	F	F	T	F	F	⋯
y	0	1	2	0	1	2	3	0	1	2	⋯

The reset is *strong* [Berry 2000, §3.6] in that it takes effect in the very instant that `r` is *T*.

While the reset operator has an imperative character, it can in fact be expressed in a recursive dataflow program [Caspi 1994, §4.1] [Hamon and Pouzet 2000, §3.1], as in the following pseudocode.

```
node reset_nat(r: bool) returns (y: int);
var c: bool;
let
  c = true -> if r then false else pre c;
  y = merge c (nat(() when c)) (reset_nat(r when not c));
tel
```

The `c` variable is true while `r` is false during which time `y` is defined by a first instance of `nat`. If `r` becomes true, `c` becomes false forever and `y` is defined by a fresh recursive instance of `reset_nat`. Accepting the well-foundedness of the above program and optimizing the tail recursion that it contains requires discovering invariants on `c` that are beyond practical compilers and such programs are rejected. Nevertheless, the intuition of 'a fresh instance at each reset' is central to the semantic model presented in the next section.

Not only is the reset operator convenient for programming and for compiling other control structures, it can be compiled to efficient, modular code by introducing a recursive *reset* function for each node as will be explained in section 3.

While we focus on a subset of Scade 6, the principles studied are fundamental to the dataflow languages that underlie tools for Model-Based Design. Simulink, for instance, provides 'blocks' for *resettable*[5] and *enabled*[6] *subsystems*. In our context, they are simply compositions of ad hoc nodes with reset, sampling, and delay operators, as in the following equations.

```
out1, out2 = (restart resettable_subsystem every rt) (in1, in2);
out3, out4 = merge r ((restart enabled_subsystem every rt) ((in3, in4) when r)) (0.0, 0.0);
rt = (false fby (not r)) and r;
```

A resettable subsystem is specified directly with the reset operator. The node that we name here `resettable_subsystem` is an arbitrary one containing the subsystem to reset whenever `rt` is true. We mimic a Simulink convention by defining `rt` as a 'rising trigger' on `r`. An enabled subsystem only executes when a control signal is positive. This is modeled above by sampling the inputs with `when` and using `merge` to specify the 'output when disabled' value. The possibility to reset 'states when enabling' is expressed with the reset operator. Note that we ignore the subtleties of Simulink's flexible treatment of sample rates [Tripakis et al. 2005, §5.2.4].

2 DATAFLOW SEMANTICS

Formal semantic models have been fundamental in the development of synchronous languages in general, and of Lustre and Scade 6 in particular. Compared to functional and imperative languages, however, less attention has been given to their mechanization within ITPs. As usual, definitions must capture intuitions about the language, act as specifications for compilation (the present focus), and serve as a base for program verification (not considered here). We build on the predicates introduced by Bourke et al. 2017 and present an additional rule to account for the reset operator.

2.1 Abstract Syntax

The abstract syntax of the language treated is structured to express its normalized form. There are two categories of expressions. The first are the basic expressions.

$$e ::= c \mid x^t \mid e \,\mathsf{when}\, [\mathsf{not}]\, x \mid \diamond^t e \mid e \oplus^t e$$

They comprise constants, variables, the sampling operator, unary operators, and binary operators. Types are given by explicit annotations (t) or reconstructed recursively; we denote them by type e.

[5] See https://mathworks.com/help/simulink/slref/resettablesubsystem.html.
[6] See https://mathworks.com/help/simulink/slref/enabledsubsystem.html.

The second category are the control expressions.

$$ce ::= \texttt{merge}\ x\ ce\ ce \mid \texttt{if}\ e\ \texttt{then}\ ce\ \texttt{else}\ ce \mid e$$

They comprise operators for oversampling and multiplexing, and basic expressions.

There are four equation forms: basic equations, instantiations, instantiations with reset, and initialized delays.

$$\begin{aligned} eq ::= {} & x =_{ck} ce \\ \mid {} & x, \ldots, x =_{ck} f(e, \ldots, e) \\ \mid {} & x, \ldots, x =_{ck} (\texttt{restart}\ f\ \texttt{every}\ x^{ck})(e, \ldots, e) \\ \mid {} & x =_{ck} c\ \texttt{fby}\ e \end{aligned}$$

Each equation is annotated with a clock of the following form, where b is true or false.

$$ck ::= \texttt{base} \mid ck\ \texttt{on}\ (x = b)$$

An equation whose value is present whenever the enclosing node is active, that is, whenever at least one input is present, is annotated with base. The on constructor signifies sampling by combining a clock, a variable, and a boolean constant: it is introduced by when and removed by merge. Nodes are modeled as dependent records, though here we elide the well-formedness predicates.

$$node ::= \{\!|\ \texttt{name} = f; \texttt{in} = x^{t,ck}, \ldots, x^{t,ck}; \texttt{var} = x^{t,ck}, \ldots, x^{t,ck}; \texttt{out} = x^{t,ck}, \ldots, x^{t,ck}; \texttt{eqs} = eq, \ldots, eq\ |\!\}$$

Besides the node name, there are disjoint lists of input, local, and output variables with their types and clocks, and a list of equations that define the local and output variables. In the following, we use the standard dot notation to specify the values of record fields, writing, for example, $n.\texttt{name}$ and $n.\mathbf{eqs}$. Bold characters are used consistently for fields and variables that represent lists.

A program, denoted G, is simply a list of uniquely named nodes, where, recursively, the head node may only instantiate those in the tail.

2.2 Expressions

The semantic predicates for expressions are presented in figs. 2 and 3. Each relates a base clock b, a partial mapping of variables to values R, and an expression e to a value v at an instant. An element of type value is either absent, denoted $\langle\rangle$, or present with a value from CompCert [Blazy and Leroy 2009]: a boolean (T or F), a signed or unsigned integer, or a floating-point number.[7]

For basic expressions: constants are present with a value defined by CompCert when the base clock is true and absent otherwise. Variables take the value defined for them by the environment. By convention, a predicate on a partial mapping, for example, $R(x) = v$, is not defined when x is not defined in R (the Coq development asserts that $R(x) = \texttt{Some}\ v$). The when operator transmits the value of the sampled expression or replaces it with the absent value depending on the current value of the variable. It is absent if both the expression and variable are absent. The type system checks that the variable yields a boolean value and the clock system checks that the variable and subexpression are either both present or both absent at an instant. The underlying semantics of unary and binary operators comes from CompCert and depends on the argument types. The operators are not necessarily defined for all values—integer division by zero, for example, is undefined.

The instantaneous values of control expressions are defined similarly. For merge, either the first argument is present and its value determines which of the other two is also present, or all three arguments are absent. For if/then/else the three arguments must be either all present or all absent at an instant. Separate type and clock rules suffice to detect and signal the undefined cases.

The instantaneous semantics of clocks, see fig. 4, defines a semantics for these annotations. The base clock represents the base clock b of the context. If the subclock ck is false then the variable must be absent and the clock is also false. Otherwise, if the subclock is true, then the variable must

[7] Earlier articles, like [Bourke et al. 2017], denote absent values by 'abs' and write lists as '$\overrightarrow{eqn}$'.

$$\frac{}{\mathsf{true}, R \vdash c \downarrow \langle [\![c]\!] \rangle} \qquad \frac{}{\mathsf{false}, R \vdash c \downarrow \langle\rangle} \qquad \frac{R(x) = v}{b, R \vdash x \downarrow v}$$

$$\frac{R(x) = \langle \mathsf{T} \rangle \quad b, R \vdash e \downarrow \langle v \rangle}{b, R \vdash e \ \mathsf{when}\ x \downarrow \langle v \rangle} \qquad \frac{R(x) = \langle \mathsf{F} \rangle \quad b, R \vdash e \downarrow \langle v \rangle}{b, R \vdash e \ \mathsf{when}\ x \downarrow \langle\rangle} \qquad \frac{R(x) = \langle\rangle \quad b, R \vdash e \downarrow \langle\rangle}{b, R \vdash e \ \mathsf{when}\ x \downarrow \langle\rangle}$$

$$\frac{R(x) = \langle \mathsf{F} \rangle \quad b, R \vdash e \downarrow \langle v \rangle}{b, R \vdash e \ \mathsf{when\ not}\ x \downarrow \langle v \rangle} \qquad \frac{R(x) = \langle \mathsf{T} \rangle \quad b, R \vdash e \downarrow \langle v \rangle}{b, R \vdash e \ \mathsf{when\ not}\ x \downarrow \langle\rangle}$$

$$\frac{b, R \vdash e \downarrow \langle v \rangle \quad [\![\diamond, \mathsf{type}\, e]\!]\, v = v'}{b, R \vdash \diamond e \downarrow \langle v' \rangle} \qquad \frac{b, R \vdash e \downarrow \langle\rangle}{b, R \vdash \diamond e \downarrow \langle\rangle} \qquad \frac{b, R \vdash e_1 \downarrow \langle\rangle \quad b, R \vdash e_2 \downarrow \langle\rangle}{b, R \vdash e_1 \oplus e_2 \downarrow \langle\rangle}$$

$$\frac{b, R \vdash e_1 \downarrow \langle v_1 \rangle \quad b, R \vdash e_2 \downarrow \langle v_2 \rangle \quad v_1\, [\![\oplus, \mathsf{type}\, e_1, \mathsf{type}\, e_2]\!]\, v_2 = v}{b, R \vdash e_1 \oplus e_2 \downarrow \langle v \rangle}$$

Fig. 2. Instantaneous semantics of basic expressions

$$\frac{R(x) = \langle \mathsf{T} \rangle \quad b, R \vdash\!: e_t \downarrow \langle v_t \rangle \quad b, R \vdash\!: e_f \downarrow \langle\rangle}{b, R \vdash\!: \mathtt{merge}\ x\ e_t\ e_f \downarrow \langle v_t \rangle} \qquad \frac{R(x) = \langle \mathsf{F} \rangle \quad b, R \vdash\!: e_t \downarrow \langle\rangle \quad b, R \vdash\!: e_f \downarrow \langle v_f \rangle}{b, R \vdash\!: \mathtt{merge}\ x\ e_t\ e_f \downarrow \langle v_f \rangle}$$

$$\frac{R(x) = \langle\rangle \quad b, R \vdash\!: e_t \downarrow \langle\rangle \quad b, R \vdash\!: e_f \downarrow \langle\rangle}{b, R \vdash\!: \mathtt{merge}\ x\ e_t\ e_f \downarrow \langle\rangle} \qquad \frac{b, R \vdash e \downarrow \langle\rangle \quad b, R \vdash\!: e_t \downarrow \langle\rangle \quad b, R \vdash\!: e_f \downarrow \langle\rangle}{b, R \vdash\!: \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \downarrow \langle\rangle}$$

$$\frac{b, R \vdash e \downarrow \langle T \rangle \quad b, R \vdash\!: e_t \downarrow \langle v_t \rangle \quad b, R \vdash\!: e_f \downarrow \langle v_f \rangle}{b, R \vdash\!: \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \downarrow \langle v_t \rangle}$$

$$\frac{b, R \vdash e \downarrow \langle F \rangle \quad b, R \vdash\!: e_t \downarrow \langle v_t \rangle \quad b, R \vdash\!: e_f \downarrow \langle v_f \rangle}{b, R \vdash\!: \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \downarrow \langle v_f \rangle} \qquad \frac{b, R \vdash e \downarrow v}{b, R \vdash\!: e \downarrow v}$$

Fig. 3. Instantaneous semantics of control expressions

$$\frac{}{b, R \vdash \mathsf{base} \downarrow b} \qquad \frac{b, R \vdash ck \downarrow \mathsf{true} \quad R(x) = \langle \mathsf{if}\ b\ \mathsf{then\ T\ else\ F} \rangle}{b, R \vdash ck\ \mathsf{on}\ (x = b) \downarrow \mathsf{true}}$$

$$\frac{b, R \vdash ck \downarrow \mathsf{false} \quad R(x) = \langle\rangle}{b, R \vdash ck\ \mathsf{on}\ (x = b) \downarrow \mathsf{false}} \qquad \frac{b, R \vdash ck \downarrow \mathsf{true} \quad R(x) = \langle \mathsf{if}\ b\ \mathsf{then\ F\ else\ T} \rangle}{b, R \vdash ck\ \mathsf{on}\ (x = b) \downarrow \mathsf{false}}$$

Fig. 4. Instantaneous semantics of clocks

$$\frac{b, R \vdash e \downarrow \langle c \rangle \quad b, R \vdash ck \downarrow \mathsf{true}}{b, R \vdash e :: ck \downarrow \langle c \rangle} \qquad \frac{b, R \vdash e \downarrow \langle\rangle \quad b, R \vdash ck \downarrow \mathsf{false}}{b, R \vdash e :: ck \downarrow \langle\rangle}$$

Fig. 5. Instantaneous semantics of clock-annotated expressions

be present and either T or F, and the constant determines the value of the clock. In the semantic definitions for equations, we use the predicate presented in fig. 5 to enforce a relation between the presence or absence of an expression and the value of an associated clock.

The definitions in figs. 2 to 5 are just the standard rules—see, for example, [Colaço and Pouzet 2003, fig. 2]—defined at a single instant and in a relational style. This style is natural since many combinations of values are not permitted, for instance, three present values in a `merge`, and it is effective for verifying language properties and compilation passes in an ITP.

2.3 Nodes and Equations

We now come to the definitions for nodes and the four types of equations. Simply defining instantaneous rules and iterating them, as we did for expressions, is inadequate for treating the initialized delay operator (`fby`), and thus also for treating nodes and their instantiations. We instead work with streams of values.

There are two natural choices for modeling streams in an ITP: as functions from a natural number to some value, $\texttt{nat} \to \alpha$, whose application we denote xs_i, or using the standard coinductive type $\texttt{Stream}\ \alpha$ with a single constructor. The former admits reasoning by induction on natural numbers which is less technical than reasoning by coinduction in an ITP, whereas the latter often permits simpler definitions closer in style to those of pen-and-paper models; albeit still usually as relations rather than coiterative functions. In fact, we implement both models and prove transfer theorems for passing from one to the other. We focus here on the streams-as-functions model since it is the one used throughout the compiler back end.

The mutually inductive semantic rules for equations and nodes are presented in fig. 6. The definition for nodes is central: in a program G, a node named f relates inputs xs to outputs ys. Both xs and ys have type $\texttt{nat} \to \texttt{list value}$ to facilitate lifting the instantaneous semantics of expressions and verifying the compilation to code that executes cycle-by-cycle. The coinductive model uses the more natural type, `list` (`Stream value`), and we show the isomorphism between the two in the transfer theorems. An isomorphism exists in this case since for any valid xs, we have $H_*(n.\mathbf{in}) = xs$, and thus $\forall i$, length xs_i = length $n.\mathbf{in}$, and similarly for ys.

In the rule for nodes, the antecedents constrain the existence of a base clock bs, a node n, and a *history* H. The base clock is only true whenever at least one input is present, that is,

$$\text{base-of-now}\ v = \text{exists}_{\mathbb{B}}\ (\lambda\, w.\ w \neq_{\mathbb{B}} \langle\rangle)\ v$$
$$\text{base-of}\ vs = \lambda\, i.\ \ \text{base-of-now}\ vs_i$$

The node is found by looking up the name f in G. The history formalizes the kind of table used in sections 1.1 and 1.2 to describe node dynamics. As for the representation of streams, and for similar reasons, H has type $\texttt{nat} \to \texttt{ident} \to \texttt{option value}$: it is a stream of partial mappings from variable names to values. The coinductive model uses the more natural type $\texttt{ident} \to \texttt{option}\ (\texttt{Stream value})$ and the required isomorphisms are shown in the proofs of the transfer theorems. We write H_* to project the stream of values for an identifier, that is, $\forall n, H_*(x)_n = H_n(x)$, provided $H_i(x)$ is always defined; by convention, the projection of a list of identifiers is a stream of lists of values. The ITP tracks these tedious details. The rows of the history corresponding to input and output variables are constrained to match, respectively, xs and ys. Input streams must respect their clocks, that is, a stream is present iff its clock is true:

$$\begin{aligned}\text{respects-clock-now}\ b\ R\ x^{t,ck} = {} & \big(b, R \vdash ck \downarrow \text{true} \leftrightarrow \exists c, R(x) = \langle c \rangle\big) \\ & \wedge \big(b, R \vdash ck \downarrow \text{false} \leftrightarrow R(x) = \langle\rangle\big)\end{aligned}$$
$$\text{respects-clock}\ bs\ H\ x^{t,ck} = \forall i, \text{respects-clock-now}\ bs_i\ H_i\ x^{t,ck}$$

$$\frac{bs = \text{base-of } xs \quad G(f) = n \quad H_*(n.\mathbf{in}) = xs \quad H_*(n.\mathbf{out}) = ys \quad \text{respects-clock } bs\ H\ n.\mathbf{in} \quad \forall eq \in n.\mathbf{eqs},\ G, bs, H \vdash eq}{G \vdash f(xs) \Downarrow ys}$$

$$\frac{bs, H \vdash\!: e :: ck \Downarrow H_*(x)}{G, bs, H \vdash x =_{ck} e} \qquad \frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \quad G \vdash f(vs) \Downarrow H_*(\boldsymbol{x})}{G, bs, H \vdash \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})}$$

$$\frac{bs, H \vdash e :: ck \Downarrow vs \quad H_*(x) \approx \text{fby}\,(\llbracket c \rrbracket, vs)}{G, bs, H \vdash x =_{ck} c \text{ fby } e} \qquad \frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad \text{bools-of } H_*(y)\ rs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \quad \forall k,\ G \vdash f\left(\text{mask}^k_{rs}\ vs\right) \Downarrow \text{mask}^k_{rs}\, H_*(\boldsymbol{x})}{G, bs, H \vdash \boldsymbol{x} =_{ck} \left(\texttt{restart}\ f\ \texttt{every}\ y^{ck_y}\right)(\boldsymbol{e})}$$

Fig. 6. Dataflow semantics: mutually inductive semantics of equations and nodes

Finally, and most importantly, the quantification over $n.\mathbf{eqs}$ ensures that each equation imposes a constraint on H independently of its relative order (we write '$\in$' for both set and list membership). A useful intuition is to start by considering the set of all tables that relate each input variable to the corresponding element of xs, and then for each equation successively, to exclude tables that do not satisfy the imposed constraint, before finally projecting the output variables into ys.

Each of the other four rules in fig. 6 defines the constraint imposed by a given type of equation.

For basic equations, the same stream must be associated with the defining expression e and the defined variable x. The instantaneous semantics of clock-annotated expressions (fig. 5) is lifted in the obvious way: $\forall n,\ bs_n, H_n \vdash\!: e :: ck \downarrow H_*(x)_n$.

For an equation defined by an initialized delay, the value of x must be extensionally equal to the application of the semantic function fby to the value of c given by CompCert and the stream vs associated with the expression e. The semantic function is defined as follows.

$$\begin{aligned}
\text{fby}\,(u, vs)\,n &= \text{if } vs_n = \langle\rangle \text{ then } \langle\rangle \text{ else } \langle\, \text{hold}\,(u, vs, n)\,\rangle \\
\text{hold}\,(u, vs, 0) &= u \\
\text{hold}\,(u, vs, m+1) &= \text{if } vs_m = \langle v \rangle \text{ then } v \text{ else hold}\,(u, vs, m)
\end{aligned}$$

A naive definition that shifts all values by one 'column', $\text{fby}\,(u, vs)\,n = \text{if } (n = 0) \text{ then } u \text{ else } vs_{n-1}$, is insufficient to properly treat absent values. The auxiliary function serves to 'hold' the absent values in place, while the present values 'slide behind' them. The use of a clock-annotated predicate together with the equation's clock, ck, is necessary to properly constrain absent values in x, otherwise an equation like `x = false fby (not x)` could be satisfied, for example, by both $\langle T\rangle, \langle F\rangle, \langle T\rangle, \ldots$, and $\langle\rangle, \langle T\rangle, \langle\rangle, \langle F\rangle, \ldots$, and infinitely many other streams. This is an idiosyncrasy of the normalized language where the first argument must be a constant; when both arguments are expressions, the first determines the overall tempo. We add similar constraints to the other equation rules, even though it is not strictly necessary, to simplify reasoning about them.

For a node instantiation, the list of argument expressions must evaluate to a stream of lists of input values, and the defined variables must be equal to the outputs defined by the rule for nodes. The constraint between the semantics of the clock annotation and the base clock derived from the inputs is similar in spirit to the use of clock-annotated predicates in the two earlier rules.

It turns out that there is a natural generalization of the rule for node instantiations to account for resets. Two ideas are needed. The first comes from the `reset_nat` pseudocode of section 1.2 where a new instance is created whenever a reset occurs. The second is to create all such instances

at the outset and to successively shift the focus from one to another. The formal definition includes the same antecedents as before to evaluate argument expressions and link their values to the clock annotation. It adds the following antecedent to associate the value of y with a stream of booleans.

$$\text{bools-of } xs\ rs = \forall n, \text{if } rs_n \text{ then } (xs_n = \langle T \rangle) \text{ else } (xs_n = \langle F \rangle \vee xs_n = \langle\rangle)$$

In the last antecedent, there is an instance of the rule for nodes for each natural number k, with the following function applied over the stream of input values.

$$\begin{aligned} \text{mask}^k_{rs}\ vs &= \lambda\, n.\ \text{if } (\text{count } rs\ n = k) \text{ then } vs_n \text{ else } (\text{map } (\lambda x.\ \langle\rangle)\ vs_0) \\ \text{count } rs\ n &= \text{if } rs_n \text{ then } c + 1 \text{ else } c \\ &\qquad \text{where } c = (\text{if } n = 0 \text{ then } 0 \text{ else count } rs\ (n-1)) \end{aligned}$$

The count auxiliary keeps track of the instance active at each n by counting the number of resets. The fourth row of fig. 8 shows its value in the earlier example. The mask function for the kth instance transmits vs from the kth true value on rs until just before the $(k+1)$th true value on rs, and is otherwise equal to a list of absent values. This has a similar effect to the `when` operators in the `reset_nat` pseudocode: the kth instance is activated on the kth reset and then constrains the output streams until the $(k+1)$th instance takes over. By construction, when the inputs of a node instance are all absent, so too are its outputs thanks to base-of and the clocking constraints on equations. The masking also ensures that the quantified instances remain synchronized which avoids having to reason through a mapping from reset counts to instant numbers.

This completes the description of the dataflow semantics of normalized Lustre programs. The proposed semantics for node instantiation with reset is modular since the additional rule can be added or removed without modifying any of the other rules from [Bourke et al. 2017]. The new rule is consistent with the constraint-based style which works well for reasoning about compilation correctness in an ITP. Finally, we do not present the details here, but the proposed semantics extends naturally for coinductive streams and for (non-normalized) Lustre.

2.4 Memory Semantics

To verify the translation of normalized Lustre programs into imperative code, Bourke et al. 2017, §3.2, introduce an alternative semantics that specifies a node's behavior in terms of an explicit stream of memory trees M. This *memory semantics* permits proving a lemma that explicitly relates successive values of M to the contents of registers before and after execution of the generated code. This lemma is composed with one that passes from the dataflow semantics to the memory semantics to give a correctness theorem for the translation pass. While the correctness theorem is stated in terms of the dataflow semantics, it is too weak to show directly by induction; the memory semantics provides the formal scaffolding necessary to circumvent this problem.

It turns out that this approach can be extended to treat node instantiations with reset by adapting the idea from the previous section of shifting to a new instance when a reset occurs. In the memory semantics, each instance has its own stream of memory trees. The `fbys` constrain the next memory of a particular instance, but the instances are 'switched' when a reset occurs giving the degree of freedom necessary to represent reinitialized memory. We now present this extended model for the dataflow language before describing, in section 3, its application to translation correctness.

The semantic rules for expressions, figs. 2 to 5, are unchanged. The new mutually inductive rules for nodes and equations are shown in fig. 7. We compare each to the corresponding rule in fig. 6.

The new rule for nodes is parameterized by M of type `nat` ⟶ `memory val`, where `val` is the same as `value` without presence/absence and `memory` α is the following recursive record type.

$$\texttt{memory}\ \alpha = \{\!|\ \texttt{values} : \texttt{ident} \longrightarrow \texttt{option}\ \alpha;\ \texttt{instances} : \texttt{ident} \longrightarrow \texttt{option}\ (\texttt{memory}\ \alpha)\ |\!\}$$

$$\frac{bs = \text{base-of } xs \quad G(f) = n \quad H_*(n.\mathbf{in}) = xs \quad H_*(n.\mathbf{out}) = ys \quad \text{respects-clock } bs\ H\ n.\mathbf{in} \quad \forall eq \in n.\mathbf{eqs},\ G, bs, H, M \vdash eq \quad \text{m-closed}\ M\ n.\mathbf{eqs}}{G, M \vdash f(xs) \Downarrow ys}$$

$$\frac{bs, H \vdash\!\!: e :: ck \Downarrow H_*(x)}{G, bs, H, M \vdash x =_{ck} e} \qquad \frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad bs, H \vdash ck \Downarrow \text{base-of } \boldsymbol{vs} \quad \text{hd}\,\boldsymbol{x} = i \quad G, M_*[i] \vdash f(vs) \Downarrow H_*(\boldsymbol{x})}{G, bs, H, M \vdash \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})}$$

$$\frac{bs, H \vdash e :: ck \Downarrow vs \quad M_0(x) = [\![c]\!] \quad \forall n, \begin{cases} H_n(x) = \langle\rangle \wedge M_{n+1}(x) = M_n(x) & \text{if } vs_n = \langle\rangle \\ H_n(x) = \langle M_n(x)\rangle \wedge M_{n+1}(x) = v & \text{if } vs_n = \langle v\rangle \end{cases}}{G, bs, H, M \vdash x =_{ck} c\ \texttt{fby}\ e}$$

$$\frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad \text{bools-of}\, H_*(y)\ rs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \quad \text{hd}\,\boldsymbol{x} = i \quad \forall k, \exists M^k,\ G, M^k \vdash f\left(\text{mask}^k_{rs}\ vs\right) \Downarrow \text{mask}^k_{rs}\, H_*(\boldsymbol{x}) \wedge \text{m-masked}^k_{rs}\, M_*[i]\ \ M^k}{G, bs, H, M \vdash \boldsymbol{x} =_{ck} \left(\texttt{restart}\ f\ \texttt{every}\ y^{ck_y}\right)(\boldsymbol{e})}$$

Fig. 7. Explicit memory semantics: mutually inductive semantics of equations and nodes

We write $M_n(x)$ to denote the value associated with x in the values environment of M_n, and $M_n[i]$ to denote the (sub-)memory associated with i in the instances environment. As for histories, we add a $*$ to project to a stream. The first five antecedents are the same as those for the dataflow semantics and variables are still constrained by a history H. There is still an antecedent that quantifies over equations, each of which may now also constrain M. The last antecedent requires that for any identifier defined in the values field there is a corresponding `fby` equation, and that any identifier defined in the instances field is the first variable in some node instantiation with or without reset.

$$\begin{aligned}\text{m-closed}\, M\ \boldsymbol{eq} = \forall n,\ &(\forall x,\ (\exists v,\ M_n(x) = v) \longrightarrow (x = \cdot\ \texttt{fby}\ \cdot \in \boldsymbol{eq})) \\ &\wedge (\forall i,\ (\exists M_i,\ M_n[i] = M_i) \longrightarrow (\exists f,\ i :: \cdot = [\texttt{reset}]\ f\ [\texttt{every}\ \cdot]\,(\cdot) \in \boldsymbol{eq}))\end{aligned}$$

This predicate facilitates stating and reasoning about memory equivalence. Without it, an equivalence between two memories would be relative to a domain—the `fby` and instance variables that are constrained,—which changes as one descends into the tree. Including them in the intermediate model captures a property of the system that is exploited and transmitted by later proofs.

The rule for basic equations is essentially as in the dataflow semantics. It constrains H but not M.

Compared to the previous model, the rule for `fby` equations includes the same antecedent on the defining expression, but instead of applying semantic functions to search backward through streams (`fby` and hold), it constrains an element of M from one instant to the next. The main antecedent is shown at right. At any instant n, if vs is absent, then so is the value of x and the memorized value does not change. Otherwise, x is present with the memorized value and the current value of vs is memorized. The remaining antecedent constrains the initial value for x in M. This rule is identical to the one proposed by Bourke et al. 2017.

In the rule for node instantiations, the first two antecedents are the same as in the previous model. We choose to use the first variable defined by the equation to identify the node instance and its associated memory. The rule for nodes is applied as before to define output values and to constrain the instance's (sub-)memories.

The rule for node instantiations with reset shares three antecedents with the dataflow model, and also uses the first defined variable to identify the instance. We use the same idea of shifting

r	F	F	F	T	F	F	F	T	F	F	$\cdots$
y	0	1	2	0	1	2	3	0	1	2	$\cdots$
$M(m)$	0	1	2	3	1	2	3	4	1	2	$\cdots$
count r	0	0	0	1	1	1	1	2	2	2	$\cdots$
$M^0(m)$	0	1	2	3	3	3	3	3	3	3	$\cdots$
$M^1(m)$	0	0	0	0	1	2	3	4	4	4	$\cdots$
$M^2(m)$	0	0	0	0	0	0	0	0	1	2	$\cdots$
$\vdots$											

Fig. 8. Example: instantiation of `nat` with reset on `r` with memories.

to a fresh instance whenever a reset occurs. Each instance k has its own memory M^k, which is constrained exactly as in the dataflow model by a recursive invocation of the rule for nodes. The new element is to combine these local memories into a constraint for the overall memory of the actual instance, that is, to constrain $M_*[i]$. To do this, we generalize the masking idea used for the streams of inputs and outputs by introducing the following predicate.

$$\text{m-masked}^k_{rs}\, M\, N = (\forall n,\ \text{count}\ rs\ n = (\text{if}\ rs_n\ \text{then}\ k+1\ \text{else}\ k) \longrightarrow M_n = N_n)$$

As for the previous definition of mask, the idea is that count determines the active instance based on the number of resets that have occurred. If the kth instance is active at an instant n, then that instance's memory N determines the value of the overall memory M. The only subtlety is that at instants where a reset occurs, M is determined by the previous ($(k-1)$th) instance. Intuitively, a cycle of the generated code begins with the previous state value, resets the memory, and ends after calculating the updated state value for the next instant; this will become clearer in the next section.

The effect of memory masking for the example `y = (restart nat every r)()` is shown in fig. 8. The count r row gives the current instances over time, below it are presented the $M^k(m)$ values for $k = 0$, 1, and 2. The dashed gray lines show the intervals selected by m-masked and recomposed to give the overall value of $M(m)$.

We show that any node with a dataflow semantics also has a memory semantics.

Lemma 2.1. *Given a program G containing a node named f, for any streams xs and ys such that $G \vdash f(xs) \Downarrow ys$, there exists an M such that $G, M \vdash f(xs) \Downarrow ys$.*

The proof follows by induction on G and n.**eqs**, with a case analysis on equations that constructs the M values from the bottom up. The converse is also true.

Lemma 2.2. *Given a program G containing a node named f, for any stream of memories M and streams xs and ys such that $G, M \vdash f(xs) \Downarrow ys$, then $G \vdash f(xs) \Downarrow ys$.*

Another property is important for reasoning about the memory semantics.

Lemma 2.3. *Given G, M, xs and ys such that $G, M \vdash f(xs) \Downarrow ys$, if, for all j strictly less than some n, $xs_j = [\langle\rangle; \cdots; \langle\rangle]$, then $M_n \approx M_0$.*

In other words, the internal state of a node that receives nothing but absent values is observably equivalent to its initial state. Two memories are observationally equivalent if their values fields contain the same elements with the same values, and their instances fields contain the same elements with observationally equivalent values. It is a technical detail due to the representation in Coq we use for partial maps.

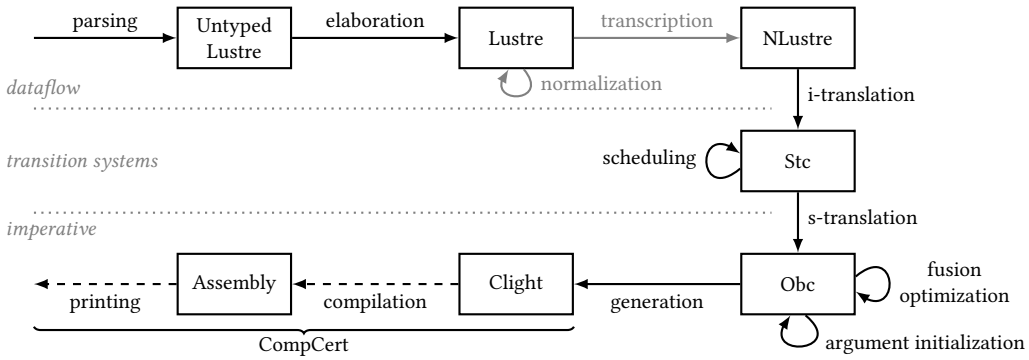


Fig. 9. The architecture of the compiler

3 COMPILATION

Modern compilers for dataflow synchronous languages, like the SCADE Suite KCG code generator [Colaço et al. 2017] and the academic Heptagon [Gérard et al. 2012] and Lustre v6 [Jahier et al. 2019] compilers, follow the scheme outlined by Biernacki et al. 2008: control structures are successively translated into combinations of primitive operators, equations are normalized and scheduled, each node is translated into reset and step methods in a simple object-oriented language, and, finally, imperative functions with explicit state arguments are generated. This scheme was formalized and verified by Bourke et al. 2017 for the dataflow primitives. We build on their work to add support for the reset operator. The basic idea is simple: a node instantiation with reset is translated into a conditional call to a reset method followed by a call to a step method. But generating code where adjacent conditionals are easily optimized away, which is important to reduce estimated Worst-Case Execution Times (WCETs), requires more effort. Unlike Bourke et al. 2017, we divide the translation of dataflow nodes to imperative code into two steps by introducing an intermediate language that exposes instance memories and introduces an independent reset construct without fixing an execution order. This choice enables optimization and permits the expression of a key invariant of the correctness proof.

The new compiler architecture is shown in fig. 9. First, a source file is *parsed* using a verified parser [Jourdan et al. 2012]. Then, *elaboration* adds type and clock annotations and validates the syntactic predicates required by the correctness theorem. At this point a *normalization* pass is usually applied before *transcription* into NLustre, the language presented in section 2.1, but as we have not yet treated these transforms, elaboration currently produces an NLustre program. In the original compiler, the NLustre equations are scheduled before translation directly into Obc, which is further transformed to ensure variable *initialization* and *fuse* adjacent conditionals before Clight is *generated* for *compilation* by CompCert. In our compiler, translation occurs in two steps: *i-translation* introduces explicit instance memories and produces code in the new Stc language, it is this code which is *scheduled*, before *s-translation* sequentializes the program into Obc.

As a concrete example of the basic translation pass, consider the `ins` node from section 1.1 which does not use the reset operator. Both the original and new compilers generate practically identical Obc code; it is shown in fig. 10b. The node has been translated into a class with an instance memory `xe` for the `euler` instance and two memories for the variables defined by `fby` equations. The class has two methods. The `reset` method initializes the two memories with constants from the original program and calls a `reset` method with an explicit instance name (`xe`). The `step` method declares the same inputs and outputs as the source node and a local variable `xe`. Its body comprises a

sequence of assignment statements and a call to the `step` method of `euler` with an explicitly named instance. The method call arguments have been annotated with braces to indicate that they are always defined and do not require explicit initialization [Bourke and Pouzet 2019].[8] The equations have been scheduled so that the variables defined by basic equations and node instantiations are written before being read, and the variables defined by `fby` equations are read before being written. The scheduler tries to group together equations with similar clock annotations and control expressions to permit the fusion of adjacent conditional statements. In this case, the conditional statements introduced by the translation of the `merge` control expression and the clock annotation on the equation defining `xe` have been fused into a single test on `alarm`. Executing the `reset` method and then repeatedly executing the `step` method calculates 'column-by-column' the synchronized streams specified by the dataflow semantics.

As the `ins` example shows, a node instantiation is translated into a single method call. In contrast, the translation of a node instantiation with reset, introduces two (ordered) method calls. Consider, for example, the following two equations.

```
x, alarmx = (restart ins every r)(gps, xv);
y, alarmy = (restart ins every r)(gps, yv);
```

A direct translation to Obc would give the statement below on the left, where each equation is translated into a call to reset followed by a call to step. The result is correct but the fusion optimization is ineffective because the conditional statements are not adjacent. Ultimately, we would like to generate the code on the right, whose execution gives the same result.

```
if (r) { ins(x).reset() };
x, alarmx := ins(x).step({gps}, {xv});
if (r) { ins(y).reset() };
y, alarmy := ins(y).step({gps}, {yv});
```

```
if (r) { ins(x).reset(); ins(y).reset() };
x, alarmx := ins(x).step({gps}, {xv});
y, alarmy := ins(y).step({gps}, {yv});
```

The problem is that Obc is an imperative language; its semantics is defined as a sequence of updates to environments defining the state and local variables. Reordering the assignment statements in Obc or within CompCert and, above all, formally guaranteeing that their semantics is preserved is possible but needlessly difficult, and probably only really useful for the type of code generated by a dataflow synchronous language compiler. On the other hand, reordering equations in NLustre is trivial—the node rules in figs. 6 and 7 are invariant under permutations of the equations—but the reset is indissociable from instantiation. Our solution is to introduce a second intermediate language where resets are explicit but where scheduling is trivial to perform and verify.

3.1 Synchronous Transition Code: Syntax and Semantics

The dataflow semantics of section 2.3 is based on composing stream functions, and consuming and producing streams. In contrast, the intermediate language Stc, for *Synchronous Transition Code*, is based on composing state transitions, and consuming and producing values. Given input values at an instant, a transition imposes constraints between two states, and determines output values. Individual transitions are either composed in sequence, to encode a reset followed by a step, or in parallel, to encode simultaneous constraints. Composite transitions are built-up recursively from transitions on local state memories, mirroring the structure of an original dataflow program. Each composite transition is synchronous: seen from outside, it is a single, atomic constraint between two states. In this model, unlike in the dataflow model, it is not possible to iterate transitions locally to define behaviors over time since this prevents sequential composition in a parent transition, or, put more directly, it precludes external reset. Instead, only complete systems are iterated. The

[8] The streams passed to a node are not necessarily always present when the base clock of the node is true, and thus the variables passed to a method in the generated Obc are not necessarily always defined. As Clight does not permit undefined function arguments, extra assignment statements are introduced when required by the initialization pass.

```
system ins {
 system xe: euler;
 init k: int32 = 0,
      px: float64 = 0.;

 transition (gps: float64, xv: float64)
 returns (x: float64, alarm: bool)
 var xe: float64 when not alarm
 {
  next k = k + 1;
  alarm = (k >= 50);
  xe = euler<xe>(gps when not alarm,
                 xv when not alarm);
  x = merge alarm (px when alarm) xe;
  next px = x;
 }
}
```

(a) Stc result of i-translation

```
class ins {
 instance xe: euler;
 memory k: int32;
 memory px: float64;

 reset () {
   state(k) := 0;
   state(px) := 0.;
   euler(xe).reset()
 }

step (gps: float64, xv: float64)
returns (x: float64, alarm: bool)
var xe: float64
{
  alarm := (state(k) >= 50);
  state(k) := state(k) + 1;
  if (alarm) {
    x := state(px)
  } else {
    xe :=
      euler(xe).step({gps}, {xv});
    x := xe
  };
  state(px) := x
 }
}
```

(b) Obc result of s-translation (and the fusion optimization)

Fig. 10. Example: successive translations of the `ins` node

next section describes the translation of NLustre into Stc, and its verification using the memory semantics introduced in section 2.4.

An example Stc system is shown in fig. 10a. It declares a subsystem named `xe` based on a previous declaration (not shown) of `euler`, and two local states `k` and `px` with types and initial values. Our realization of Stc is not fully general as it only allows two transitions: a reset transition defined implicitly based on initial variable values, and a default transition defined explicitly. This suffices to compile the dataflow language treated in this article and renders the formal presentation easier to digest.[9] The definition of the default transition specifies input, output, and local variables with types and clocks, and otherwise comprises a set of constraints on the local and output variables, and the system's next state, in terms of its current state. As for the dataflow language, the order of constraints is inconsequential and variable values may be present or absent. The syntax of expressions and control expressions is exactly as in section 2.1 and we reuse the semantic definitions of section 2.2.

The example with two equations from the last section can now be translated into the Stc constraints shown below on the left, which are readily rescheduled into the order shown below on the right to ensure their translation into the desired Obc code.

```
x, alarmx = ins<x>(gps, xv);
reset(ins<x>) every r;
y, alarmy = ins<y>(gps, dy);
reset(ins<y>) every r;
```

```
reset(ins<x>) every r;
reset(ins<y>) every r;
x, alarmx = ins<x>(gps, xv);
y, alarmy = ins<y>(gps, dy);
```

The abstract syntax of transition constraints distinguishes the following four forms.

$$
\begin{aligned}
tc ::= \;& x =_{ck} ce \\
\mid \;& \texttt{next}\ x =_{ck} e \\
\mid \;& \texttt{reset}\ f\texttt{<}x\texttt{>}\ \texttt{every}\ ck \\
\mid \;& x, \ldots, x =_{ck} f\texttt{<}x, k\texttt{>}(e, \ldots, e)
\end{aligned}
$$

There are basic constraints, `next` constraints on local state variables, reset transitions, and default transitions. Constraints are annotated with activation clocks, which are left implicit in the concrete syntax since they are easily inferred from variable declarations. The x variables between angle

[9] A generalization to multiple transitions seems natural, but we lack satisfying formalizations of arbitrarily long sequences of transitions and the 'diamond patterns' of orthogonal transitions on a subsystem. The utility of these features is unclear.

$$\frac{b = \text{base-of-now}\ \boldsymbol{xs} \quad P(f) = s \quad R(s.\mathbf{in}) = \boldsymbol{xs} \quad R(s.\mathbf{out}) = \boldsymbol{ys} \quad \text{respects-clock-now}\ b\ R\ s.\mathbf{in} \quad \forall tc \in s.\mathbf{tcs},\ P, b, R, S, I, S' \vdash tc \quad \text{s-closed}\ P\ f\ S \quad \text{s-closed}\ P\ f\ I \quad \text{s-closed}\ P\ f\ S'}{P, S, S' \vdash f\ (\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

$$\frac{b, R \vdash e :: ck \downarrow R(x)}{P, b, R, S, I, S' \vdash x =_{ck} e} \qquad \frac{b, R \vdash \boldsymbol{e} \downarrow \boldsymbol{vs} \quad b, R \vdash ck \downarrow \text{base-of-now}\ \boldsymbol{vs} \quad k = 0 \longrightarrow I[s] \approx S[s] \quad P, I[s], S'[s] \vdash f\ (\boldsymbol{vs}) \Downarrow R(\boldsymbol{x})}{P, b, R, S, I, S' \vdash \boldsymbol{x} =_{ck} f\texttt{<}s, k\texttt{>}\ (\boldsymbol{e})}$$

$$\frac{b, R \vdash e :: ck \downarrow \langle S'(x) \rangle \quad R(x) = \langle S(x) \rangle}{P, b, R, S, I, S' \vdash \texttt{next}\ x =_{ck} e} \qquad \frac{b, R \vdash e :: ck \downarrow \langle\rangle \quad R(x) = \langle\rangle \quad S'(x) = S(x)}{P, b, R, S, I, S' \vdash \texttt{next}\ x =_{ck} e}$$

$$\frac{b, R \vdash ck \downarrow \text{true} \quad \text{initial-state}\ P\ f\ I[i]}{P, b, R, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck} \qquad \frac{b, R \vdash ck \downarrow \text{false} \quad I[i] \approx S[i]}{P, b, R, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

$$\frac{P, S, S' \vdash f\ (xs_n) \Downarrow ys_n \quad P, S' \vdash f\ (xs) \overset{n+1}{\curvearrowright} ys}{P, S \vdash f\ (xs) \overset{n}{\curvearrowright} ys}$$

Fig. 11. Stc: mutually inductive semantics of systems and transition constraints, and their use in a coinductive rule for iterated systems

braces in transition constraints identify a declared subsystem. In a well-formed system, the k variable in a default constraint must equal 1 if the set of constraints also contains a reset transition and 0 otherwise. These variables are omitted in the concrete syntax as they are also easily inferred.

3.1.1 Abstract Syntax. Systems are modeled as dependent records, though here we again elide the well-formedness predicates. We also drop all pretence of supporting multiple transitions.

$$\begin{aligned} \textit{system} ::= \{\!|\ & \texttt{name} = f; \texttt{init} = x^{c,ck}, \ldots, x^{c,ck}; \texttt{sub} = x^{x}, \ldots, x^{x}; \\ & \texttt{in} = x^{t,ck}, \ldots, x^{t,ck}; \texttt{var} = x^{t,ck}, \ldots, x^{t,ck}; \texttt{out} = x^{t,ck}, \ldots, x^{t,ck}; \texttt{tcs} = tc, \ldots, tc\ |\!\} \end{aligned}$$

The `init` field enumerates the local states annotated with initial values and clocks. The `sub` field enumerates the subsystem instance names, each annotated with the name of the defining system. The remaining fields define mutually disjoint lists of input, local, and output variables for the default transition, and the list of its defining transition constraints.

A program, denoted P, is a list of uniquely named systems, where, recursively, the head node may only instantiate those in the tail.

3.1.2 Semantics. Where the memory semantics of section 2.4 constrains a stream of environments and streams of memory trees, the semantics of Stc constrains an environment and memory trees. The rules are shown in fig. 11.

As for the dataflow and memory semantics, the rule for nodes relates inputs to outputs, but here they are both lists of values rather than streams of lists of values. The rule constrains a program P and the memory trees S and S'. Its antecedents require the existence of a base clock value b, a system s, an environment R (replacing the stream of environments H), and an intermediate state I. The intermediate state is a memory tree used as the post-state of reset transitions on subsystems and the pre-state of default transitions on subsystems. The first five antecedents are the same as those in the other models except that they apply at a single instant. A quantification over transition constraints ensures that each has an influence on the environment R, and the memories S, I, and S', and that their relative order is unimportant. The last three antecedents are more technical. The

predicate s-closed expresses the same kind of constraints on the domains of the memory trees S, I and S' as the predicate m-closed described in section 2.4. It ensures that any identifier defined in the values field corresponds to a declared local state, and that any identifier defined in the instances field corresponds to a declared subsystem. Contrary to m-closed, the predicate is recursive on the subsystem instances to facilitate proofs.

$$\begin{aligned}\text{s-closed } P\ f\ S = \exists b,\ & P(f) = b \wedge (\forall x, (\exists v, S(x) = v) \longrightarrow x \in b.\mathbf{init}) \\ & \wedge (\forall i, (\exists S',\ S[i] = S') \longrightarrow (\exists g, i^g \in b.\mathbf{sub} \wedge \text{s-closed } P\ g\ S[i]))\end{aligned}$$

The rule for basic expressions is essentially the same as in the other two models. It only constrains R and not S, I, or S'.

The `next` constraints in Stc replace the `fby` equations of the dataflow language. There are two associated semantic rules. If the defining expression evaluates to a present value, the post-state must map the defined variable to that value, and the environment must map the defined variable to the value from the pre-state. Otherwise, if the defining expression evaluates to an absent value, the defined variable is also absent in the environment and its value in the post-state matches its value in the pre-state. Neither rule constrains the intermediate state I.

There are also two rules for `reset` transitions. When the triggering clock evaluates to true in the current environment, the associated instance memory of the intermediate state is constrained by the following predicate.

$$\begin{aligned}\text{initial-state } P\ f\ S = \exists b,\ & P(f) = b \wedge (\forall x^{c,ck} \in b.\mathbf{init},\ S(x) = [\![c]\!]) \\ & \wedge (\forall i^g \in b.\mathbf{sub},\ \text{initial-state } P\ g\ S[i])\end{aligned}$$

Basically, each local state must be mapped to the value of its declared initial constant and all subsystems are likewise recursively constrained. When the triggering clock evaluates to false, the instance memories in the pre- and intermediate states must be equivalent.

In the rule for default transitions, the first two antecedents are essentially the same as those for node instantiation in the dataflow and memory semantics. In the case where the list of equations does not include a reset transition for the same instance, that is, when $k = 0$, the third antecedent adds an equivalence constraint between elements of I and S.[10] The last antecedent applies the rule for systems to constrain the appropriate instances of the intermediate state, used recursively as the pre-state and the post-state.

The last rule in fig. 11 is defined coinductively relative to a program P and a pre-state S. It constrains the streams xs and ys on the domain $[n, \infty)$ by asserting, for each n, the existence of a state S' constrained by the instantaneous rule for systems.

We show that the semantic model of Stc satisfies the following property.

Lemma 3.1. *Given a program P, S, S', $\boldsymbol{xs}$ and $\boldsymbol{ys}$ such that $P, S, S' \vdash f\ (\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$, if $\boldsymbol{xs} = [\langle\rangle; \cdots; \langle\rangle]$, then $\boldsymbol{ys} = [\langle\rangle; \cdots; \langle\rangle]$ and $S' \approx S$.*

In other words, when all arguments of a system are absent, so are its outputs, and its state does not change. This property is important in the proof of correctness of s-translation because the code generated for a node instance is not called when its inputs are absent.

[10] This may seem quite ad hoc, but a natural generalization is to consider I as a set of memory trees indexed by the sequence numbers labeling transition constraints. A transition labeled with k would then constrain $I_{k-1}[s]$, or $S[s]$ if $k = 0$, and $I_k[s]$. The only complication is to ensure that the last one also constrains $S'[s]$.

$$
\begin{aligned}
\text{i-tr-eq } (x =_{ck} e) &= [x =_{ck} e] \\
\text{i-tr-eq } (x =_{ck} c \texttt{ fby } e) &= [\texttt{next } x =_{ck} e] \\
\text{i-tr-eq } ((i :: \boldsymbol{x}) =_{ck} f\,(\boldsymbol{e})) &= [(i :: \boldsymbol{x}) =_{ck} f\texttt{<}i, 0\texttt{>}\,(\boldsymbol{e})] \\
\text{i-tr-eq } \left((i :: \boldsymbol{x}) =_{ck} \left(\texttt{restart } f \texttt{ every } y^{ck_y}\right)(\boldsymbol{e})\right) &= \left[\texttt{reset } f\texttt{<}i\texttt{>} \texttt{ every } \left(ck_y \texttt{ on } (y = \texttt{true})\right);\right. \\
&\qquad \left.(i :: \boldsymbol{x}) =_{ck} f\texttt{<}i, 1\texttt{>}\,(\boldsymbol{e})\right]
\end{aligned}
$$

Fig. 12. Translation of NLustre equations to Stc transition constraints

3.2 The I-Translation Pass: from NLustre to Stc

3.2.1 Syntactic Transformation. The translation of an NLustre program into Stc is relatively direct. Two major modifications are made: memory and instance names are distinguished, and node instantiations with reset are split into distinct reset and default transition constraints.

Programs are translated node-by-node by a function named i-tr. A node n is translated into a system s with the same name. A declaration is added to s.**init** for each `fby` equation and to s.**sub** for each node instantiation, with or without reset, using the first defined variable as an instance identifier. The n.**in** and n.**out** declarations are transferred, respectively, to s.**in** and s.**out** without modification, and n.**var** is filtered into s.**var** by removing variables defined by `fby` equations. Figure 10a shows the result of translating the `ins` node from section 1.1.

The translation to s.**tcs** is defined as concat-map i-tr-eq (n.**eqs**). The i-tr-eq function is shown in fig. 12. Basic equations are translated directly, as are `fby` equations but for the elision of the initial constant, which has been shifted to the corresponding declaration of x in s.**init**. Node instantiations without reset are translated into default transitions; a well-formedness predicate guarantees the existence of at least one defined variable. Each node instantiation with reset is translated into two transition conditions on the same instance.

3.2.2 Correctness. The correctness of the i-translation pass is expressed and shown relative to the memory semantics of section 2.4 by the following three lemmas.

The first lemma establishes that the initial memory of a node instance in the memory semantics satisfies the initial-state predicate. The proof follows by an easy induction on G.

Lemma 3.2. *Given a program G containing a node named f, and streams xs, M, and ys such that $G, M \vdash f(xs) \Downarrow ys$, then* initial-state (i-tr G) f M_0.

The core of the correctness problem is stated as follows.

Lemma 3.3. *Given a well-clocked program G containing a node named f, and streams xs, M, and ys such that $G, M \vdash f(xs) \Downarrow ys$, then, for all n,* (i-tr G), $M_n, M_{n+1} \vdash f\,(xs_n) \Downarrow\!\Downarrow ys_n$.

The proof proceeds by induction on G. From $G, M \vdash f(xs) \Downarrow ys$, one concludes the existence of a base clock bs, a node n, and a stream of environments H. The proof continues by induction on n.**eqs**. For arbitrary eq, ***tcs***, and I, assuming that eq has a semantics, $G, bs, H, M \vdash eq$, and, by induction, that for the translated equations, $\forall n, tc \in \boldsymbol{tcs}$, (i-tr G), $bs_n, H_n, M_n, I_n, M_{n+1} \vdash tc$, one must exhibit an I' such that $\forall n, tc \in$ (i-tr-eq eq) $+\!\!+$ ***tcs***, (i-tr G), $bs_n, H_n, M_n, I'_n, M_{n+1} \vdash tc$. The proof continues by case analysis on eq. Basic and `fby` equations are trivial: I is unchanged. A node instantiation translates to a default transition with $k = 0$, and thus, for the instance variable i, $\forall n, I'_n = I_n[i \mapsto M_n[i]]$ and the outer induction hypothesis applies. A node instantiation with reset translates into a reset transition and a default transition with $k = 1$; we can reason in the

$$\frac{}{\text{well-sch}^{\boldsymbol{ins}}_{\boldsymbol{mems}}\ [\,]} \qquad \frac{\text{well-sch}^{\boldsymbol{ins}}_{\boldsymbol{mems}}\ \boldsymbol{tcs} \qquad \forall x \in \text{Free}(tc),\ \begin{cases} x \notin \text{Def}(\boldsymbol{tcs}) & \text{if } x \in \boldsymbol{mems} \\ x \in \text{Var}(\boldsymbol{tcs}) \cup \boldsymbol{ins} & \text{if } x \notin \boldsymbol{mems} \end{cases} \qquad \forall i,\ tc = (\texttt{reset}\ \cdot\ \texttt{<}i\texttt{>}\ \texttt{every}\ \cdot) \longrightarrow (\cdot = \cdot\ \texttt{<}i\texttt{,}\ \cdot\texttt{>}\ (\cdot)) \notin \boldsymbol{tcs}}{\text{well-sch}^{\boldsymbol{ins}}_{\boldsymbol{mems}}\ (tc :: \boldsymbol{tcs})}$$

$$\begin{aligned} \text{Var}(x = \cdot) &= \{x\} & \text{Def}(x = \cdot) &= \{x\} \\ \text{Var}(\texttt{next}\ x = \cdot) &= \emptyset & \text{Def}(\texttt{next}\ x = \cdot) &= \{x\} \\ \text{Var}(\texttt{reset}\ \cdot\ \texttt{<}\cdot\texttt{>}\ \texttt{every}\ \cdot) &= \emptyset & \text{Def}(\texttt{reset}\ \cdot\ \texttt{<}\cdot\texttt{>}\ \texttt{every}\ \cdot) &= \emptyset \\ \text{Var}(\boldsymbol{x} = \cdot\ \texttt{<}\cdot\texttt{,}\ \cdot\texttt{>}\ (\cdot)) &= \boldsymbol{x} & \text{Def}(\boldsymbol{x} = \cdot\ \texttt{<}\cdot\texttt{,}\ \cdot\texttt{>}\ (\cdot)) &= \boldsymbol{x} \end{aligned}$$

Fig. 13. Well-scheduled Stc transition constraints for a given set of local states *mems* and list of inputs *ins*.

correctness proof about their combined effect and later reorder them independently. For the instance variable i and reset stream rs, the witness is $\forall n, I'_n = I_n[i \mapsto (\text{if } rs_n \text{ then } M_0 \text{ else } M_n)[i]]$, allowing the application of lemma 3.2 for the case $rs_n = \text{true}$. The outer induction hypothesis again applies for the default transition; lemma 2.3 is needed for $rs_n = \text{true}$ since the memory semantics passes from the kth to the $(k+1)$th reset instance. In this case, the correspondence exploits M^k for the pre-state, M^{k+1} at the same instant for the intermediate state, and M^{k+1} at the next instant for the post-state. For example, consider the fourth instant in fig. 8, the Stc code for the example takes a reset transition from 3 to 0, and a default transition from 0 to 1, only the 3 and the 1 being observable from the outside. The mechanized proof adapts these core arguments and intuitions with some minor technical details.

The following lemma is a direct consequence of the two preceding ones.

Lemma 3.4. *Given a well-clocked program G containing a node named f, and streams xs, M and ys such that $G, M \vdash f(xs) \Downarrow ys$, then* (i-tr G), $M_0 \vdash f\,(xs) \overset{0}{\frown} ys$.

Correctness in terms of the reference dataflow semantics is easily established by combining the previous result and lemma 2.1.

Corollary 3.5. *Given a well-clocked program G containing a node named f, and streams xs and ys, such that $G \vdash f(xs) \Downarrow ys$, there exists a memory tree S_0 such that* initial-state (i-tr G) f S_0 *and* (i-tr G), $S_0 \vdash f\,(xs) \overset{0}{\frown} ys$.

3.3 The S-Translation Pass: from Stc to Obc

The translation from Stc to Obc is all but identical to the scheme described for Lustre by Biernacki et al. 2008 and verified by Bourke et al. 2017. Only minor adaptations are required to treat the distinction between reset and default transitions. We include the definitions for completeness, but focus on the changes to the correctness proof.

3.3.1 Syntactic Transformations. The order of transition constraints in an Stc program is irrelevant to its semantics, but the s-translation pass translates the syntax directly to Obc, which has the effect of replacing constraints by assignment statements and fixing an order of evaluation. It is thus important to first schedule the transition constraints to ensure that the generated code calculates the correct result. Ideally, the schedule should try to group them based on their clock annotations and control expressions to enhance the effect of the subsequent fusion optimization.

We adapted the approach of Bourke et al. 2017 which applies a heuristic implemented in OCaml to find a suitable ordering, a sorting function in Coq whose output is guaranteed to be a permutation

of its input, and a verified translation validator to establish the required well-scheduling predicate or signal an error. The only non-trivial change for Stc is to ensure that reset transitions are executed before corresponding default transitions.

The well-scheduling predicate is shown in fig. 13. It is relative to a list of inputs and a set of local states, both are readily constructed for a system s from, respectively, $s.\mathbf{in}$ and $s.\mathbf{init}$. The transition conditions must be sorted in the reverse order of their realization in the generated code. The empty list is well scheduled. For $tc :: \boldsymbol{tcs}$, $\boldsymbol{tcs}$ must be well scheduled, and for every free variable x in an expression or clock in tc, if x is a local state, then it must not be defined in $\boldsymbol{tcs}$—variables defined by `next` must be read before being written,—otherwise, x must be defined by a basic or default transition constraint, or as an input—other variables must be written before being read. Finally, if tc is a reset transition, then $\boldsymbol{tcs}$ cannot contain a default transition on the same instance.

The s-translation pass maps a function named s-tr-system across the systems in a program. For a system s, it generates the following Obc class.

$$\begin{aligned}&\{\!|\ \texttt{name} = s.\texttt{name};\ \texttt{insts} = s.\mathbf{sub};\ \texttt{mems} = \texttt{map}\ (\lambda\, x^{c}.\ x^{\texttt{ctype}\ c})\ s.\mathbf{init};\\ &\quad \texttt{methods} = [\texttt{reset-method}\ s;\ \texttt{step-method}\ s]\ |\!\}\end{aligned}$$

The class generated for the `ins` example is shown in fig. 10b: it contains instance declarations, memory declarations, and two methods.

Obc method bodies are expressed in an imperative language with expressions and statements.

$$e ::= c \mid x^{t} \mid \texttt{state}(x)^{t} \mid \diamond^{t}\ e \mid e \oplus^{t} e \mid \{e\}$$

An expression is either a constant, a variable, a state variable, a unary operator, a binary operator, or a validity assertion. State variables persist across method calls, variables do not.

$$s ::= \texttt{skip} \mid x := e \mid \texttt{state}(x) := e \mid x, \ldots, x := c(x).m(e, \ldots, e) \mid \texttt{if}\ e\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \mid s\ ;\ s$$

A statement is a no-op, a variable assignment, a state variable assignment, the invocation of a class method for a given instance, a conditional, or a sequential composition. The semantics of expressions and statements is unsurprising. It is formalized as a big step relation between pairs of an environment for variables and a memory tree for state variables and instances. We do not present the formal rules. The coinductive rule for iterating an Obc method is stated as follows.

$$\frac{P, me \vdash cls.m\,(xs_n) \downarrow me', ys_n \qquad P, me' \vdash cls.m\,(xs) \overset{n+1}{\frown} ys}{P, me \vdash cls.m\,(xs) \overset{n}{\frown} ys}$$

It defines the endlessly repeated execution of the method m in the class cls of the program P, starting in a memory me, against a stream of input lists xs to produce a stream of output lists ys, from the nth instant onward. The first antecedent specifies a single execution of the method that updates the memory me to me' and calculates the outputs ys_n. The input and output lists are of type `list` (`option val`). We define the following function to link them to present and absent values.

$$\begin{aligned}\text{toopt}\ \langle\rangle &= \text{None}\\ \text{toopt}\ \langle v \rangle &= \text{Some}\ v\end{aligned}$$

This function is lifted implicitly to lists and streams of lists.

The reset-method translation function generates a method declaration with no input, output, or local variables, and with a body defined by reset-body, which is presented in fig. 14 with the other translation functions. The generated method simply initializes local memories directly and invokes other reset methods to initialize declared instances.

The step-method translation function generates a method declaration with input, output, and local variables taken directly from, respectively, $s.\mathbf{in}$, $s.\mathbf{out}$, and $s.\mathbf{var}$, but without the clock

$$
\begin{aligned}
\text{var } x &= \text{if } x \in \mathit{mems} \text{ then } \texttt{state}\,(x) \text{ else } x \\
\\
\text{ctrl base } s &= s \\
\text{ctrl } (ck \text{ on } (x = \text{true}))\ s &= \text{ctrl } ck\ (\texttt{if } (\text{var } x)\ \{s\}\ \texttt{else}\ \{\texttt{skip}\}) \\
\text{ctrl } (ck \text{ on } (x = \text{false}))\ s &= \text{ctrl } ck\ (\texttt{if } (\text{var } x)\ \{\texttt{skip}\}\ \texttt{else}\ \{s\}) \\
\\
\text{s-tr-exp } c &= c \\
\text{s-tr-exp } x &= \text{var } x \\
\text{s-tr-exp } (e\ \texttt{when}\ x) &= \text{s-tr-exp } e \\
\text{s-tr-exp } (\diamond e) &= \diamond(\text{s-tr-exp } e) \\
\text{s-tr-exp } (e_1 \oplus e_2) &= (\text{s-tr-exp } e_1) \oplus (\text{s-tr-exp } e_2) \\
\\
\text{s-tr-cexp } x\ \left(\texttt{merge}\ y\ e_t\ e_f\right) &= \texttt{if } (\text{var } y)\ \{\text{s-tr-cexp } x\ e_t\}\ \texttt{else}\ \{\text{s-tr-cexp } x\ e_f\} \\
\text{s-tr-cexp } x\ \left(\texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f\right) &= \texttt{if } (\text{s-tr-exp } e)\ \{\text{s-tr-cexp } x\ e_t\}\ \texttt{else}\ \{\text{s-tr-cexp } x\ e_f\} \\
\text{s-tr-cexp } x\ e &= (x := \text{s-tr-exp } e) \\
\\
\text{s-tr-tc}\,(x =_{ck} e) &= \text{ctrl } ck\ (\text{s-tr-cexp } x\ e) \\
\text{s-tr-tc}\,(\texttt{next}\ x =_{ck} e) &= \text{ctrl } ck\ (\texttt{state}\,(x) := \text{s-tr-exp } e) \\
\text{s-tr-tc}\,(\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})) &= \text{ctrl } ck\ (\boldsymbol{x} := f_i.\texttt{step}\,(\text{s-tr-args } ck\ \boldsymbol{e})) \\
\text{s-tr-tc}\,(\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck) &= \text{ctrl } ck\ (f_i.\texttt{reset}\,()) \\
\\
\text{s-tr-tcs } \boldsymbol{tcs} &= \text{foldl } \left(\lambda\, t\ tc.\ \text{s-tr-tc}\,(tc)\ \texttt{;}\ t\right)\ \texttt{skip}\ \ \boldsymbol{tcs} \\
\\
\text{reset-body } s &= \left(\text{foldl } \left(\lambda\, t\ x^{c,\_}.\ t\ \texttt{;}\ \texttt{state}\,(x) := c\right)\ \texttt{skip}\ s.\textbf{init}\right)\ \texttt{;} \\
&\quad\ \left(\text{foldl } \left(\lambda\, t\ i^{f}.\ t\ \texttt{;}\ f_i.\texttt{reset}\,()\right)\ \texttt{skip}\ s.\textbf{sub}\right)
\end{aligned}
$$

Fig. 14. Translation functions from Stc to Obc

declarations. The body is defined by s-tr-tcs s.**tcs**, which translates the transition conditions one-by-one and composes them sequentially. For each transition condition, s-tr-tc introduces nested conditionals according to its clock annotation. A basic transition condition is translated into an assignment statement which may be nested in additional conditionals introduced by the translation of control expressions. A `next` transition condition is translated into a state assignment. Default and reset transitions are translated into calls to, respectively, step and reset methods for the given instance. The translation of expressions is straightforward. The translation of variables tests membership in the set of memories *mems* for the system being translated. The definition of s-tr-args is not presented, it adds validity assertions for variables whose initialization is guaranteed by the correctness proof [Bourke and Pouzet 2019], and otherwise simply invokes s-tr-exp.

After translation to Obc, the existing argument initialization, fusion optimization, and generation passes and their correctness proofs, see fig. 9, apply without modification and we rely on CompCert to produce assembly code that is guaranteed to preserve the semantics of the generated Obc and thus of the NLustre source program.

3.3.2 Correctness. Compared to Bourke et al. 2017 (§3.2), our proof of s-translation correctness need not treat instance introductions, since this is done in the preceding pass, but new invariants are required to handle the intermediate states between reset and default transitions.

In the following, we fix a set of state variables *mems* and a list of inputs ***ins*** for the translation of an Stc system to an Obc class.

The correctness proofs are structured around two key invariants. The first compares an Stc variable environment R with an Obc memory tree *me* and variable environment *ve*.

$$R \overset{xs}{\Leftrightarrow} (me, ve) = \forall x \in \boldsymbol{xs},\quad R(x) = \langle v \rangle \wedge \begin{cases} me(x) = v & \text{if } x \in mems \\ ve(x) = v & \text{if } x \notin mems \end{cases} \\ \vee\ R(x) = \langle\rangle$$

Every variable x in the list ***xs*** must be defined in R as either present or absent. If it is present with value v, then it is associated with the same value in either *me* or *ve* depending on whether or nor it is compiled as a state variable.

The second invariant relates Stc states, S, I, and S', to an Obc memory.

$$(S, I, S') \overset{tcs}{\Longleftrightarrow} me \ = \ \forall x, me(x) = (\text{if } (\texttt{next } x = \cdot) \in \boldsymbol{tcs} \text{ then } S'(x) \text{ else } S(x)) \\ \wedge \forall i, me[i] \approxeq \begin{cases} S[i] & \text{if } (\texttt{reset } \cdot \texttt{<}i\texttt{>} \texttt{ every } \cdot) \notin \boldsymbol{tcs} \wedge (\cdot = \cdot \texttt{<}i\texttt{,} \cdot \texttt{>} (\cdot)) \notin \boldsymbol{tcs} \\ I[i] & \text{if } (\texttt{reset } \cdot \texttt{<}i\texttt{>} \texttt{ every } \cdot) \in \boldsymbol{tcs} \wedge (\cdot = \cdot \texttt{<}i\texttt{,} \cdot \texttt{>} (\cdot)) \notin \boldsymbol{tcs} \\ S'[i] & \text{if } (\cdot = \cdot \texttt{<}i\texttt{,} \cdot \texttt{>} (\cdot)) \in \boldsymbol{tcs} \end{cases}$$

It is defined relative to a list of transition conditions ***tcs***. The idea is to relate *me* to the Stc specification under the assumption that it results from executing the Obc code generated for the constraints in ***tcs***. A variable x is defined in *me* iff it is defined with the same value in either S', if ***tcs*** contains the corresponding `next` constraint, or S otherwise. Similarly, an instance is defined in *me* iff it is defined with an equivalent value either in S, if ***tcs*** contains neither the corresponding reset or default transition, in I, if ***tcs*** contains the reset transition but not the default transition, or in S', if ***tcs*** contains the default transition.[11]

The mechanized proof is non-trivial, but the main arguments can be summarized by the following four lemmas. The first deals with the correctness of expression translation.

Lemma 3.6. *For an environment R, a list of variables **xs**, a memory environment me, and a value environment ve such that* $R \overset{xs}{\Leftrightarrow} (me, ve)$*, then for any expression e with value v, that is, true,* $R \vdash e \downarrow \langle v \rangle$*, if all the free variables of e are in **xs**, then me, ve* $\vdash$ (s-tr-exp *e*) $\downarrow$ Some *v*.

When the free variables of e have the same values in both the Stc environment and the Obc environments, then an induction and case analysis suffices to show that s-tr-exp is correct.

The second lemma shows key properties of the generated reset method.

Lemma 3.7. *For a program P containing a system named f, and any memory environment me, there exists an me′ such that (a)* (tr-stc *P*), *me* $\vdash$ *f*.`reset` () $\downarrow$ *me′*, [ ], *(b)* initial-state *P f me′*, *and (c) if* s-closed *P f me*, *then* s-closed *P f me′*.

In other words, a generated reset method is always well defined, the updated memory is a valid initial state, and the s-closed invariant is maintained.

The third lemma addresses the core of the problem by showing the correctness of the generated step method when it is called.

Lemma 3.8. *Given a well-clocked program P that contains a system named f; lists of present or absent values **ins** and **outs**, such that at least one element of **ins** is present; and states S and S′, related by the Stc semantics* $P, S, S' \vdash f(\boldsymbol{ins}) \Downarrow \boldsymbol{outs}$*; then for any* $me \approxeq S$*, there exists a me′ such that* (tr-stc *P*), *me* $\vdash$ *f*.`step` (toopt ***ins***) $\downarrow$ *me′*, (toopt ***outs***) *and* $me' \approxeq S'$.

[11] Since equality and equivalence are readily lifted to partial maps, we prefer to elide this detail in the presented formula.

The proof is long and intricate, but essentially involves an induction on P, and then, for an arbitrary system s and its semantics in terms of R, S, I, and S', an induction on $s.\mathbf{tcs}$ to show the two aforementioned invariants. For arbitrary $tc :: \boldsymbol{tcs}$, one assumes $R \overset{\boldsymbol{xs}}{\Leftrightarrow} (me, ve)$ and $(S, I, S') \overset{\boldsymbol{tcs}}{\Longleftrightarrow} me$, where $\boldsymbol{xs}$ lists the variables constrained in $\boldsymbol{tcs}$, and me and ve result from executing s-tr-tcs $\boldsymbol{tcs}$ in an environment containing input values. One must then show that the invariants are preserved for each of the four cases of tc. Lemma 3.6 together with well-sch suffices to establish the correctness of any expressions in tc, lemma 3.7 is used for reset transitions, and the outer inductive hypothesis is used for default transitions. In particular, the proof uses lemma 3.1 when the activation clock of a default transition is false. After showing that $s.\mathbf{tcs}$ preserves both invariants, the first one implies that the calculated outputs match those in R, and the second implies that $me' \approx S'$.

This lemma only applies when at least one input is present, that is, when the base clock is true. The other case is easy: if all inputs are absent, the base clock is false, $S' = S$, and the step method is not invoked.

The final lemma gives the correctness of the top-level system that is executed in a loop, it follows more or less directly from lemmas 3.7 and 3.8.

Theorem 3.9. *For a well-clocked P containing a system named f; streams of lists of present or absent values ins and outs, where at least one element of ins_n is present at each instant n; and a state S such that* initial-state $P\ f\ S$ *and* $P, S \vdash f\,(ins) \overset{0}{\curvearrowright} outs$*; there is a memory environment $me \approx S$ such that, for any m,* (tr-stc P), $m \vdash f.\mathtt{reset}\,() \downarrow me, [\,]$, *and* (tr-stc P), $me \vdash f.\mathtt{step}\,(\mathtt{toopt}\ ins) \overset{0}{\curvearrowright} (\mathtt{toopt}\ outs)$.

4 RELATED WORK

4.1 Modeling Explicit Resets

Caspi 1994 shows that recursive block diagrams can express the resetting of stream functions (§4.1). Caspi and Pouzet 1997 add this feature (§A.1) to their dataflow language. These ideas are later adapted and extended by Hamon and Pouzet 2000, where a node instantiation 'acts as a totally new function' whenever its reset signal is true. In their formalization, clocks are modeled as pairs of boolean streams. The first indicates presence, as usual, and the second indicates resetting. The semantics of `fby` takes this 'semantic wire' into account, unlike in our model where resets are treated orthogonally. A reset construct is supported by Lucid Synchrone [Pouzet 2006, §1.6.3], a higher-order synchronous dataflow language.

The idea that resetting a node creates a new instance is exploited by Cohen et al. 2012 for generating parallel code from Lustre: a new thread is launched when a reset occurs (§2.2). The semantics is defined by recursively unrolling instantiations—see the Instantiate rule in their fig. 1. In fig. 6, we define the same construction in the same spirit, but our use of explicit quantification and the mask operator is novel and, we claim, more suited for reasoning in an ITP.

Paulin-Mohring 2009 formalizes Kahn networks in an ITP, but does not treat compilation or the reset construct.

4.2 Verified Compilation of Dataflow Languages

Besides the work of Bourke et al. 2017, on which we build directly, we know of two other ITP-based compilers for block-diagram languages.

Auger 2013 and Auger et al. 2014 verify several passes of a Lustre compiler with reset in Coq. Auger represents streams as lists with the latest value cons-ed onto earlier values (§2.2.4), writing, for example, $((([\,] \cdot 0) \cdot 1) \cdot 2) \cdot 3$. His semantic rules constrain an environment mapping variables to lists. The rule for node instantiations with reset applies a *sequencing operator* (definition 4.2.2),

that has type `list (list val)` × `list value` ⟶ `list value`, and the following (partial) definition.

$$\begin{aligned}
\mathrm{seq}\ ([[\,]], [\,]) &= [\,] \\
\mathrm{seq}\ (\boldsymbol{xs}, \boldsymbol{r} \cdot \langle\rangle) &= \mathrm{seq}\ (\boldsymbol{xs}, \boldsymbol{r}) \cdot \langle\rangle \\
\mathrm{seq}\ (\boldsymbol{xs} \cdot (\boldsymbol{x} \cdot v), \boldsymbol{r} \cdot \langle \mathsf{F} \rangle) &= \mathrm{seq}\ (\boldsymbol{xs} \cdot \boldsymbol{x}, \boldsymbol{r}) \cdot \langle v \rangle \\
\mathrm{seq}\ (\boldsymbol{xs} \cdot ([\,] \cdot v), \boldsymbol{r} \cdot \langle \mathsf{T} \rangle) &= \mathrm{seq}\ (\boldsymbol{xs}, \boldsymbol{r}) \cdot \langle v \rangle
\end{aligned}$$

The reset signal $\boldsymbol{r}$ constrains the number and lengths of finite segments in $\boldsymbol{xs}$ whose values are transmitted one-by-one according to the presence of $\boldsymbol{r}$. Auger's Every rule (fig. 5.7) quantifies over node instances and each contributes a segment to $\boldsymbol{xs}$. By contrast, our model is defined over streams and constraints are patched together by mask operators rather than explicit sequencing. Working with lists engenders uninteresting proof obligations for empty lists and lists of unequal size, and passing from sets of lists to streams is not trivial. We do not know how to extend the sequencing operator to streams, nor how to manipulate it easily in an ITP.

Auger 2013 treats normalization to an intermediate language 'Lsni' (§8) with explicit memories for `fby` equations and node instantiations. The semantics of Lsni is defined at a single instant by constraining a local environment for expressions and two memories for tracking the pre- and post-states of `fby`s and node instances. The rule for node instantiations with reset (fig. 8.8) introduces an intermediate memory that must equal either an initial memory or the existing instance memory according to the reset stream. Our memory semantics rules are similar except that we constrain streams of nodes and memories, and treat resets by patching together individual constraints. Like Lsni, Stc defines one cycle of a system in terms of memory manipulations and uses an intermediate state to account for resets, but it distinguishes reset and default transitions, which enables more optimal scheduling. Finally, unlike us, Auger 2013 does not show end-to-end correctness.

The Lustre compiler of Shi et al. 2017, 2019 is specified in Coq, generates Clight, and many of its compilation passes have been verified. The compiler supports records, arrays, array iterators, and the 3-argument `fby` of Scade 6, but not the reset operator. The semantics is defined sequentially: a node's equations are sorted, at first 'virtually' in the defining predicate and later syntactically by a scheduling pass, and then evaluated in order. By defining an imperative semantics—equations are evaluated sequentially to build an environment,—Shi et al. 2017, 2019 sidestep the problems we solve, but scheduling ('sequentializing') is difficult. We believe they will also find it difficult to treat normalization, state machines, and other higher-level features.

Translation validation is an alternative to verified code generation. Such compilers exist from Simulink to C [Ryabtsev and Strichman 2009] and from Signal to C [Ngo et al. 2015], but the validators themselves are not verified, nor do they treat the reset operator.

Our work complements general-purpose verified compilers like CompCert [Leroy 2009] and CakeML [Kumar et al. 2014]. We focus on the idiosyncrasies of a domain-specific language for embedded reactive systems. While the implementation is based on CompCert and Coq, the models and invariants are more general. It may be possible, for instance, to develop similar front-end passes to target CakeML at the StackLang level [Tan et al. 2016] where garbage collection can be avoided.

4.3 Other Reactive Languages

Statecharts [Harel 1987] is the prototypical reactive language. Of the diverse semantic models, the one used in the Statemate tool [Harel and Naamad 1996, §8] defines an interpretation algorithm that calculates the effect of a step, including the generation and delayed application of variable updates. The state exit, state entry, and *history-clear* actions implement the reset behaviors implicit in a model. We study a different, less expressive model.

Argos formalizes a subset of Statecharts as operators on Boolean Mealy Machines [Maraninchi and Rémond 2001]. Resetting is built into the refinement operator (their definition 7 in terms of states and transitions): when entering a state, the refining machine is started in its initial state, and the refining machine of the previous state is 'killed'. Argos can be translated into equations [Maraninchi and Halbwachs 1996] expressed in the intermediate DC language of guarded assignments, with basic ('EQU') and next transitions ('MEM'), whose semantics does not distinguish absence or treat composite transitions as does Stc. The translation function (§4.2.2) generates 'alive' and 'kill' conditions for refined states which are tested to reset refining equations to their initial states. In other words, the basic idea is to add explicit signals and tests rather than generate and call reset functions.

Mode-Automata [Maraninchi and Rémond 2003] combine Argos and Lustre. They too are translated into the DC intermediate language. In Mode-Automata, all variables are global and maintain their values across mode changes, so refinement (§3.2.2) no longer involves a reset.

The automata of Scade 6 [Colaço et al. 2005] build on ideas from Statecharts, Argos, and Mode-Automata. Their compilation into dataflow primitives exploits node instantiation with reset.

Esterel is a special-purpose language for real-time systems [Berry 1989]. The reset operator is generalized by a loop *P* each *d* construction [Berry 2000, §4.7.11], which is, in turn, defined in terms of loop and abort constructions [Berry 1993]. The semantics of Pure Esterel, a restriction to boolean signals, is defined by a *statement transition* relation [Berry 2002, §6.2] where a single environment applies to all components; that is, the same environment occurs on both sides of a sequential composition. The environment's coherence is ensured by the *constructive causality* of local signal rules [Berry 2002, §7.4]. In comparison, the transition rules for Stc constrain a sequence of distinct environments within a node and force the synchronization of pre- and post-environments, and the hiding of intermediate environments across nodes. The well-sch predicate of section 3.3.1 uses the simpler causality of Lustre that proscribes instantaneous feedback loops. In the translation of Pure Esterel to circuits, the lifetimes of subprocesses are controlled by propagating control wires [Berry 2002, §11]. This translation is formalized in Coq and its correctness verified in unpublished work by G. Berry and L. Rieg. The scheme used to compile Esterel programs to circuits is different to the one used to compile Lustre programs to sequential code. Esterel programs with data are formalized by a micro-step semantics as in standard sequential languages [Potop-Butucaru et al. 2007, §5]. A local variable is simply (re)initialized in the environment when its declaration is (re)entered. Our constraint-based semantic models are thus not directly applicable.

4.4 Simulink and Stateflow

While SCADE Suite is preferred for certified applications, Simulink, together with the state-machine-inspired Stateflow block, is the de facto standard for the Model-Based Design of control software.

There are many partial formalizations of Simulink/Stateflow. Alur et al. 2008 formalize a subset by translation to linear hybrid systems; enabled subsystems are treated but it is not clear whether or how resets are handled. They focus on verification and simulation, not code generation. Chapoutot and Martel 2009 apply abstract interpretation to generate numerical overapproximations; they handle subsystems but not enabled or resettable ones. Chen et al. 2009 define a transformation to the Timed Interval Calculus in PVS, treat enabled and triggered subsystems (§4.5 and appendix C) but not resets, and focus on verification not compilation. Building on the ClawZ formalization [Adams and Clayton 2005; Arthan et al. 2000] in ProofPower, Cavalcanti et al. 2011 propose a framework for proving refinement relations between discrete-time diagrams and parallel Spark Ada implementations. They treat enabled subsystems but do not explain if or how resets are handled. We focus on sequential code generation: any program accepted by the compiler should be correctly and automatically transformed into executable code. Bouissou and Chapoutot 2012 define an operational semantics for Simulink that treats a subset of discrete- and continuous-time blocks. Enabled and triggered

subsystems are inlined and augmented with switch blocks, but resetting and code generation are ignored. Zou et al. 2013 encode diagrams as Hybrid CSP terms in Isabelle/HOL. The work is extended to handle Stateflow [Zou et al. 2015] and to provide invariant generation and proofs of program properties [Chen et al. 2017]. Enabled and triggered subsystems are treated (§3.2.6) by flattening them and adding a guard that synchronizes with an input trigger before executing the translated process. Resettable subsystems are not supported and the transformation to Hybrid CSP is not formally verified. The resulting process-algebraic models describe models as step-by-step calculations whereas we work with compositions of stream functions.

Hamon and Rushby 2004 and Hamon 2005 formalize the semantics of Stateflow and show that it is essentially an imperative language with a graphical syntax. As for the work on Statecharts, these proposals are not directly relevant to the problems treated in earlier sections.

In terms of resets, the work of Zhou and Kumar 2012 is the most relevant. They model diagrams by composing transition systems. Enabled subsystems are reset by defining (§4) a *conditioning rule* operator on subsystems. Resetting is modeled by a function that updates state and output variables. The function does not arise from compositions, as in Lustre, but is rather constructed by the transformation separately for each subsystem. Transitions are composed sequentially—according to the execution order calculated by Simulink—to update the state as in a sequential language.

Like the formalizations cited above, our work treats a block-diagram language for programming embedded controllers. It is distinguished by a semantics based on streams with explicit presence and absence, and defined by mutually inductive constraints with support for resetting subsystems.

Tripakis et al. 2005 propose a translation from discrete-time Simulink models into Lustre. It supports triggered and enabled subsystems §§6.4 and 6.5 but not the reset feature. Resets should be easy to add, however, and our proposal would then provide an (indirect) mechanized semantics and a verified code generator for a subset of discrete-time Simulink.

5 CONCLUSION

We propose a constraint-based semantics to treat an often ignored but fundamental feature of programming languages for Model-Based Design. We show its utility for reasoning within an ITP by proving the correctness of two compilation passes. The proofs remain intricate, but the introduction of a model with explicit memory and a novel intermediate language based on synchronized transitions permit straightforward lemma statements and engender natural proof obligations.

Corollary 3.5 and theorem 3.9 are chained together to recover the correctness theorem stated by Bourke et al. 2017 that links the semantics of the source dataflow program, now extended with the reset operator, to that of the assembly code generated by the CompCert backend.

Our model is stream based with an explicit representation of presence and absence to facilitate reasoning about deterministic, iterated, conditionally activated functions. Even though not all reactive languages are formalized in this way, our ideas and definitions may still provide a useful starting point for formalizing other reset operators, especially for reasoning within an ITP.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed suggestions, Alain Girault for his felicitous observations, Cédric Pasteur for pointing out connections with related work, and Jean-Louis Colaço, Pierre-Évariste Dagand, and Xavier Leroy for their insightful remarks and support.

This work was partially supported by the Bpifrance Invest for the Future Program (“Programme d’Investissements d’Avenir”) in the ES3CAP project.

REFERENCES

Mark M. Adams and Philip B. Clayton. 2005. ClawZ: Cost-Effective Formal Verification for Control Systems. In *7th Int. Conf. on Formal Methods and Software Engineering (ICFEM 2005) (LNCS)*, Kung-Kiu Lau and Richard Banach (Eds.), Vol. 3785. Springer, Manchester, UK, 465–479. https://doi.org/10.1007/11576280_32

Rajeev Alur, Aditya Kanade, S. Ramesh, and K.C. Shashidhar. 2008. Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models. In *Proc. 8th ACM Int. Conf. on Embedded Software (EMSOFT 2008)*. ACM Press, Atlanta, GA, USA, 89–98. https://www.cis.upenn.edu/~alur/Emsoft08GM.pdf

Rob D. Arthan, Paul Caseley, Colin O'Halloran, and Alf Smith. 2000. ClawZ: control laws in Z. In *2nd Int. Conf. on Formal Methods and Software Engineering (ICFEM 2000)*, Shaoying Liu, John A. McDermid, and Michael G. Hinchey (Eds.). IEEE Computer Society, York, UK, 169–176. https://doi.org/10.1109/ICFEM.2000.873817

Cédric Auger. 2013. *Compilation certifiée de SCADE/LUSTRE.* Ph.D. Dissertation. Univ. Paris Sud 11, Orsay, France. https://tel.archives-ouvertes.fr/tel-00818169/document

Cédric Auger, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 2014. A Formalization and Proof of a Modular Lustre Code Generator. (2014). In preparation.

Gérard Berry. 1989. Real Time Programming: Special Purpose or General Purpose Languages. In *Proc. 11th Int. Federation for Information Processing (IFIP) World Computer Congress*, Gerhard Ritter (Ed.). Int. Federation for Information Processing (IFIP), San Francisco, USA, 11–17. https://hal.inria.fr/inria-00075494/document

Gérard Berry. 1993. Preemption in Concurrent Systems. In *Foundations of Software Technology and Theoretical Computer Science (LNCS)*, R. K. Shyamasundar (Ed.), Vol. 761. Springer, Bombay, India, 72–93. http://www-sop.inria.fr/members/Gerard.Berry/Papers/preemption.zip

Gérard Berry. 2000. *The Esterel v5 Language Primer* (5.91 ed.). Ecole des Mines and INRIA. http://www-sop.inria.fr/members/Gerard.Berry/Papers/primer.zip

Gérard Berry. 2002. *The Constructive Semantics of Pure Esterel* (draft version 3 ed.). Sophia-Antipolis, France. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf

Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. ACM Press, Tucson, AZ, USA, 121–130. https://www.di.ens.fr/~pouzet/bib/lctes08a.pdf

Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Automated Reasoning* 43, 3 (Oct. 2009), 263–288. https://hal.inria.fr/inria-00352524/document

Olivier Bouissou and Alexandre Chapoutot. 2012. An operational semantics for Simulink's simulation engine. In *Proc. 13th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2012)*, Reinhard Wilhelm, Heiko Falk, and Wang Yi (Eds.). ACM Press, Beijing, China, 129–138. https://perso.ensta-paristech.fr/~chapoutot/recherche/lctes12_bc.pdf

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM Press, Barcelona, Spain, 586–601. https://doi.org/10.1145/3062341.3062358

Timothy Bourke, Lélio Brun, and Marc Pouzet. 2018. Towards a verified Lustre compiler with modular reset. In *Proc. 21st Int. Workshop on Software and Compilers for Embedded Systems (SCOPES'18)*. ACM, Sankt Goar, Germany, 14–17. https://doi.org/10.1145/3207719.3207732

Timothy Bourke and Marc Pouzet. 2019. Arguments cadencés dans un compilateur Lustre vérifié. In *30^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2019)*, Nicolas Magaud and Zaynah Dargaye (Eds.). Les Rousses, Haut-Jura, France, 109–124. https://hal.inria.fr/hal-02005639/document

Paul Caspi. 1992. Clocks in dataflow languages. *Theoretical Computer Science* 94, 1 (March 1992), 125–140. https://doi.org/10.1016/0304-3975(92)90326-B

Paul Caspi. 1994. Towards recursive block diagrams. *Annual Review in Automatic Programming* 18 (1994), 81–85. https://doi.org/10.1016/0066-4138(94)90015-9

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1987)*. ACM Press, Munich, Germany, 178–188. https://doi.org/10.1145/41625.41641

Paul Caspi and Marc Pouzet. 1997. *A Co-iterative Characterization of Synchronous Stream Functions.* Research Report 97-07. VERIMAG, Gières, France.

Paul Caspi and Marc Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *First Workshop on Coalgebraic Methods in Computer Science (CMCS'98) (ENTCS)*, Vol. 11. Elsevier Science, Lisbon, Portugal, 1–21. https://doi.org/10.1016/S1571-0661(04)00050-7

Ana Cavalcanti, Phil Clayton, and Colin O'Halloran. 2011. From control law diagrams to Ada via Circus. *Formal Aspects of Computing* 23, 4 (July 2011), 465–512. https://www-users.cs.york.ac.uk/~alcc/publications/papers/CCO11.pdf

Alexandre Chapoutot and Matthieu Martel. 2009. Abstract Simulation: A Static Analysis of Simulink Models. In *Proc. Int. Conf. on Embedded Software and Systems (ICESS 2009)*. IEEE Computer Society, IEEE, Zhejiang, China, 83–92. http://www.ensta-paristech.fr/~chapoutot/recherche/icess09_chapoutot_martel.pdf

Chunqing Chen, Jin Song Dong, and Jun Sun. 2009. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing* 21, 5 (Oct. 2009), 451–483. https://doi.org/10.1007/s00165-009-0108-9

Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou. 2017. MARS: A Toolchain for Modelling, Analysis and Verification of Hybrid Systems. In *Provably Correct Systems (NASA Monographs in Systems and Software Engineering)*, Mike G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog (Eds.). Springer, Cham, Switzerland, 39–58. https://doi.org/10.1007/978-3-319-48628-4_3

Albert Cohen, Léonard Gérard, and Marc Pouzet. 2012. Programming Parallelism with Futures in Lustre. In *Proc. 12th ACM Int. Conf. on Embedded Software (EMSOFT 2012)*. ACM Press, Tampere, Finland, 197–206. https://doi.org/10.1145/2380356.2380394

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2005. A Conservative Extension of Synchronous Data-flow with State Machines. In *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. ACM Press, Jersey City, USA, 173–182. https://doi.org/10.1145/1086228.1086261

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. Scade 6: A Formal Language for Embedded Critical Software Development. In *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*. IEEE Computer Society, Nice, France, 4–15. https://hal.inria.fr/hal-01666470/document

Jean-Louis Colaço and Marc Pouzet. 2003. Clocks as First Class Abstract Types. In *Proc. 3rd Int. Conf. on Embedded Software (EMSOFT 2003) (LNCS)*, Vol. 2855. Springer, Philadelphia, PA, USA, 134–155. https://doi.org/10.1007/978-3-540-45212-6_10

Jean-Louis Colaço and Marc Pouzet. 2004. Type-based initialization analysis of a synchronous dataflow language. *Int. J. Software Tools for Technology Transfer* 6, 3 (Aug. 2004), 245–255. https://www.di.ens.fr/~pouzet/bib/sttt04.pdf

Coq Development Team. 2019. *The Coq proof assistant reference manual*. Inria. https://coq.inria.fr/distrib/current/refman/ v. 8.9.

Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. 2012. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *Proc. 13th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2012)*, Reinhard Wilhelm, Heiko Falk, and Wang Yi (Eds.). ACM Press, Beijing, China, 51–60. https://www.di.ens.fr/~guatto/papers/lctes12.pdf

Grégoire Hamon. 2005. A Denotational Semantics for Stateflow. In *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. ACM Press, Jersey City, USA, 164–172. https://doi.org/10.1145/1086228.1086260

Gégoire Hamon and Marc Pouzet. 2000. Modular Resetting of Synchronous Data-Flow Programs. In *Proc. 2nd ACM SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming (PPDP 2000)*, Frank Pfenning (Ed.). ACM, Montreal, Canada, 289–300. https://doi.org/10.1145/351268.351300

Grégoire Hamon and John Rushby. 2004. An Operational Semantics for Stateflow. In *Proc. 7th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'04) (LNCS)*, Michel Wermelinger and Tiziana Margaria-Steffen (Eds.), Vol. 2984. Springer, Barcelona, Spain, 229–243. http://www.csl.sri.com/users/rushby/papers/sttt07.pdf

David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

David Harel and Amnon Naamad. 1996. The STATEMATE Semantics of Statecharts. *ACM Trans. Software Engineering and Methodology (TOSEM)* 5, 4 (Oct. 1996), 293–333. https://doi.org/10.1145/235321.235322

Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. 2019. *The Lustre V6 Reference Manual*. Verimag, Grenoble. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) parsers. In *21st European Symposium on Programming (ESOP 2012), held as part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012) (LNCS)*, Helmut Seidl (Ed.), Vol. 7211. Springer, Tallinn, Estonia, 397–416. https://hal.inria.fr/hal-01077321/document

Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Proc. Int. Federation for Information Processing (IFIP) Congress 1974*, Jack L. Rosenfeld (Ed.). North-Holland, Stockholm, Sweden, 471–475. https://perso.ensta-paristech.fr/~chapoutot/various/kahn_networks.pdf

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*. ACM Press, San Diego, CA, USA, 179–191. https://cakeml.org/popl14.pdf

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Comms. ACM* 52, 7 (2009), 107–115. https://hal.inria.fr/inria-00415861/document

Florence Maraninchi and Nicolas Halbwachs. 1996. Compiling Argos into Boolean equations. In *Proc. 4th Int. Symp. Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '96) (LNCS)*, Bengt Jonsson and Joachim Parrow (Eds.), Vol. 1135. Springer, Uppsala, Sweden, 72–89. http://www-verimag.imag.fr/~halbwach/FTRTFT96.ps

Florence Maraninchi and Yann Rémond. 2001. Argos: an automaton-based synchronous language. *Computer Languages* 27, 1–3 (2001), 61–92. https://hal.archives-ouvertes.fr/hal-00273055/document

Florence Maraninchi and Yann Rémond. 2003. Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems. *Science of Computer Programming* 46, 3 (2003), 219–254. https://doi.org/10.1016/S0167-6423(02)00093-X

Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier. 2015. Translation Validation for Synchronous Data-Flow Specification in the SIGNAL Compiler. In *Proc. 35th IFIP WG 6.1 Int. Conf. on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015) (LNCS)*, Susanne Graf and Mahesh Viswanathan (Eds.), Vol. 9039. Springer, Grenoble, France, 66–80. https://hal.inria.fr/hal-01767328

Christine Paulin-Mohring. 2009. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin (Eds.). CUP, Cambridge, UK, 383–413. https://hal.inria.fr/inria-00431806/document

Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries: The Revised Report.* CUP, Cambridge, UK. https://www.haskell.org/definition/haskell98-report.pdf

Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel.* Springer, New York, NY, USA. https://doi.org/10.1007/978-0-387-70628-3

Marc Pouzet. 2006. *Lucid Synchrone, v. 3. Tutorial and reference manual.* Université Paris-Sud. https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf

Michael Ryabtsev and Ofer Strichman. 2009. Translation Validation: From Simulink to C. In *Proc. 21st Int. Conf. on Computer Aided Verification (CAV 2009) (LNCS)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, Grenoble, France, 696–701. https://doi.org/10.1007/978-3-642-02658-4_57

Gang Shi, Yuanke Gan, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew. 2017. A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs. In *Proc. 39th Int. Conf. on Software Engineering Companion (ICSE-C'17)*. IEEE Press, Buenos Aires, Argentina, 109–111. https://doi.org/10.1109/ICSE-C.2017.83

Gang Shi, Yucheng Zhang, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew. 2019. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. *Science China Information Sciences* 62, 1 (Jan. 2019), article 12801. https://doi.org/10.1007/s11432-016-9270-0

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *Proc. 21st ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2016)*. ACM Press, Nara, Japan, 60–73. https://cakeml.org/icfp16.pdf

Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. 2005. Translating Discrete-Time Simulink to Lustre. *ACM Trans. Embedded Computing Systems* 4, 4 (Nov. 2005), 779–818. http://www-verimag.imag.fr/~tripakis/papers/acm-tecs05.pdf

Changyan Zhou and Ratnesh Kumar. 2012. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems* 22, 2 (June 2012), 223–247. http://home.engineering.iastate.edu/~rkumar/PUBS/ss2efa1.pdf

Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. 2015. Formal Verification of Simulink/Stateflow Diagrams. In *Proc. 13th Int. Symp. Automated Technology for Verification and Analysis (ATVA 2015) (LNCS)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.), Vol. 9364. Springer, Shanghai, China, 464–481. http://lcs.ios.ac.cn/~znj/papers/atva2015b.pdf

Liang Zou, Naijun Zhan, Shuling Wang, Martin Fränzle, and Shengchao Qin. 2013. Verifying Simulink Diagrams via a Hybrid Hoare Logic Prover. In *Proc. 13th ACM Int. Conf. on Embedded Software (EMSOFT 2013)*. IEEE, Montreal, Canada, 9:1–9:10. https://www.scedt.tees.ac.uk/s.qin/papers/emsoft13-final.pdf