A Formalization and Proof of a Modular Lustre Compiler

Cédric Auger\textsuperscript{a}, Jean-Louis Colaço\textsuperscript{b}, Grégoire Hamon\textsuperscript{c}, Marc Pouzet\textsuperscript{d,∗}

\textsuperscript{a}LRI, Université Paris-Sud 11, 91405 Orsay cedex, France.
\textsuperscript{b}Prover Technology SA, 21 rue Alsace Lorraine 31000 Toulouse, France.
\textsuperscript{c}The MathWorks, 3 Apple Hill Drive, Natick, MA 01760-2098, USA.
\textsuperscript{d}Ecole Normale Supérieure, DI, 45 rue d’Ulm, 75230 Paris cedex 05, France.

Abstract

This paper presents a formalization of the compilation of a synchronous data-flow language into an imperative sequential language. We consider MiniLS, a minimalistic yet full-featured synchronous language reminiscent of Lustre. It provides original constructs such as a reset and an n-ary merge operator. These constructs play a central role in generating efficient code and in making the language suitable as a backend for compiling advanced features such as hierarchical state machines.

We introduce a generic imperative language to represent transition functions and a clock-directed translation from the source into this language. This translation is modular: every synchronous function is translated into a single transition function. We address the target code generation phase by presenting code emitters to Java and C.

The paper comes with a precise description of each compilation step, a formal semantics for the source and destination languages and a proof that whenever the compilation succeeds, it produces a sequential program which is semantically equivalent to the source. To our knowledge, this is the first synchronous realistic compiler with such a correctness property. The formalization of the compilation process and its proof of correctness are an important step toward fully-verified implementations of compilers for languages based on synchronous block diagrams.

Keywords: Real-time systems; Synchronous languages; Compilation; Semantics; Type systems

1. Introduction

Synchronous block diagram formalisms like SCADE\textsuperscript{1} or Simulink\textsuperscript{2} are widely used for the design of embedded systems. There are many benefits in using such languages in this context. They allow a high-level description of the system and naturally lend themselves to a hierarchical design. The translation of synchronous block diagrams into sequential code is well understood and reasonably simple, it also produces very efficient code. These benefits match the conflicting needs for both highly efficient and highly trusted code when dealing with critical, but resource constrained systems.

The principles behind code generation for synchronous block diagrams have been studied and documented since the early days of the languages Lustre [2] and Signal [3]. However, actual code generator implementations, in particular industrial ones, differ from these principles, be it by supporting advanced features such as control structures, or by adapting them to support modular

\textsuperscript{1}http://www.esterel-technologies.com/products/scade-suite/
\textsuperscript{2}http://www.mathworks.fr/products/simulink/index.html

\textsuperscript{∗}Corresponding author

Email addresses: Cedric.Auger@lri.fr (Cédric Auger), Jean-Louis.Colaco@prover.com (Jean-Louis Colaço), Gregoire.Hamon@mathworks.com (Grégoire Hamon), Marc.Pouzet@ens.fr (Marc Pouzet)

\textsuperscript{This work was supported by the SYNCHRONICS large scale initiative of INRIA. A preliminary version of this work appeared in [1]. The present paper details the compilation technique and organisation of the compiler. Moreover, the semantics development and the proof of compilation correctness are entirely new.

Preprint submitted to Elsevier October 24, 2012
code generation. The importance for high confidence in code generators prompted the certification of the SCADE compiler, so that it could be used in the most critical software (norm DO178B, level A). Nonetheless, this certification does not require a formal definition of the compilation process.

In this work, we describe and formalize a simple, yet full-featured, synchronous data-flow language named MiniLS. We formalize its compilation into sequential code and prove the correctness of the compilation process. Fully verified implementations of compilers have recently become a possibility, thanks to progress in the fields of both theorem proving and compiler technology [4, 5]. Our goal is to open the door to fully verified implementations of code generators for languages based on synchronous block diagrams. The present paper describes the techniques and first experiments toward that end.

1.1. Compiling Synchronous Data-flow Languages

**Transition Function and Initial State**

Synchronous data-flow languages are functional languages over streams — infinite sequences of values. Given a stream function \( f : \text{Stream}(T) \rightarrow \text{Stream}(T') \) and an equation \( y = f(x) \), the code generation consists in building a pair \((f_t, s_0)\) where \( f_t \) is the transition function of type \( S \times T \rightarrow T' \times S \) and \( s_0 \) is the initial state of type \( S \). The pair is such that \((y_n, s_{n+1}) = f_t(s_n, x_n)\) for every time step \( n \), where \( x = (x_i)_{i \in \mathbb{N}} \) and \( y = (y_i)_{i \in \mathbb{N}} \). The transition function takes a state and the current input, and it returns the current output and a new state. In actual implementations, the transition function is written in an imperative style with in-place modification of the state. Synchrony finds a practical justification here: an infinite stream of type \( \text{Stream}(T) \) is represented by a value of type \( T \) and no intermediate memory nor complex buffering mechanism is needed. This principle generalizes to functions with multiple inputs and multiple outputs.

**Modular Code Generation**

A stream function is defined by a set of equations over streams. Code generation is obtained through a static scheduling of the equations according to data dependencies. Separate, or modular, code generation produces a single transition function for each node definition. As noticed by Gonthier [6], this modular code generation is not always possible. Indeed, if \( \text{copy} \) is a two-input/two-output stream function such that \( \text{copy}(x, y) = (x, y) \), then the equation \((y, z) = \text{copy}(t, y)\) defines two perfectly valid streams \( y \) and \( z \) (since \( y = t \) and \( z = y = t \)). However, it is not possible to generate a single transition function for \( \text{copy} \) which would work correctly in all valid contexts. This observation has led to two main approaches to the compilation problem: maximal expressiveness through inlining, or modular code generation by imposing restrictions on feedback loops.

**Maximal Expressiveness and Inlining**

The first approach aims to keep maximal expressiveness of the source language and consists in compiling the program after a full inlining of function calls. This is the solution adopted in the academic compiler of Lustre. The resulting set of equations is statically scheduled and translated into imperative code. Then, a forward or backward enumeration of state variables can be applied to generate an explicit finite state automaton [2, 7]. Unfortunately, the size of the generated code explodes in practice. Enumeration must be restricted to a selected set of state variables but finding adequate variables that lead to efficient code in terms of both size and time remains difficult. Modular code generation is sacrificed in this solution and it is not used in industrial compilers.

**Modular Code Generation by Restrictions**

Modular compilation is mandatory in industrial compilers like the one of the SCADE-Suite. In such compilers, every stream function is translated into one imperative function with no preliminary inlining unless requested by the programmer. Stronger causality constraints need to be imposed: every feedback loop must cross an explicit delay and this is well accepted by SCADE users. Modular compilation is also justified by the need for traceability of the generated code and
the simplicity of the code-generation step, as required by certification authorities in the context of critical software.

Between these two approaches — maximal code duplication vs maximal code sharing — an intermediate solution has been proposed by Raymond [8]. It consists in generating for every stream function \( f \) a set of transition functions, rather than a single one, which are called directly according to the data-dependencies between inputs and outputs at an instantiation of \( f \). The same idea has also been considered for SIGNAL [9]. New algorithms have been proposed recently in [10] and [11]. These solutions are not implemented in real compilers so we stay with the simpler and traditional modular compilation technique.

1.2. MiniLS, a clocked declarative synchronous language and its compilation

The purpose of this paper is to present and formalize the translation of a synchronous language into sequential code. The input language is MiniLS, a data-flow language reminiscent of Lustre. We make it minimal, yet expressive enough to encode richer constructs such as hierarchical automata. The language provides an \( n \)-ary merge operator as a way to combine complementary streams together with a reset construct to restart a component in a modular way. The compilation proceeds as follows. The input program is first type-checked, giving a program annotated with type information. Then, the compiler applies a static analysis, named the clock calculus, whose goal is to annotate every expression \( e \) with a clock. The clock is a boolean formula that defines the instants where \( e \) has a defined value. Then, the source program is put into normal form and statically scheduled. Programs are translated modularly into programs in an object-based intermediate language representing transition functions. Finally, we illustrate the versatility of this intermediate language by giving a translation into JAVA and C.

Clocks play a central role during the translation in order to generate efficient sequential code. This clock-directed approach contrasts with compilation methods based on enumeration techniques [7]. The use of an intermediate language and the special treatment of clocks lead to a concise description of the compilation process, yet it produces efficient sequential code. Moreover, several steps in the compiler are expressed as source-to-source transformations. This makes them easier to define and to formally check. Finally, the ability to generate efficient code for control structures is also key in making MiniLS a good target for compiling languages with more advanced features, such as state-machines [12, 13].

This paper includes a formal description of the semantics of the source language, the intermediate sequential language and of the essential steps of the compiler. This formalization allows a proof of the correctness of the compilation process, i.e., when the compilation succeeds, the source code and the target code are semantically equivalent.

1.3. Paper Organization

The paper is composed of two parts: the first presents the compilation process and anybody interested in building a compiler for a language like Lustre should be able to do so from its content. This part is composed of Sections 2 to 7. In Section 2 we present MiniLS and its static semantics. In Section 3, we address the question of exposing the state variables of the program and the static scheduling of equations. Section 4 introduces the intermediate sequential language which is the target of the translation presented in Section 5. In Section 6, we describe code generation to JAVA and C, and in Section 7, we sketch the implementation of a complete compiler. The second part of the paper, in Section 8, contains formal semantics of both the source and the intermediate languages, as well as the proof of semantics preservation of the translation. We present it separately from the first part as it is not mandatory to the understanding of the architecture of the compiler and each of its steps. Finally, in Sections 10 and 11, we discuss related and future work.

2. A Clocked Data-flow Language

In this section, we present MiniLS, a synchronous data-flow kernel that is powerful enough to encode Lustre. This language contains advanced features with respect to Lustre: a means to
reset a function application in a modular way, value constructors belonging to enumerated types and a filtering mechanism.

A program is defined by a number of node definitions. Each node computes its outputs from its inputs via a collection of parallel equations. Figure 1 presents a SCADE block diagram and its MINILS representation. The node counting has two input parameters tick and top, an output parameter o and it counts the number of top events between two tick events. The subexpression 0 fby o denotes the previous value of o, initialized with value 0, and v is an auxiliary variable equal to 1 when top is true and equal to 0 otherwise.

Code generation begins after two static analyses: the type and clock verifications, after which, every expression is annotated with type and clock information. To simplify the presentation, this paper considers the annotated language directly. The source language can be recovered by dropping the annotations.

2.1. The Annotated Language

A program in the kernel language comprises a list of global type (td) and node (d) declarations. bt denotes a type identifier, either abstract (type bt) or introducing n enumerated values (type bt = C1 + ... + Cn) pairwise different. A global node declaration node f(p1) returns (p2) var p3 in D defines a node f with inputs p1, outputs p2 and local variables p3 and D, a list of equations. An equation (x = a) defines the value of x or it can be the application of a function with n inputs and m outputs (x1,...,xm) = f(a1,...,an) every a. The application is reset every time a is true.

Finally, it can be the conjunction of two equations sets (D and D). a stands for an expression e annotated with its type (bt) and (ck). Expressions (e) are made of constants (v), variables (x), initialized delays (v fby a), the point-wise application of an external operator op (e.g., +, not) to its arguments (op (a1,...,an)), a sampling operation (a when C(x)) and a combination operation (merge x (C1 → a1) ... (Cn → an)). A value (v) can be a constructor (C) belonging to an enumerated type or an immediate value (i) (e.g., an integer). We assume the existence of an initial environment defining the boolean type bool = False + True. In the same way, combinatorial functions are provided externally. The grammar is given below:

--- count the number of tops between two ticks
node counting (tick:bool; top:bool) returns (o:int)
var v:int in
  o = if tick then v else (0 fby o) + v
and
  v = if top then 1 else 0

Figure 1: The counting node in SCADE and in MINILS

---
\[ td ::= \text{type } bt \mid \text{type } bt = C + \ldots + C \]
\[ a ::= e^c_{bt} \]
\[ d ::= \text{node } f(p) \text{ returns } (p) \text{ var } p \text{ in } D \]
\[ e ::= \text{node } f(a) \mid \text{var } a \text{ when } C(x) \]
\[ p ::= x : bt ; \ldots ; x : bt \]
\[ D ::= x = a \mid \text{and } D \mid \text{every } (x, \ldots, x) = f(a, \ldots, a) \]
\[ v ::= C \mid i \]
\[ ck ::= \text{base} \mid \text{ck on } C(x) \]

If \( a \) is an expression and \( D \) a set of equations, \( \text{Vars}(a) \) is the set of variables appearing in \( a \) and \( \text{Vars}(D) \) the set of variables appearing in \( D \).

The expression \( a \text{ when } C(x) \) is the sampled stream of \( a \) on the instants where \( x \) equals \( C \).

Dually, \( \text{merge} \) is the combination operator: if \( a \) is a stream producing values belonging to a finite enumerated type \( bt = C_1 + \ldots + C_n \) and \( a_1, \ldots, a_n \) are mutually complementary streams (i.e., at a given instant, at most one stream is producing a value), then it combines them to form a faster stream.

\[(x_1, \ldots, x_m) = f(a_1, \ldots, a_n) \text{ every } a \]

is the resettable function application: the internal state of the application of \( f \) is reset every time the boolean stream \( a \) is true and the result is stored into \((x_1, \ldots, x_m)\). We simply write \((x_1, \ldots, x_m) = f(a_1, \ldots, a_n)\) as a shortcut for \((x_1, \ldots, x_m) = f(a_0, \ldots, a_n) \text{ every } False\).

The first argument in the initialized delay \( v \text{ fby } a \) is a constant. If \( op \) is a combinatorial function, \( op(a_1, \ldots, a_n) \) applies it point-wise to its arguments (classical arithmetic operations are written in infix form). The semantics of data-flow primitives does not depend on the annotations so we omit them in the example below.

| \( x \) | \( x_0 \) | \( x_1 \) | \( x_2 \) | \( x_3 \) | \ldots |
| \( y \) | \( y_0 \) | \( y_1 \) | \( y_2 \) | \( y_3 \) | \ldots |
| \( v \text{ fby } x \) | \( v \) | \( x_0 \) | \( x_1 \) | \( x_2 \) | \ldots |
| \( x + y \) | \( x_0 + y_0 \) | \( x_1 + y_1 \) | \( x_2 + y_2 \) | \( x_3 + y_3 \) | \ldots |

This kernel language is slightly different from the original LUSTRE in the choice of the basic primitives. It is no less expressive, however.

2.2. Derived Operators

The classical sampling operators, conditional and delays of LUSTRE can be encoded in the basic calculus.

2.2.1. Sampling and Conditional

The kernel provides a general sampling mechanism based on enumerated types. The sampling operation \( e \text{ when } x \) of LUSTRE, where \( x \) is a boolean stream, is written \( e \text{ when } \text{True}(x) \). In the same way, \( e \text{ when } \text{not } x \) is written \( e \text{ when } \text{False}(x) \).

The conditional \( \text{if/then/else} \) is built from the more elementary operators \( \text{merge} \) and \( \text{when} \).

\[
\text{if } x \text{ then } e_2 \text{ else } e_3 = \text{merge } x \\\n\left( \text{True } \to e_2 \text{ when } \text{True}(x) \right) \\\n\left( \text{False } \to e_3 \text{ when } \text{False}(x) \right)
\]
2.2.2. Initialization and Delay

The initialization operation \( e_1 \rightarrow e_2 \) of LUSTRE returns the very first value of \( e_1 \) then the current value of \( e_2 \). The uninitialized delay \( \text{pre}(e) \) is a shortcut for \( \text{nil fby e} \) where \( \text{nil} \) stands for any constant value which has the type of \( e \).

\[
e_1 \rightarrow e_2 = \text{if True fby False then } e_1 \text{ else } e_2
\]

\[
\text{pre}(e) = \text{nil fby e}
\]

These operators are illustrated on the following chronogram.

<table>
<thead>
<tr>
<th>( h )</th>
<th>True</th>
<th>False</th>
<th>True</th>
<th>False</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x_0 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>( x_3 )</td>
<td>...</td>
</tr>
<tr>
<td>( y )</td>
<td>( y_0 )</td>
<td>( y_1 )</td>
<td>( y_2 )</td>
<td>( y_3 )</td>
<td>...</td>
</tr>
<tr>
<td>( x \rightarrow y )</td>
<td>( x_0 )</td>
<td>( y_1 )</td>
<td>( y_2 )</td>
<td>( y_3 )</td>
<td>...</td>
</tr>
<tr>
<td>( \text{pre}(x) )</td>
<td>( \text{nil} )</td>
<td>( x_0 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>...</td>
</tr>
<tr>
<td>( x \text{ when True}(h) )</td>
<td>( x_0 )</td>
<td>( x_2 )</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( y \text{ when False}(h) )</td>
<td>( y_1 )</td>
<td>( y_3 )</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{merge } h (\text{True } \rightarrow z) (\text{False } \rightarrow t) )</td>
<td>( x_0 )</td>
<td>( y_1 )</td>
<td>( x_2 )</td>
<td>( y_3 )</td>
<td>...</td>
</tr>
</tbody>
</table>

If \( x \) is a sequence, then \( z = x \text{ when True}(h) \) is a sub-sequence of \( x \) made of the instants where \( h \) is true. \( t = y \text{ when False}(h) \) is the sub-sequence of \( y \) made of the instants where \( h \) is false. Finally, \( \text{merge } h (\text{True } \rightarrow z) (\text{False } \rightarrow t) \) merges the two streams.

2.2.3. The Counting Example

Ignoring type and clock annotations, the counting example of Figure 1 is written below. We will refer to this example throughout the paper.

```plaintext
node counting(tick: bool, top: bool) returns (o: int)
var (v: int)
in
\[
o = \text{merge } \text{tick} (\text{True } \rightarrow v \text{ when True}(\text{tick}))
\quad (\text{False } \rightarrow ((0 \text{ fby } o) + v) \text{ when False}(\text{tick}))
and \( v = \text{merge } \text{top} (\text{True } \rightarrow (1 \text{ when True}(\text{top}))))
\quad (\text{False } \rightarrow (0 \text{ when False}(\text{top}))))
```

2.3. Annotating Expressions with their Type and Clock

The code generation phase applies once type and clock verification has been performed. At the end of these two steps, every expression is annotated with its type and clock. We only present the annotated language together with well-formation rules on those annotations, themselves automatically computed by the compiler.

Type checking does not raise any difficulty and we do not detail it here. After typing, the clock calculus is performed. This analysis is specific to synchronous languages and ensures that a program can be executed synchronously [15]. Moreover, the information computed during this analysis is essential to the compilation process presented later.

Clocks

The clock of a signal \( s \) is a boolean sequence which is true wherever \( s \) is present. It is represented as a boolean formula and we consider the following clock language:

\[
\text{ck} ::= \text{base} | \text{ck on } C(x)
\]

\[\text{The purpose of the initialization analysis is to check that the result of the program does not depend on the actual value of nil. Its description is beyond the scope of this paper [14].}\]
base stands for the base clock, that is, the constant sequence made of only true values. \(ck\) on \(C(x)\) defines a sub-clock of \(ck\): it evaluates to true when \(ck\) evaluates to true, \(x\) is present and evaluates to the constant \(C\).

A clock environment \(\mathcal{H}\) associates clocks to variables. It is of the form:

\[
\mathcal{H} ::= [x_1 : ck_1, ..., x_n : ck_n]
\]

where \(x_i \neq x_j\) for \(i \neq j\). \(\mathcal{H}_1, \mathcal{H}_2\) defines the union of \(\mathcal{H}_1\) and \(\mathcal{H}_2\) provided that their domains are disjoint. The clock constraints are defined by the following predicates.

1. \(\mathcal{H} \vdash a : ck\) and \(\mathcal{H} \vdash e : ck\) state that the annotated expression \(a\) and un-annotated expression \(e\) have clock \(ck\) in the clock environment \(\mathcal{H}\).
2. \(\mathcal{H} \vdash D\) states that \(D\) is a set of well-clocked equations in the clock environment \(\mathcal{H}\).
3. \(\vdash d\) checks the clock rule of the definition \(d\); \(\vdash p : \mathcal{H}\) builds an initial environment for local variables \(p\) and \(\vdash_{base} p : \mathcal{H}\) builds an initial environment for input/output parameters.

Their definitions are given in Figure 2. They are based on a simplification of the clock calculus of Lucid Synchrone [16]. The main restriction is that inputs and outputs of a node must be synchronous (rule (app)).

In the rules for when and merge, we assume that the expressions are type correct, in particular the control variable \(x\) and the type constructors have the same type \(bt\) with \(bt = C_1 + ... + C_n\). As stated previously, the clock calculus is applied after type checking, programs that are not type correct have therefore already been rejected. This system supports inference, clocks therefore do not have to be explicitly given in the source language. E.g., the programmer writes \((v\ fby\ x + y\ fby\ y)\) instead of \((v\ fby\ x^{ck} + y^{ck})^{ck}\), if \(ck\) is the clock of \(x\) and \(y\). Similarly the expression:

\[
\text{merge } h (\text{True } \rightarrow z^{ck} \text{ on True}(h))(\text{False } \rightarrow t^{ck} \text{ on False}(h))^{ck}
\]
is simply written \texttt{merge h (True \rightarrow z)(False \rightarrow t)} in the source code, provided \texttt{ck} is the clock of \texttt{h}.

### 2.3.1. The Counting Example

Given the typed counting program, the clock calculus returns the following program where every expression is now correctly annotated by its clock. The clock \texttt{base} is written \texttt{b} and type annotations are omitted.

\begin{verbatim}
node counting(tick : bool, top : bool) returns (o : int)
var (v : int) in
\begin{align*}
o &= \text{merge} \, \text{tick} \, (\text{True} \rightarrow (v^b \text{ when True}(\text{tick}))^{ck_1}) \\
&\quad \quad (\text{False} \rightarrow (((0 \text{ fby } o^b)^b + v^b)^b \text{ when False}(\text{tick}))^{ck_2})^b \\
\text{and } v &= \text{merge} \, \text{top} \, (\text{True} \rightarrow (1^b \text{ when True}(\text{top}))^{ck_3}) \\
&\quad \quad (\text{False} \rightarrow (0^b \text{ when False}(\text{top}))^{ck_4})^b \\
\text{with } ck_1 &= b \text{ on True}(\text{tick}), ck_2 = b \text{ on False}(\text{tick}), \\
ck_3 &= b \text{ on True}(\text{top}) \text{ and } ck_4 = b \text{ on False}(\text{top})
\end{align*}
\end{verbatim}

### 3. Normalization and Scheduling

The language of Section 2 is declarative with the evaluation of expressions controlled by the clock formalism. In order to generate sequential code from a program, we need (1) to identify its state as it needs special handling during the translation, and (2) to find a static order in which the computation can be done. Identification of the state is done by a normalization transformation that creates intermediate equations for stateful computation. After this normalization, finding a static evaluation order is a much simpler task.

#### 3.1. Putting Equations in Normal Form

We define a normal form that isolates stateful computations. The associated normalization transformation takes a program in the annotated language and normalizes it. The transformation traverses a set of equations and introduces new equations for each stateful computation.

#### 3.1.1. Example

The following equations (omitting type annotations and only printing clocks for the delays):

\begin{align*}
z &= ((4 \text{ fby } o)^{ck} \ast 3) \text{ when True}(c) + k \\
\text{and } o &= \text{merge } c \, (\text{True} \rightarrow (5 \text{ fby } z)^{ck \text{ on True}(c)} + (2 \text{ when True}(c))) \\
&\quad \quad (\text{False} \rightarrow ((6 \text{ fby } x)^{ck}) \text{ when False}(c))
\end{align*}

are rewritten into:

\begin{align*}
t_1 &= (4 \text{ fby } o)^{ck} \\
\text{and } t_2 &= (5 \text{ fby } z)^{ck \text{ on True}(c)} \\
\text{and } t_3 &= (6 \text{ fby } x)^{ck} \\
\text{and } z &= ((t_1 \ast 3) \text{ when True}(c)) + k \\
\text{and } o &= \text{merge } c \, (\text{True} \rightarrow t_2 + (2 \text{ when True}(c))) \\
&\quad \quad (\text{False} \rightarrow t_3 \text{ when False}(c))^{ck}
\end{align*}

Here we have introduced the new local variables \texttt{t\_1}, \texttt{t\_2} and \texttt{t\_3}. The extraction is made through a linear traversal, introducing equations for each stateful computation.
\[\text{NormA}_{\text{sub}}(e_{bt}^{ck}) = \text{NormE}_{\text{sub}}^{bt,ck}(e)\]
\[\text{NormE}_{\text{sub}}^{bt,ck}(v \ fby a) = \text{let } a', \ sub' = \text{NormA}_{\text{sub}}(a) \text{ in} x, sub' + [(v \ fby a')_{bt}^{ck} / x] \text{ where } x \notin \text{Dom}(sub')\]
\[\text{NormE}_{\text{sub}}^{bt,ck}(a \text{ when } C(x)) = \text{let } a', \ sub' = \text{NormA}_{\text{sub}}(a) \text{ in } a' \text{ when } C(x), sub'\]
\[\text{NormE}_{\text{sub}}^{bt,ck}(x) = x, sub\]
\[\text{NormE}_{\text{sub}}^{bt,ck}(v) = v, sub\]
\[\text{NormE}_{\text{sub}}^{bt,ck}(\text{op}(\vec{a})) = \text{let } \vec{a}', \ sub' = \text{NormA}_{\text{list}}(\vec{a}) \text{ in op}(\vec{a}), sub'\]
\[\text{NormE}_{\text{sub}}^{bt,ck}(\text{merge } x (C \rightarrow a)) = \text{let } \vec{a}', \ sub' = \text{NormA}_{\text{list}}(\vec{a}) \text{ in} y, sub + [\text{merge } x (C \rightarrow a')_{bt}^{ck} / y] \text{ where } y \notin \text{Dom}(sub')\]
\[\text{NormCA}_{\text{sub}}((\text{merge } x (C \rightarrow a))_{bt}^{ck}) = \text{let } \vec{a}', \ sub' = \text{NormA}_{\text{list}}(\vec{a}) \text{ in} (\text{merge } x (C \rightarrow a'))_{bt}^{ck}, sub'\]
\[\text{NormCA}_{\text{sub}}(a) = \text{NormA}_{\text{sub}}(a) \text{ for the remaining forms of } a\]
\[\text{NormD}_{\text{sub}}(x = a) = \text{let } a', \ sub' = \text{NormA}_{\text{sub}}(a) \text{ in } x = a', sub'\]
\[\text{NormD}_{\text{sub}}(\vec{x} = f(\vec{a}) \text{ every } a_0) = \text{let } \vec{a}', \ sub' = \text{NormA}_{\text{list}}(\vec{a}) \text{ in} \text{let } a'_0, sub'' = \text{NormA}_{\text{sub'}}(a_0) \text{ in} \vec{x} = f(\vec{a}') \text{ every } a'_0, sub''\]
\[\text{NormD}_{\text{sub}}(D_1 \text{ and } D_2) = \text{let } D_1, sub_1 = \text{NormD}_{\text{sub}}(D_1) \text{ in} \text{let } D_2, sub_2 = \text{NormD}_{\text{sub}}(D_2) \text{ in } D_1 \text{ and } D_2, sub_2\]
\[\text{NormA}_{\text{list}}(a_1, ..., a_n) = \text{let } a_1, sub_1 = \text{NormA}_{\text{sub}}(a_1) \text{ in} \text{...let } a_n, sub_n = \text{NormA}_{\text{list}}(a_n) \text{ in } (a_1, ..., a_n), sub_n\]
\[\text{NormCA}_{\text{list}}(a_1, ..., a_n) = \text{let } a_1, sub_1 = \text{NormCA}_{\text{sub}}(a_1) \text{ in} \text{...let } a_n, sub_n = \text{NormCA}_{\text{list}}(a_n) \text{ in } (a_1, ..., a_n), sub_n\]

**Figure 3:** A normalization function

**Definition 1 (Normal Form).** The predicate NA(a) holds if the annotated expression a is in normal form. Similarly, NCA(a) holds for normalized control expressions and NEQ(D) for normalized collections of equations.

\[
\begin{array}{cccc}
\text{NA}(x_{bt}^{ck}) & \text{NA}(v_{bt}^{ck}) & \text{NA}(a) & \text{NA}(a_1) \ldots \text{NA}(a_n) \\
\text{NCA}(a_1) & \ldots & \text{NCA}(a_n) & \text{NA}((a \text{ when } C(x))_{bt}^{ck}) \ldots \text{NA}((\text{op}(a_1, ..., a_n))_{bt}^{ck}) \\
\text{NEQ}(x_1, ..., x_n) = f(a_1, ..., a_n) \text{ every } a & \text{NA}(a) & \text{NA}(a) & \text{NCA}(a) \\
\end{array}
\]

There are various ways to define the normalization function. It mainly introduces auxiliary equations, replacing certain sub-expressions with a variable. Instead of conforming to a particular normalization function, we define the valid transformation from one equation to another.
Definition 2 (Convertibility). Let $D[a]$ denote a set of equations $D$ containing an expression $a$. $D[a] \cong D'$ holds when:

$$D[a] \cong x = a \text{ and } D[x] \quad \text{if } x \not\in \text{Vars}(D[a])$$

$$D_1 \cong D_3 \quad \text{if } D_1 \cong D_2 \land D_2 \cong D_3$$

$\text{Norm}(D)$ is the set of all normalized equations convertible with $D$.

$$\text{Norm}(D) = \{D' \mid D' \cong D' \land \text{NEQ}(D')\}$$

3.1.2. The Counting Example

Once normalized, the set of equations of the counting node becomes:

$$t = (0 \text{ fby } o^b)^b$$

and

$$o = (\text{merge } \text{tick } (\text{True } \rightarrow (v^b \text{ when True}(\text{tick}))^{ck_1})$$

$$(\text{False } \rightarrow ((v^b + v^b)^b \text{ when False}(\text{tick}))^{ck_2})^b)$$

and

$$v = (\text{merge } \text{top } (\text{True } \rightarrow (1^b \text{ when True}(\text{top}))^{ck_3})$$

$$(\text{False } \rightarrow (0^b \text{ when False}(\text{top}))^{ck_4})^b)$$

with $b = \text{base}$, $ck_1 = b \text{ on True}(\text{tick})$, $ck_2 = b \text{ on False}(\text{tick})$, $ck_3 = b \text{ on True}(\text{top})$ and $ck_4 = b \text{ on False}(\text{top})$

The variable $t$ is introduced to store the result of the fby expression.

A possible implementation of the normalization transformation is presented in Figure 3. Let $\text{sub} = [a_1/x_1, \ldots, a_n/x_n]$ denote a substitution. $\text{NormA}_{\text{sub}}(a)$ returns a normalized expression for $a$ and a substitution $\text{sub}'$; $\text{NormE}_{bt,ck}(e)$ returns a normalized expression for $e$ annotated with $bt$ and $ck$ and a substitution $\text{sub}'$. $\text{NormCA}_{\text{sub}}(a)$ returns a normalized control expression for $a$ and a new substitution $\text{sub}'$. $\text{NormD}_{\text{sub}}(D)$ normalizes the definitions $D$ and returns a new substitution $\text{sub}'$. Given $D$ and $\text{NormD}_3(D) = D'[a_1/x_1, \ldots, a_n/x_n]$, it is easy to write a function that checks that $D$ and $x_1 = a_1$ and ... $x_n = a_n$ and $D'$ are indeed convertible: it consists in applying the substitution to $D'$ and to check that the result equals $D$.

Remark 1 (Intermediate language.). It would be possible to introduce a new intermediate language for normalized expressions instead of staying in the same language. This is essentially a matter of taste. When proving the correctness of the translation, a source-to-source transformation has the advantages to reduce the number of languages to formalize. On the other hand, it requires carrying the invariant that a program is normalized. A variation is to generate an intermediate language where only equations (instead of every expression) are annotated with a clock $ck$ interpreted as a guard. The principle of the code generation would stay unchanged nonetheless.

3.2. Static Scheduling

Once a program has been normalized, we need to find a static evaluation order to compute the equations. Following the definition given in [7], an expression $a$ statically depends on $x$ if $x$ appears free in $a$ and not on the right of a delay fby. $\text{Left}(a)$ returns the set of such variables. We overload the notation for $\text{Left}(e)$ and $\text{Left}(D)$. The transitive closure of this relation defines the notion of static dependency. The corresponding graph must be acyclic and the program is said causally correct; otherwise, the compilation stops. Definitions are given in Figure 4.

Once normalized, the equations must be statically scheduled according to the dependencies between them. Nonetheless, in preparation for the code generation step, the scheduling must take into account the difference between local variables (defined by an equation $x = a$) and state variables (defined by an equation $x = (v \text{ fby } a)^{ck}$).

Definition 3 (Sequential Order between Normalized Equations).
An equation reading \(x\) is the set of (write) variables defined by
\(\{x\}\).

Left \((v \text{ fby } a)\) = \(\emptyset\).

Left \((a_1, \ldots, a_n)\) \(= \bigcup_{1 \leq i \leq n} \text{Left} (a_i)\).

Left \((x)\) = \(\{x\}\).

Left \((v)\) = \(\emptyset\).

Left \((\text{merge } x (C_1 \rightarrow a_1) \ldots (C_n \rightarrow a_n))\) = \(\bigcup_{1 \leq i \leq n} \text{Left} (a_i) \cup \{x\}\).

Left \((a \text{ when } C)\) = \(\{x\} \cup \text{Left} (a)\).

Left \((x = a)\) = \(\text{Left} (a)\).

Left \((\bar{x} = f (a_1, \ldots, a_n) \text{ every } a_0)\) = \(\bigcup_{0 \leq i \leq n} \text{Left} (a_i)\).

\[\text{Def} (D_1 \text{ and } D_2) = \text{Def} (D_1) \cup \text{Def} (D_2)\]

\[\text{Def} (x = e) = \{x\}\]

\[\text{Vars} (\text{base}) = \emptyset\]

\[\text{Vars} (\text{ck on } C) = \text{Vars} (\text{ck}) \cup \{x\}\]

\[\text{NA}(a) = \emptyset\]

\[\text{NA}(a) = \text{Left} (a) \cup \text{Vars} (a)\]

\[\text{NA}(a) = \emptyset\]

\[\text{NA}(a) = \text{Left} (a) \cup \text{Vars} (a)\]

\[\text{NA}(a) = \emptyset\]

Figure 4: Syntactic Dependencies

1. An equation \(x = a\) defines the value of a local variable and it must be scheduled before all
   equations that read \(x\). This is the regular data-dependency.

2. An equation \((v \text{ fby } a)\) defines a state variable \(x\) and it must be scheduled after all
   equation reading \(x\). This corresponds to an anti-dependency.

**Definition 4 (Scheduling equivalence).* Two collections \(D_1\) and \(D_2\) are schedule equivalent,
written \(D_1 \overset{\text{SCH}}{\approx} D_2\), when they are equal up to a permutation of their equations.

\[
D \overset{\text{SCH}}{\approx} D \quad \text{eq} \text{ and } \text{eq} \text{ and } \text{eq} \text{ and } \text{eq} \text{ and } \text{eq} \quad \frac{D \overset{\text{SCH}}{\approx} D'}{D \overset{\text{SCH}}{\approx} D'}
\]

\[
D_1 \overset{\text{SCH}}{\approx} D_2 \text{ and } D_2 \overset{\text{SCH}}{\approx} D_3 \quad \frac{D_1 \overset{\text{SCH}}{\approx} D_3}{D_1 \overset{\text{SCH}}{\approx} D_3}
\]

Given \(D_1\) and \(D_2\), writing a function that checks that \(D_1 \overset{\text{SCH}}{\approx} D_2\) holds is trivial. A static schedule of a sequence of equations is obtained from a topological sort according to the order given in Definition 3. We characterize equations that are both normalized and scheduled. We say that such equations are well-formed.

**Definition 5 (Sequentially Ordered Equations).* An equation \(D\) is scheduled when there exist
sets of names \(r, w, \text{mem}\) such that \(\text{SCH}(D) : r, w, \text{mem}\) holds, where \(r\) is the set of read
variables from \(D\), \(w\) is the set of (write) variables defined by \(D\), \emph{mem} is the set of memories, and
the predicate is defined:

\[
\text{SCH}(x = a) : \text{Left} (a), \{x\}, \emptyset \quad \text{NA}(a)
\]

\[
\text{SCH}(x = (v \text{ fby } a)_{\text{ck}}) : \text{Left} (a), \emptyset, \{x\}
\]

\[
\{x_1, \ldots, x_m\} \cap (\bigcup_{0 \leq i \leq n} \text{Left} (a_i)) = \emptyset \quad \forall 0 \leq i \leq n, \text{NA}(a_i)
\]

\[
\text{SCH}(\bar{x} = f (\bar{a}) \text{ every } a_0) : \bigcup_{0 \leq i \leq n} \text{Left} (a_i), \{x_1, \ldots, x_m\}, \emptyset
\]

\[
\text{SCH}(\text{eq}) : r, w, \text{mem} \quad \text{SCH}(D) : r', w', \text{mem'} \quad w' \cap r = \emptyset \quad \text{mem} \cap r' = \emptyset
\]

\[
\text{SCH}(\text{eq and } D) : r \cup r', w \cup w', \text{mem} \cup \text{mem'}
\]
For short, we write \( \text{SCH}(D) \) when there exist \( r, w \) and \( m \) such that \( \text{SCH}(D) : r, w, m \).

The property of being scheduled means that an equation never reads a non-state variable before it is written, and that no state variable is modified before being read. Note that checking that a sequence of equations is well formed is straightforward so the scheduling process can be programmed in any (untrustworthy) language.

**Remark 2 (Extra copy variables).** Even when a collection of equations is causally correct, it may be necessary to introduce auxiliary variables so as to schedule it. Consider the two equations (omitting annotations) on the left, below:

\[
\begin{align*}
x &= 0 \text{ fby } y \\
\text{and } y &= 1 \text{ fby } x
\end{align*}
\]

The two equations are in normal form but the sequence is not scheduled (according to \( \text{SCH}(.) \)): the first equation would have to be scheduled after the second and vice versa. One solution is to introduce an extra variable as shown on the right. This can be achieved by systematically adding an extra copy variable during the normalization process, e.g.:

\[
\text{NormE}_{ab,ck}^b(v \text{ fby } a) = \text{let } a', \text{sub}' = \text{NormA}_{ab}^c(a) \text{ in } \\
x, \text{sub}' + [(v \text{ fby } x')^c_{ab}/x, a'/x'] \text{ where } x, x' \notin \text{Dom}(\text{sub}')
\]

or by considering a more elaborate normalization function or as an extra pass which depends on data-dependences and add necessary copy variables. By simply characterising only the normal and scheduled forms, we give the compiler the liberty to consider several possible implementations of the normalization process.

### 3.2.1. Example

A possible static scheduling of Example 3.1.1 is given below.

\[
\begin{align*}
z &= ((t_1 + 3) \text{ when True}(c)) + k \\
\text{and } o &= (\text{merge } c (\text{True } \rightarrow t_2 + (2 \text{ when True}(c))) \\
&\quad \text{ (False } \rightarrow t_3 \text{ when False}(c)))^ck \\
\text{and } t_2 &= (5 \text{ fby } z)^ck \text{ on True}(c) \\
\text{and } t_3 &= (6 \text{ fby } x)^ck \\
\text{and } t_1 &= (4 \text{ fby } o)^ck
\end{align*}
\]

### 3.2.2. The Counting Example

After scheduling, the counting example becomes:

\[
\begin{align*}
v &= (\text{merge } top (\text{True } \rightarrow (1^b \text{ when True}(top))^ck_1) \\
&\quad (\text{False } \rightarrow (0^b \text{ when False}(top))^ck_4))^b \\
\text{and } o &= (\text{merge } tick (\text{True } \rightarrow (v^b \text{ when True}(tick))^ck_1) \\
&\quad (\text{False } \rightarrow ((t^b + v^b)^b \text{ when False}(tick))^ck_2))^b \\
\text{and } t &= (0 \text{ fby } o)^b
\end{align*}
\]

with \( ck_1 = b \text{ on True}(tick), ck_2 = b \text{ on False}(tick), \)
\( ck_3 = b \text{ on True}(top) \) and \( ck_4 = b \text{ on False}(top) \)

Whereas before scheduling equations were unordered, order is now meaningful: \( t \) has to be computed after \( o \), and \( v \) has to be computed before \( o \). Notice that the identification of the state done during normalization made the scheduling very simple by introducing the equation for \( t \).
3.3. Data-flow Optimizations

The data-flow nature of the language makes the implementation of classical graph-based optimizations, such as copy elimination, common sub-expression elimination or inlining simple. We do not consider them here.

4. A Simple Object-based Language

We now define an intermediate target language to represent the transition function obtained after compiling a set of equations. The objects of object-oriented programming are an efficient way of encapsulating a state and the collection of functions that manipulate it. We are not interested in inheritance or object polymorphism, but only in the capability to encapsulate a piece of memory managed exclusively by a set of methods. Here, we define a very simple object-based language (called Obc in the sequel). Adopting this point of view has two main advantages compared to a direct translation into a target language. First, object orientation is a well-known paradigm, and this may help to understand the basic principles of the first level of our transformation. Second, using it as a generic intermediate language allows one to derive a very simple translation to a target language like C or JAVA.

A stateful stream function or node can be compiled into a simple class definition with instance variables and two methods step and reset. State variables \( m \) are used to represent the internal state of the node (i.e., one for each delay) whereas instances \( j \) store the state of internal nodes. The method step inherits its signature from the node it was generated from, and it implements a single step of the node. The method reset is parameterless, and it is in charge of the initialization of the state variables. One difference with respect to object orientation is the absence of dynamic object creation; this is not necessary as we do not consider recursive definitions. The syntax of the language is given below.

\[
\begin{align*}
  d & := \text{class } f = \{ \text{memory} = m; \cr & \quad \text{instances} = j; \cr & \quad \text{reset} = S; \cr & \quad \text{step}(p) \text{ returns}(p) \text{ var } p \text{ in } S \} \\
  c & := x | \text{state}(x) | v \\
  v & := C | i \\
  j & := o : f, ..., o : f \\
  p & := x : bt, ..., x : bt \\
  m & := x : bt = v, ..., x : bt = v \\
  S & := x := c | \text{state}(x) := c | \text{skip} | S ; S \\
  & \mid \text{case } (c) \{ C : S_1 ; ... ; C : S_n \} \\
  & \mid \text{o.reset} \\
  & \mid (x, ..., x) = o.\text{step}(c, ..., c) \\
  L & := [S, ..., S]
\end{align*}
\]

A program consists of a sequence of global definitions \( d \) of classes. An instruction \( S \) may be an assignment to a local variable \( x := c \) or to a state variable \( \text{state}(x) := c \), a void statement \( (\text{skip}) \), a sequence \( (S ; S) \), a control structure \( \text{case } (c) \{ C_1 : S_1 ; ... ; C_n : S_n \} \), a re-initialization method invocation of an object \( o \) \( (o.\text{reset}) \), or an invocation of the step method of object \( o \) \( (o.\text{step}(c_1, ..., c_n)) \). The \( C_i \) must be pairwise different. If \( c \) is of type \( bt = C_1 + ... + C_i + ... + C_n \), we shall write indifferently \( \text{case } (c) \{ C_1 : \text{skip}; ... ; C_i : S ; ... ; C_n : \text{skip} \} \) or \( \text{case } (c) \{ C_1 : S \} \). An expression \( c \) can be either an access to a local variable \( x \) or to a state variable \( \text{state}(x) \), an immediate integer constant \( i \) or a value constructor \( (C) \), or a function call \( (\text{op}(c_1, ..., c_n)) \). A machine \( (f) \) defines a set of memories \( (m) \), a set of instances \( (j) \) for objects used inside the body of the methods and two methods reset and step.

Remark 3 (Adding more methods). Obc gives the minimal constructions for the code generation technique considered in this paper. Adding more methods would make it possible to give access to each component of a structured output (so as to avoid copying the output when calling a node, for example). It would also make Obc a target for static scheduling algorithms proposed in [10] and [11].

13
5. The Translation

We now present the main translation. It translates a scheduled program in normal form from MINILS into an OBC program.

5.1. Notations

We introduce the following notations.

1. If \( p = [x_1:bt_1;...;x_n:bt_n] \) and \( p_2 = [x'_1:bt'_1;...;x'_k:bt'_k] \) then \( p_1 + p_2 = [x_1:bt_1;...;x_n:bt_n;x'_1:bt'_1;...;x'_k:bt'_k] \) provided \( x_i \neq x'_j \) for all \( i, j \) such that \( 1 \leq i \leq n, 1 \leq j \leq k \). \([\]\) denotes the empty list of variable declarations.

2. In the same way, we write \( m_1 + m_2 \) for the composition of two substitutions on memory variables and \( j_1 + j_2 \) on object instances. \( p \setminus m \) removes declarations from \( p \) that already appear in \( m \).

3. If \( L_1 = [S_1,...,S_n] \) and \( L_2 = [S'_1,...,S'_k] \) are two lists of instructions, we write \( L_1 @ L_2 \) for the concatenation \( [S_1,...,S_n,S'_1,...,S'_k] \). \( \text{ConcatList}([S_1,...,S_n]) \) equals \( S_1;...;S_n \).

5.2. Control Structures

The translation makes a direct use of clock annotations: an equation \( x = \epsilon_{a,b}^c \) is executed if and only if \( ck \) evaluates to true. Thus, clocks in the source language become control structures in the target language. As an example, a computation \( S \) on clock base on \( C_1(x_1) \) on \( C_1(x'_1) \) is translated to an instruction case \( S \) { case \( C_1 : (x_1) \) \{ \( C_1 : S \} \}}.

In order to reduce the number of control structures, we make use of two functions: Control(\( \_ \_ \)) and Join(\( \_ \_ \_ \)). Given an instruction \( S \), Control(\( ck, S \)) returns a control structure such that \( S \) is executed only when \( ck \) is true.

\[
\begin{align*}
\text{Control(base}, S) & = S \\
\text{Control(} \text{ck on} C(x), S & = \text{Control(} \text{ck, case} \ (x) \ \{ \ C : S \})
\end{align*}
\]

If \( S_1 \) and \( S_2 \) are two instructions, Join(\( S_1, S_2 \)) joins the two according to the following definitions.

\[
\begin{align*}
\text{Join(} \\text{case} \ (c) \ \{ \text{case} \ (c) \ \{ C_1 : S_1;...;C_n : S_n \}, \text{case} \ (c) \ \{ C_1 : S_1;...;C_n : S'_n \}) & = \text{case} \ (c) \ \{ C_1 : \text{Join}(S_1,S'_1);...;C_n : \text{Join}(S_n,S'_n) \}) \\
\text{Join}(S_1,S_2;S_3) & = \text{Join}(S_1,S_2);S_3 \\
\text{Join}(S_1,S_2) & = S_1;S_2 \text{ otherwise}
\end{align*}
\]

JoinList([\( S \]) = S

JoinList([\( S_1,...,S_n \]) = \text{Join}(S_1,\text{JoinList}([S_2,...,S_n]))

Remark 4 (Control optimization). To get efficient sequential code, the scheduling function must gather equations activated by the same clock, as much as possible. This function can be arbitrarily complex and does not have to be proved correct nor developed in a safe language. The correctness of the compilation only rely on a straightforward validation function that verifies that two sequences of equations are equal up to a permutation and that the second one verifies Definition 5.

5.3. Definition of the Translation Function

The translation is defined by a set of mutually recursive functions. In the following, \( m \) stands for a memory environment, \( S \) an instruction that initializes the memory, \( j \) an environment for node instances, \( d \) an environment for local variables, and \( L \) a list of instructions. If \( NA(a), TE_m(a) \) defines the translation of an annotated expression \( a \) under \( m \). It returns an expression \( c \) in OBC.
$TE_m (c)$ defines the translation for an un-annotated normalized expression $e$. For non-normalized arguments, the translation is undefined.

$$TE_m (e^{ck}_{bt}) = TE_m (e)$$

$$TE_m (v) = v$$

$$TE_m + [x:bt = v] (x) = state (x)$$

$$5TE_m (x) = x \text{ otherwise}$$

$$TE_m (a \text{ when } C(x)) = TE_m (a)$$

$$TE_m (op(a_1, ..., a_n)) = let \{c_1, ..., c_n\} = TEList_m [a_1, ..., a_n] \text{ in } op(c_1, ..., c_n)$$

$TCA_m (x, a)$ defines the translation of an expression $a$ to store in variable $x$. $a$ is in normal form, that is, $NCA(a)$. Otherwise, the function is undefined.

$$TCA_m (y, merge x (C \rightarrow a)_{bt}^{ck}) = case (x) \{C : TCA_m (y, a)\}$$

$$TCA_m (x, a) = x := TE_m (e) \text{ otherwise}$$

$TEq_{m,S,j,L}(eq)$ defines the translation of an equation, which must be in normal form, that is, $NEQ(eq)$, otherwise the function is undefined. We use two auxiliary functions: the operation $TELList_m [a_1, ..., a_n]$ translates a list of expressions and returns a list of expressions in the target language. $TELList([D_1, ..., D_n])$ translates a list of equations.

$$TELList_m [a_1, ..., a_n] = [TE_m (a_1), ..., TE_m (a_n)]$$

$$TELList(eq) = TEq_{[\langle \text{skip} \rangle]} (eq)$$

$$TELList(eq \text{ and } D) = TEq_{TELList(D)} (eq)$$

The translation of a normalized equation is defined below.

$$TEq_{m,S,j,L}(x = e^{ck}_{bt}) = \langle m, S, j, [Control(ck, TCA_m (x, e^{ck}_{bt}))] @ L\rangle$$

$$TEq_{m,S,j,L}(x = (v \text{ fby } a)_{bt}^{ck}) =$$

$$\text{let } m' = m + [x : bt = v] \text{ in}$$

$$\langle m', state (x) := v; S, j, [Control(ck, state (x) := c)] @ L\rangle$$

$$TEq_{m,S,j,L}((x_1, ..., x_k) = f(a_1, ..., a_n) \text{ every } e^{ck}_{bt})$$

$$\text{let } c_0 = TE_m (e^{ck}_{bt}) \text{ in}$$

$$\text{let } [c_1, ..., c_n] = TELList_m [a_1, ..., a_n] \text{ in}$$

$$\langle m, o.reset; S, [o : f] + j$$

$$[Control(ck, case (c_0) \{[True : o.reset]\})] @ L\rangle$$

$$[Control(ck, (x_1, ..., x_k) = o.step(c_1, ..., c_n)] @ L\rangle \text{ where } o \notin Dom(j)$$

An equation $x = e^{ck}_{bt}$ is translated into an assignment to the local variable $x$ when $ck$ is true. A memory equation $x = (v \text{ fby } a)_{bt}^{ck}$ is translated into an assignment of the state variable $x$, executed when $ck$ is true. For a call to a node $f$, the translation introduces a fresh name $o$ which is an instance of the machine $f$. The initialization code consists in calling the $\text{reset}$ method of $o$. The code to execute for the node instance $o$ is thus (1) a call to its reset method when $c_0$ is true (that is, $o.reset$) followed by (2) a call to its step function $(x_1, ..., x_k) = o.step(c_1, ..., c_n)$). These two actions must be performed only when the clock $ck$ of $e_0$ is true. Finally, the code generation of a node consists in first scheduling the collection of equations and translate them sequentially.

$$TP(\text{node } f(p) \text{ returns } (q) \text{ var } d \text{ in } D) =$$

$$\text{let } \langle m, S, j, L \rangle = TELList(D) \text{ in}$$

$$\text{class } f = \langle \text{memory} = m;$$

$$\text{instances} = j;$$

$$\text{reset} = S;$$

$$\text{step}(p) \text{ returns } (q) \text{ var } d \text{ in } JoinList(L)\rangle$$

where $SCH(D)$
5.3.1. Example

The example given in paragraph 3.1.1 is translated into the sequence:

```plaintext
case (c){
  True: z := state(t1) * 3 + k; o := state(t2) + 2; state(t2) := z
  False: o := state(t3)
};
state(t3) := x; state(t1) := o
```

The initialization code is:

```plaintext
state(t2) := 5; state(t3) := 6; state(t1) := 4
```

This example shows the effect of scheduling decisions on the resulting code. Control optimization, i.e., fusion of control structures, requires putting equations together on the same clocks, provided data dependencies are respected. One could perform control optimization on the target code, but this would require rebuilding much of the data-flow information that is already available in the source program.

5.3.2. The Counting Example

The counter is translated into:

```plaintext
class counting = {
  memory t:int=0
  reset () = state(t1):=0
  step(tick:bool,top:bool) returns (o:int)
  var v:int in
  case (top){
    True:v:=1; False:v:=0
  }; case(tick){
    True:o:=v; False:o:=state(t) + v
  };
state(t):=o
}
```

5.3.3. The Activation Condition Example

We illustrate the compilation of the so-called activation condition building block of SCADE, once translated using merge and when. It is instantiated on a simple counter. The activation condition, implemented by the node condact below, takes an input i and a boolean sequence c. When c is true, it runs the node count on the current value of i. When c is false it returns its previous output:

```plaintext
node count(i : int) returns (o : int) o = (0 fby o) + i
node condact(c : bool; i : int) returns (o : int)
  var x : int in
  x = count(i when True(c))
  and o = merge c (True → x)(False → (0 fby o) when False(c))
```
After normalization and scheduling, the nodes `count` and `condact` are translated into:

```plaintext
class count = {
    memory t:int = 0
    reset () =
    state(t) := 0
    step(i:int) returns (o:int)
        o:= state(t) + i;
    state(t) := o
}
class condact = {
    memory t:int = 0
    instances o1:count
    reset () =
    o1.reset ();
    state(t) := 0
    step(c:bool;i:int) returns (o:int)
        var x:int in
        case(c) {
            True: x := o1.step(i);
            o := x;
            False: o := state(t)
        }
    state(t) := o
}
```

Note that `o1.step(i)` is only executed when `c` is true as must be expected. In the kernel language, the application `count(i when True(c))` should be first translated into the application `count(i when True(c)) everyFalse`. Then, either a special treatment of a false reset is taken into account or eliminated afterwards.

The condact example illustrates the memory model used for the generated code: it is essentially a tree structure: each machine allocates the memory for its sub-machines. There is no dynamic allocation of memory, thus conforming to current practice in safety critical embedded applications.

**Remark 5 (Elimination of copies).** As mentioned in [7], a stream `x` and its previous value `pre x` can be stored in the same variable if the computation of `x` is not followed by a use of `pre x`. This optimization is classical in compilers for synchronous languages and is a particular form of copy elimination [17]: remove a copy `t_1 := t_2` by storing `t_1` and `t_2` into the same location. As for control optimization (Remark 4), its applicability depends on the scheduling function. Consider the following sequence and suppose that the variable `y` is not read after equation (1).

$$
\begin{align*}
(1) & \quad x = y + 1 \\
(2) & \quad \text{... } x \text{ ...} \\
(3) & \quad \text{...} \\
(4) & \quad \text{and } y = 0 \text{ fby } x
\end{align*}
$$

The corresponding sequential code is given on the left below. On the right is the code where `x` is stored into `state(y)` (the dummy equation (4) will finally be removed).

$$
\begin{align*}
(1) & \quad x := \text{state}(y) + 1 \\
(2) & \quad \text{... } x \text{ ...} \\
(3) & \quad \text{...} \\
(4) & \quad \text{state}(y) := x
\end{align*}
\quad \begin{align*}
(1) & \quad \text{state}(y) := \text{state}(y) + 1 \\
(2) & \quad \text{...state}(y) \text{ ...} \\
(3) & \quad \text{...} \\
(4) & \quad \text{state}(y) := \text{state}(y)
\end{align*}
$$

This optimization can be ensured by reinforcing the invariant of Definition 5: for every equation `y = v fby x`, `y` must not appear free after the definition of `x`. The verification that the sequence of equations is well formed is simple and independent on the scheduling function.

### 6. Target code generation

The intermediate language of Section 4 is easily translated into either a full-fledged object-oriented language, such as **Java**, or into a low-level imperative language such as **C**. In this section we present both translations.
6.1. Translation into Java

As already pointed out, the intermediate language of Section 4 can be seen as a sequential language with the data encapsulation mechanism characteristic of object-oriented languages. As such, it lends itself to a straightforward translation into existing object-oriented languages such as Java.

Each machine definition is translated into a Java class definition with two methods step and reset. The state variables specified in the memory section are translated into member declarations. The instance variables specified in the instances section are translated into object creations using their default constructors. Actions and expressions are directly translated into the corresponding Java constructs. In case of multiple outputs, the return type of the step method is represented as a structure with the fields representing the subsequent elements of the tuple.

For instance, the counting example of Figure 1 is translated into the following Java code:

```java
public class counting {
    int t;

    public void reset() {
        t = 0;
    }

    public int step(boolean tick, boolean top) {
        int o; int v;
        if (top) { v = 1; } else { v = 0; }
        if (tick) { o = v; } else { o = t+v; }
        t = o;
        return o;
    }
}
```

6.2. Translation into C

For each machine, the state variables specified in the memory section and the instance variables specified in the instances section are gathered in a separate structure, used to represent the internal state of the machine. Both the reset and the step functions are translated into functions that accept an additional argument self, passed by reference, that points to a concrete instance of the corresponding state structure (object). If necessary, the return type of the step function is again represented as a structure to allow tuples to be returned.\(^5\) Actions and expressions are directly translated into the corresponding C constructs.

For instance, the counting example of Figure 1 is translated into the following C code:

```c
typedef struct {
    int t;
} counting_mem;

void counting_reset(counting_mem *self) {
    self->t = 0;
}

int counting_step(int tick, int top, counting_mem *self) {
    int o; int v;
    if (top) { v = 1; } else { v = 0; }
    if (tick) { o = v; } else { o = self->t + v; }
    self->t = o;
}
```

\(^5\)Note that it is also possible to store the output of a step function inside the internal memory of the object and to access it via auxiliary observation methods, to avoid complex copying.
7. A Complete Compiler

The content of the article up to this point can be used to implement a compiler for a full featured synchronous data-flow language in the spirit of Lustre. We have implemented such a compiler to serve as a reference implementation. In this section we give some details of this implementation and highlight possible extensions. Its organization is given in Figures 5 and 6.

Compiler Organization

Figure 5 and 7 give briefly the various steps of the compiler in the order they are applied with the corresponding number of lines of OCaml code. The MiniLS source is first typed and clocked, after which every expression is annotated with its type and clock. Programs that cannot be statically scheduled are then rejected by the causality check. This analysis follows the notion of static dependencies defined previously: every cycle in the dependence graph has to cross an explicit delay. Programs are normalized, then equations are scheduled and, finally, programs are translated into object-based code (Obc). The last step is a translation into target code (here C, Java, or Ocaml). Three optimizations are made directly on the clocked-data flow language: inlining, dead-code removal, and common sub-expression elimination. They are all defined as source-to-source transformations and benefit from the data-flow nature of the language.

Beside this reference compiler, another version is implemented in the programming language of Coq, but only up to the generation of the object-based intermediate language.

Scheduling and Data-flow Optimizations

In Section 5 we presented a control optimization that combines two consecutive control structures on the same guard. Other optimizations can be implemented in this translation, particularly around the scheduling policy. The scheduling transforms a partially ordered set of equations into a sequence of assignments. The solution is not unique in general, and we can take advantage of this freedom to favor certain optimizations. For instance, schedule consecutively equations that are guarded by the same clock, allowing more control conditions to be joined (Remark 4). Another classical optimization discussed in Remark 5 is related to the reuse of variables. The reference compiler we have developed to support the present article implements a scheduling heuristics for that...
Administrative code
abstract syntax + lexer & parser + printers 881
main driver (including symbol tables, loader, etc.) 285

Basis
graph structures 74
scheduling 67
type checking 269
clock checking 190
causality checking 30
normalization 95
control fusion 45
translation to the intermediate language 136

Emitters to concrete languages
(C, Java and OCaml) around 300 each

Optimizations
inlining 250
dead-code removal 42
data-flow network minimization 162

Language extensions
automata 107
control-structures 54
shared variables 59
reset conditions 199
translation to the basic clocked language 172

Figure 7: MiniLS in Numbers
purposes. We proceed in a simple way: in a first stage, copy variables are added systematically (cf. Remark 2) before static scheduling starts. Then, the scheduling heuristic tries to minimise both the number of opened control structures (Remark 4) and the number of copy variables (Remark 5).

**MiniLS as a Target Language**

The source language we consider is a first-order data-flow language similar to Lustre. Nonetheless, it exhibits specific constructions that make it both a good target for implementing extensions as well as a good input language for generating efficient sequential code. These constructions are the n-ary merge (instead of the current operator of Lustre) and a modular reset construct (absent in Lustre). They can both be encoded in regular Lustre but the generated code is then inefficient or calls for complex optimization techniques to cancel the effects of the encoding. Providing merge and reset as basic primitives allows for a more direct and efficient compilation.

In [13], an extension of Lustre with hierarchical state automata has been proposed and its compilation is done through a translation into a clocked data-flow kernel similar to the one considered in this article. The merge and reset constructs were used in this encoding to express the control structure of automata in a purely data-flow setting. This encoding not only gives a semantics to the whole, it also proves to be an effective way to design the compiler in the sense that the generated code is good in terms of size and efficiency. These language extensions have been implemented on top of the MiniLS compiler.

8. Semantics

This section starts the second part of the paper. It formally defines the semantics for the source and target languages and proves that the translation presented in Section 5 preserves the semantics.

The preservation is only guaranteed for programs that have passed all the compilation steps: those that are well typed, well clocked, and normalized. That is, those that can be statically scheduled and translated into sequential code. This is consistent with the functioning of a compiler, which stops on errors.

8.1. Synchronous Semantics

We first define a semantics for the annotated source synchronous data-flow language. First, we give the definition of the values and environments used to define the semantics. Second, we give a stream semantics to the synchronous primitives fby, when and merge and show how scalar constants and operators are lifted to streams. We also give a stream interpretation to clocks. Third, we define the stream operations that are necessary to give a semantics to the reset.

8.1.1. Values and Environments

A value $v$ is either an immediate constant, e.g., an integer, or an element of an enumerated type. A tagged value $w$ is either a value $v$ or the special absent value $\text{abs}$. An $s$ stands for a sequence of tagged values.

$$ v ::= C \mid i \quad w ::= \text{abs} \mid v \quad s ::= \epsilon \mid w.s $$

If $s$ is a sequence, $s^i$ is its $i$th element (if it exists) and is such that $\epsilon^i$ is undefined, $(w.s)^1 = w$ and $(w.s)^i = s^{i-1}$. The concatenation of two sequences $s_1$ and $s_2$ is written $s_1 \bullet s_2$. An instantaneous environment associates tagged values $(w)$ to variables. A synchronous environment is called a *history*. A history $H$ is a sequence $H^1.H^2...$ of instantaneous environments $R$. The global environment $G$ contains the definition of all the nodes in the program. This environment is static. It is defined once and for all at the very beginning of an execution.

$$ R ::= [w_1/x_1, ..., w_n/x_n] \text{ with } \forall i, j. (i \neq j \Rightarrow x_i \neq x_j) $$

$$ H ::= \epsilon \mid R.H $$

$$ G ::= [\langle p_1, q_1, d_1, D_1 \rangle/f_1, ..., \langle p_n, q_n, d_n, D_n \rangle/f_n] \text{ with } \forall i, j. (i \neq j \Rightarrow (f_i \neq f_j) $$
The composition and access to environments are defined below:

1. If $R$ is an environment, $\text{Dom}(R)$ is the set of names defined by $R$.
2. the composition $R_1 * R_2$ of two environments is defined provided names do not conflict ($\text{Dom}(R_1) \cap \text{Dom}(R_2) = \emptyset$). If $x \in \text{Dom}(R_1)$, then $(R_1 + R_2)(x) = R_1(x)$ and if $x \notin \text{Dom}(R_1)$, then $(R_1 + R_2)(x) = R_2(x)$. If $\vec{s} = (s_1, ..., s_n)$, $\vec{x} = (x_1, ..., x_n)$ and $p = x_1 : t_1; ...; x_n : t_n$ then $R + [\vec{s}/\vec{x}] = R + [s_i/p] = R + [s_1/x_1] + ... + [s_n/x_n]$.
3. If $R$ is an environment, $(R + [w/x])(y) = w$ if $x = y$ and $(R + [w/x])(y) = R(y)$ otherwise. $R(y) = \text{abs}$ if $y \notin \text{Dom}(R)$. If $\vec{x} = (x_1, ..., x_n)$ and $p = x_1 : t_1; ...; x_n : t_n$ then $R(\vec{x}) = R(p) = (R(x_1), ..., R(x_n))$.
4. If $H_1$ and $H_2$ are two histories of the form $H_1 = (R_{11}, H'_{11})$ and $H_2 = (R_{21}, H'_{21})$, the composition $H_1 * H_2$ is defined as $H_1 * H_2 = (R_{12}, H'_1 + H'_2)$. If $H$ is a history, $H + \epsilon = \epsilon + H = \epsilon$.
5. If $H$ is a history, $H = R.H'$, then $H(x) = w_x.H'(x)$ where $w_x = R(x)$. If $H$ is a history, and $H = \epsilon$, $H(x) = \epsilon$. If $\vec{s} = (s_1, ..., s_n)$, $\vec{x} = (x_1, ..., x_n)$ and $p = x_1 : t_1; ...; x_n : t_n$ then $H + [\vec{s}/\vec{x}] = H + [s_i/p] = H + [s_1/x_1] + ... + [s_n/x_n]$ and $H(\vec{x}) = H(p) = (H(x_1), ..., H(x_n))$.
6. If $G$ is the global environment, $G(f)$ returns the definition associated to $f$ in $G$. The declaration $\text{node } f(p_1) \text{returns } (p_2) \text{ var } p_3 \text{ in } D$ of $f$ extends $G$ into $G + [(p_1, p_2, p_3, D)/f]$.
7. The prefix order $\leq$ between sequences is such that for all $s_1, s_2, w, \epsilon \leq s$ and $w.s_1 \leq w.s_2$ if $s_1 \leq s_2$.

8.1.2. Synchronous Stream Primitives

We define stream primitives in figure 8. $\text{const}^\#(v)$ defines the semantics of an immediate constant $v$. $\text{const}^\#(v)$ takes a boolean sequence, an immediate value $v$ and produces a constant
sequence with value $v$. $\text{lift}_{s}^{#}(f)(s_{1}, ..., s_{n})$ is a lifting operator: it applies a combinatorial function $f$ point-wise to its arguments. All arguments must be synchronous – all present or absent at each instant – otherwise, no reaction is possible. A delay $\text{fby}_{s}^{#}(s)$ causes the initialization value $v_{0}$ to its input. The semantics of the sampling operation $\text{when}_{C_{1}}^{#}(s_{1}, s_{2})$ is to return the current value of $s_{1}$ when $s_{2}$ equals $C$, otherwise, it returns the absent value. $s_{1}$ and $s_{2}$ must also be synchronous. $\text{merge}_{s}^{#}(s, (C_{1} \rightarrow s_{1}) ... (C_{n} \rightarrow s_{n}))$ returns the current value of $s_{i}$ when the current value of $s$ is $C_{i}$. In that case, all other branches must produce an absent value. Note that the semantics of these primitives is partial: non synchronous programs are not given a semantics.

We define an interpretation for clocks in figure 9. $\text{base}_{H}^{#}$ is the interpretation of the base clock in an environment $H$: it is a sequence of true values with the same length as $H$. $\text{clock}(s)$ returns the boolean clock of a sequence $s$. The current value of $\text{clock}(s)$ is true when the current value of $s$ is present. An expression $e$ with clock type $ck$ on $C(x)$ is present when $x$ is present and it equals $C$. If $b$ is the interpretation of a clock $ck$ then $\text{on}_{C}^{#}(b)(x)$ is a boolean sequence whose $n$-th value is true when the $n$-th value of $b$ is true and the $n$-th value of $x$ is $C$. If $H$ is an environment:

$$H(\text{base}) = \text{base}_{H}^{#}$$

$$H(\text{ck on } C(x)) = \text{on}_{C}^{#}(H(ck))(H(x))$$

### 8.1.3. Sequence of Behaviors

**Definition 6 (Sequence).** Given a stream of (present or absent) boolean values $r$ (for reset), a stream of present values $b$ (for before) and a stream of (present or absent) values $a$ (for after), $\text{seq}(0)(r)(b)(a)$ returns a stream $t$ (for total) of (present or absent) values, such that $t$ is equal to the concatenation of $b'$ and $a$. $b'$ is the stream $b$ where absent values have been inserted wherever there are absent values in $r$. The first argument of $\text{seq}(\cdot)(\cdot)(\cdot)(\cdot)$ is used to discard the reset at the very first instant. The definition is given in Figure 10.

<table>
<thead>
<tr>
<th>$b$</th>
<th>$b_{1}$</th>
<th>$b_{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$a_{3}$</td>
<td>$a_{4}$</td>
</tr>
<tr>
<td>$r$</td>
<td>$r_{1}$</td>
<td>abs</td>
</tr>
<tr>
<td>$\text{seq}(0)(r)(b)(a)$</td>
<td>$b_{1}$</td>
<td>abs</td>
</tr>
</tbody>
</table>
If $H = [s_1/x_1, ..., s_n/x_n]$ and $H' = [s'_1/x_1, ..., s'_n/x_n]$, $\text{seq}(0)(r)(H)(H')$ is an environment such that:

$$\text{seq}(0)(r)(H)(H') = [\text{seq}(0)(r)(s_1)(s'_1)/x_1, ..., \text{seq}(0)(r)(s_n)(s'_n)/x_n]$$

In the sequel, $\text{seq}(r)(s)(s')$ is a short-cut for $\text{seq}(0)(r)(s)(s')$ and $\text{seq}(r)(H)(H')$ for $\text{seq}(0)(r)(H)(H')$.

**Lemma 1 (Distributivity of the Sequence).** For any environment $H_1, H'_1, H_2, H'_2$ such that $\text{Dom}(H_1) = \text{Dom}(H'_1)$, $\text{Dom}(H'_2) = \text{Dom}(H'_2)$, and $\text{Dom}(H_2) = \text{Dom}(H'_2)$, the following property holds:

$$(H_1 + H_2 = \text{seq}(r)(H'_1 + H'_2)(H''_1 + H''_2)) \iff (H_1 = \text{seq}(r)(H'_1)(H''_1)) \land (H_2 = \text{seq}(r)(H'_2)(H''_2))$$

**Proof:** By induction on $H_1, H_2$. 

**Remark 6 (Reset).** Sequential operators are used to give a semantics to the reset of an application. $\vec{y} = f(\vec{x})$ every $r$ resets the execution of $f(\vec{x})$ every time $r$ is true. The reset operator acts as a recursive function: it executes $\vec{y} = f(\vec{x})$ while $r$ is false then it behaves like $\vec{y} = f(\vec{x})$ for the remaining instants.

The ability to reset a node instance exists in SIMULINK but not in LUSTRE. The reset makes the data-flow semantics of MINILS more complex than that of LUSTRE. Nonetheless, it is a mandatory feature to encode richer control-structures such as automata and this is why we consider it in the present formalization.

### 8.1.4. Semantics of the Source Language

We define the semantics of the source language using four predicates:

1. $G, H \vdash e \downarrow s$ : in a global environment $G$ and a history $H$, an expression $e$ produces a sequence $s$.
2. $G, H \vdash a \downarrow s$ : in a global environment $G$ and a history $H$, an annotated expression $a$ produces a sequence $s$.
4. $G, H \vdash r D \downarrow H'$ : in a global environment $G$, a history $H$ and a reset sequence $r$, $D$ produces $H'$.

We write $H \vdash a \downarrow s$ when $\vec{a} = (a_1, ..., a_n)$, $\vec{s} = (s_1, ..., s_n)$ and $H \vdash a_i \downarrow s_i$, for $1 \leq i \leq n$. All the definitions assume the existence of a global environment $G$ which is left implicit for simplicity. The definition of the predicates is given in Figure 11.

- The first five rules consist in applying the primitive operators defined previously ($(\text{LIFT})$ to $(\text{FBV})$).
- The rule for $D_1$ and $D_2$ means that $D_1$ and $D_2$ are run in parallel provided their defined names do not interfere (rule $(\text{AND})$).
- The rule $(\text{EQ})$ states that the semantics of $x = a$ is a history $[s/x]$ if $a$ evaluates to $s$.
- The semantics of an application $\vec{y} = f(\vec{a})$ every $a'$ is given by the rule $(\text{APP})$. If $\vec{a}$ evaluates to $\vec{s}$ and the body $D$ of $f$ executed in $[\vec{s}/p]$ produces a history $H'$ then the application returns a tuple of sequences $H'(q)$. For that, we use the auxiliary predicate $\vdash r$ where $r$ defines the instant where a reset occurs.
- The behavior of a node application is the following:
  - If $r$ is always false then the collection of equations $D$ is run infinitely often.
  - Otherwise, the environment $H$ can be decomposed into the concatenation of two environments $H_1$ and $H_2$ such that $\text{seq}(r)(H_1)(H_2)$. $r$ is such that $r = \text{seq}(r)(r_1)(r_2)$ for some $r_1$ and $r_2$. $D$ is run with $H_1$ and it produces some $H'_1$. When the reset applies, a new run is made on the remaining input environment $H_2$. 

24
8.2. An Operational Semantics for the Intermediate Language

We now define a semantics for the intermediate language. The proposed semantics is operational.

8.2.1. Values, Environments and Memories

An environment \( \rho \) assigns values \( v \) to variable names, values are defined as before. Note that these are simple values, and not tagged values: there is no explicit notion of absence here.

\[
\begin{align*}
\text{(LIFT)} & \quad H \vdash \bar{a} \Downarrow \bar{s} \quad \Rightarrow \quad s = \text{lift}_{\bar{a}}^{\bar{s}}( op ) \\
\text{(MERGE)} & \quad H \vdash x \Downarrow s \quad \Rightarrow \quad H \vdash \text{merge} \ ( C \rightarrow a ) \Downarrow s' \\
\text{(FBY)} & \quad H \vdash v \Downarrow s \quad \Rightarrow \quad H \vdash \text{fby} \ ( e ) \Downarrow s' \\
\text{(VAR)} & \quad H \vdash x \Downarrow H(x) \quad \Rightarrow \quad H \vdash a \Downarrow s \quad \Rightarrow \quad s' = \text{var}( C, a ) \Downarrow s' \\
\text{(CONST)} & \quad H \vdash v_i \Downarrow \Rightarrow \quad H \vdash \text{const}( v_i ) \Downarrow \\
\text{(WHEN)} & \quad H \vdash x \Downarrow s \quad \Rightarrow \quad H \vdash \text{when}( s, H(x) ) \\
\text{(APP)} & \quad H \vdash \bar{a} \Downarrow \bar{s} \quad H \vdash a' \Downarrow r \quad \Rightarrow \quad H' = [ \bar{s}/p ] \Downarrow r \quad \text{every } a' \Downarrow [ H'(p')/\bar{y} ] \\
\text{(SEQ)} & \quad H_1 \vdash D \Downarrow H'_1 \quad H_2 \vdash D \Downarrow H'_2 \quad \Rightarrow \quad \text{seq}( r )( H_1 ) ( H_2 ) \Downarrow \text{seq}( r )( H'_1 ) ( H'_2 ) \\
\end{align*}
\]

Figure 11: Synchronous Semantics

A program consists of a sequence of machine definitions which define an initial global environment \( G_c \):

\[
G_c \ ::= \ [ g_{v_1}/f_1, \ldots, g_{v_k}/f_k ] \quad \text{where } f_i \neq f_j \text{ for all } i \neq j
\]
The global value $gv$ associated to a global name $f$ resembles a class made of state and instance variables, a step and reset method. We also define sequences of environments and memories:

$$h ::= e | \rho.h \quad hm ::= e | M.hm$$

We define compositions of environments and accesses:

- If $\rho_1$ and $\rho_2$ are two environments and $\text{Dom} \rho_1 \cap \text{Dom} \rho_2 = \emptyset$, the composition $\rho_1 + \rho_2$ is such that $(\rho_1 + \rho_2)(x) = \rho_1(x)$ is $x \in \text{Dom} \rho_1$ and $\rho_2(x)$ otherwise.

- If $M$ is a state, $M(x)$ is the value of the state variable $x$ whereas $M(o)$ is the instance value of $o$ if $M = \langle mv, jv \rangle$, $M(x) = mv(x)$ and $M(o) = jv(o)$.

- We define an environment $\rho$ such that $(\rho + 1)\rho$ means that under state $M$, the expression $e$ evaluates to $v$.

- If $o$ is an object $\langle M, f \rangle$, we define $\text{StateCode}(o) = M$, $\text{ResetCode}(o) = S_1$ and $\text{StepCode}(o) = \langle p, q, d, S_2 \rangle$ if $G_c(f) = \langle m, j, \text{reset} = S_1, \text{step} = \langle p, q, d, S_2 \rangle \rangle$.

- We write $\langle o \text{ with } M' \rangle$ for $\langle M', \text{reset} = S_1, \text{step} = \langle p, q, d, S_2 \rangle \rangle$, that is, a copy of $o$, replacing $M$ by $M'$.

- If $M = \langle mv, jv \rangle$, $\text{Mem}(M) = mv$ and $\text{Instances}(M) = jv$.

- If $G_c$ is a global environment, $G_c(f)$ returns the definition associated to $f$ in $G_c$.

The declaration:

$$\text{class } f = \langle \text{memory } = m; \text{ instances } = j; \text{ reset } = S; \text{ step } = \langle p, q, d, S \rangle \rangle$$

The semantics is functional: when executed in a given state and a given environment, the new environment and state are unique.

**Lemma 2 (Uniqueness of a Reaction).** The following two properties hold:

1. $\forall M, \rho, c, v_1, v_2. (M, \rho \vdash c \downarrow v_1) \land (M, \rho \vdash c \downarrow v_2) \Rightarrow (v_1 = v_2)$
2. $\forall M, M_1, M_2, \rho, \rho_1, \rho_2, S. (M, \rho \vdash S \downarrow M'_{1}, \rho_1) \land (M, \rho \vdash S \downarrow M_2, \rho_2) \Rightarrow (M_1 = M_2) \land (\rho_1 = \rho_2)$

**Proof:** By structural induction on $c$ and $S$. \qed

**8.2.3. Memory Allocation and Reset**

The allocation $\text{Alloc}(m, j)$ defines the initial state of the system.

$$\text{New}(f) = \langle \text{Alloc}(m, j), f \rangle$$

if $G_c(f) = \langle m, j, \text{reset} = S_1, \text{step} = \langle p, q, d, S_2 \rangle \rangle$.

$$\text{Alloc}(m, j) = \langle \text{AllocStates}(m), \text{AllocInstances}(j) \rangle$$

$$\text{AllocStates}(\{ x_1 : bt_1 = v_1, \ldots, x_n : bt_n = v_n \}) = \langle v_1/x_1, \ldots, v_n/x_n \rangle$$

$$\text{AllocInstances}(\{ f_1/o_1, \ldots, f_k/o_k \}) = \langle \text{New}(f_1)/o_1, \ldots, \text{New}(f_k)/o_k \rangle$$

**Definition 7 (Reset).** A triple $m, j, S_1$ is well-formed if $S_1$ reproduces the initial state, i.e.:

$$\text{WFReset}(m, j)(S) \overset{\text{def}}{=} \forall \rho, M, \rho', M_0. \rho \vdash S \downarrow M_0, \rho' \Rightarrow M_0 = \text{Alloc}(m, j)$$
8.2.4. Program Execution

Given a program — a collection of machine definitions — execution starts by building the environment $G_c$ containing all the definitions. Now executing the main machine consists in creating an instance $o$ of this machine, then iteratively executing its step function in a global environment. At each step, the internal memory of the machine is updated, and a new environment is returned. This is expressed by the following definition of the function $Run$:

\[
Run_M(S)(e) = \epsilon \\
M, \rho \vdash S \downarrow M', \rho' \quad Run_M(S)(h) = h' \\
Run_M(S)(\rho, h) = \rho', h'
\]

If the instance $o$ created for the main machine is such that $o = (Alloc(m, j), f)$ with $G_c(f) = \langle m, j, reset = S_1, step = \langle p, q, d, S_2 \rangle \rangle$, then executing the machine in an initial memory $M$ and a sequence of environments $h$ defining values for all the variables in $p$ returns the sequence of environments $h'$ such that $h' = Run_M(S_2)(h)$.

9. Proof of Correctness

Before proving the correctness of the translation, we introduce some properties of the semantics. These properties are used later in the proof of the translation.

9.1. Clock Coherency

We prove the adequacy of the static clock annotations defined in Section 2.3 with respect to the semantics: for any annotated expression $e_{ck}^k$, if $e$ produces a stream $s$ in an environment $H$, then $H(ck) = clock(s)$. Two auxiliary lemma are necessary.
Lemma 3 (Clock of When). The following property holds:
\[ \forall s_1, s_2. \text{clock}(s_1) = \text{clock}(s_2) \Rightarrow \text{clock}(\text{when}_C(s_1, s_2)) = \text{on}_C(\text{clock}(s_1))(s_2) \]

Lemma 4 (Clock of Merge). The following property holds:
\[ \forall s_1, \ldots, s_n, x. (\forall 1 \leq i \leq n. \text{clock}(s_i) = \text{on}_C(\text{clock}(x))(x)) \Rightarrow \text{clock}(\text{merge}_C(x, (C_1 \rightarrow s_1) \ldots (C_n \rightarrow s_n))) = \text{clock}(x) \]

Definition 8 (Compatibility). Let \( H \) denote a clock environment and \( H \), an environment of values sharing the same domain. \( H \|= H \) states that \( H \) is compatible with \( H \):
\[ (H \|= H) \triangleq \forall x \in \text{Dom}(H). \text{clock}(H(x)) = H(H(x)) \]

We now state the main property that the clock annotations of terms should satisfy.

Theorem 1 (Clock Coherency). The following properties hold:
1. \( \forall H, e, ck, H, s. (H \vdash e_{ck} : ck') \land (H \|= H) \land (H \vdash e \downarrow s) \Rightarrow (ck = ck') \land (\text{clock}(s) = H(ck)) \)
2. \( \forall H, H', H, H', D. (H \|= H') \land (H \vdash D : H') \land (H \vdash D \downarrow H') \Rightarrow (H' \|= H') \)

The first property states that whenever an environment \( H \) is compatible with a clock environment \( H \), an expression \( e \) with clock \( ck \) produces a sequence which is present iff the current value of \( ck \) is true. The second one states that if \( H \) is compatible with \( H \) and \( D \) produces an environment \( H' \) then \( H' \) is compatible with the clock environment \( H' \). This property states that clock annotations are coherent with the clock constraints.

Proof:
1. By induction on \( e \). For synchronous primitives for which the clock of input and output are the same, it is easy to check that the property holds. For filtering primitives (when and merge), we apply Lemma 3 and 4.
2. By induction on \( D \).

We can now prove the semantics preservation of the compilation method. The translation is done in several steps, and we need to prove each of these steps. We have two types of transformation to consider:

1. source-to-source transformations — namely the normalisation and scheduling steps. They can also include dataflow optimizations such as common sub-expression elimination. Such optimizations are beyond the scope of this paper but can be added as needed, and their correctness shown independently.
2. the translation itself; from a normalized and scheduled source program to object code.

9.2. Correctness of Source-to-source Transformations

In the organization of the compiler, once typed, clocked and proved to be causal, source expressions are translated into annotated expressions and we only give a semantics to programs which have passed these three steps. A program that cannot be put in a scheduled normal form (Definition 5) is said to be not causal. This is the case for the equation \( x = x + 1 \), for example.

We first formally define a notion of equivalence between source programs, then prove that both normalization and scheduling are correct.

Definition 9 (Equivalence).
1. Two expressions \( a_1 \) and \( a_2 \) are equivalent when they produce the same sequences of values, i.e.:
\[ a_1 \approx a_2 \triangleq \forall H, s. (H \vdash a_1 \downarrow s) \Leftrightarrow (H \vdash a_2 \downarrow s) \]
2. Two declarations $D_1$ and $D_2$ are equivalent when they produce the same sequences of environments:

$$D_1 \approx D_2 \overset{\text{def}}{=} \forall H, H'. (H \vdash D_1 \Downarrow H') \iff (H \vdash D_2 \Downarrow H')$$

3. Two declarations $D_1$ and $D_2$ are equivalent on $\text{dom} = \{x_1, \ldots, x_n\}$ when they produce the same values for any $x \in \{x_1, \ldots, x_n\}$.

$$D_1 \approx_{\text{dom}} D_2 \overset{\text{def}}{=} \forall H, H'. \exists H''.
\begin{align*}
(H \vdash D_1 \Downarrow H') & \Rightarrow (H \vdash D_2 \Downarrow H'') \\
\forall x \in \text{dom}, (H'(x) = H''(x))
\end{align*}$$

4. Two node definitions:

$$\text{node } f_1(p) \text{ returns } (q) \text{ var } p_1 \text{ in } D_1$$

and

$$\text{node } f_2(p) \text{ returns } (q) \text{ var } p_2 \text{ in } D_2$$

are equivalent if they produce the same outputs when receiving the same inputs:

$$f_1 \approx f_2 \overset{\text{def}}{=} D_1 \approx_{\text{Dom}(q)} D_2$$

We have the following two lemmas:

**Lemma 5 (Normalization).** The normalization preserves the semantics.

$$\forall D, D_N. (D_N \in \text{Norm}(D)) \Rightarrow (D_N \approx_{\text{Def}(D)} D)$$

**Proof:** To prove this result, we first show that the semantics is preserved by substitution. We then show that by applying a substitution to the normalized program we obtain the source program.

**Lemma 6 (Scheduling).** Scheduling preserves the semantics, i.e.:

$$\forall D_S, D. (D_S \in \text{Sch}(D)) \Rightarrow (D_S \approx D)$$

**Proof:** The only change during scheduling is the order in which definitions appear in a declaration $D$. We show that the semantics is preserved by any permutation of equations (predicate $D_1 \overset{\text{SCH}}{\approx} D_2$). This is made easy by the fact that $H + [s/x]$ is such that $x$ is not already defined in $H$ and thus, the order in which the environment is extended does not matter.

### 9.3. Correctness of the Translation Function

Intuitively we show that when a node is translated, executing it using the synchronous semantics given in Section 8.1 on a set of inputs, and executing the object code which is the result of its translation on the same set of inputs and using the operational semantics presented in Section 8.2 yields the same results.

We first define several notions of compatibility that relate elements of the source language and its semantics and elements of the intermediate language and its semantics. These notions of compatibility are central to the proof of correctness. Next, we prove an important lemma about sequential behavior. Finally, we can prove the property itself. To do this, for each syntactic class in the source language we prove that an element of this syntactic class is compatible with its translation.
9.3.1. Compatibilities

We need to relate the environments used for evaluation by each semantics. To do this, we introduce notions of compatibility between environments. The main difference between environments $H$ and $h$ is that there is no representation of the absent value in $h$. Compatibility expresses that, for an environment $R$ and a memory $\rho$, any name defined in $R$ and associated to a value different from $\text{abs}$ is defined with the same value in $\rho$. It also imposes that any name defined in $\rho$ is also defined in $R$. Absent values in $R$ can be either undefined or defined in $\rho$.

**Definition 10 (Environment Compatibility).**

1. Let $R$ be a reaction environment of the synchronous semantics and $\rho$ an environment of values. We define the relation $\rho \sim R$ such that:

   $$
   \rho \sim R \stackrel{\text{def}}{=} (\text{Dom}(\rho) \subseteq \text{Dom}(R)) \land
   (\forall x \in \text{Dom}(\rho), \forall v \neq \text{abs}. (R(x) = v) \Rightarrow (\rho(x) = v))
   $$

2. We lift this relation to sequences of environments and write $h \sim H$ such that:

   $$
   \epsilon \sim \epsilon \quad \forall R, \rho.(\rho \sim R \land h \sim H) \iff ((\rho.h) \sim (R.H))
   $$

If $hm$ is a sequence of memories $\langle mv^i, jv^i \rangle$, we simply write $hm \sim H$ when $mv \sim H$ (instances $jv$ are discarded). We also write $h + hm \sim H$ when $h^i + mv^i \sim H^i$.

Next, we define a relation between a declaration $D$ and a pair $\langle M, L \rangle$ made of an initial memory $M$ and a sequence of instructions $L$. The relation holds if executing the declaration and the code in compatible environments yields compatible environments (trace equality).

**Definition 11 (Compatibility).** A collection of normalized equations $D$ is compatible with a pair $\langle M, L \rangle$ where $M$ is an initial memory and $L$ is a sequence of instructions $L$ when:

$$
D \triangleright \langle M, L \rangle \stackrel{\text{def}}{=} (\forall H, H', h, hm.
(h + hm \sim H) \land (H + H' \upharpoonright D \downarrow H')
\Rightarrow \exists h', hm'.
(hm'^n = M) \land
(\forall i \leq \text{length}(h). (hm^i + hm'^i, h^i \uparrow L \downarrow hm^i + hm'^{i+1}, h^i + h'^i) \land
(hm'^i + h'^i \sim H'^i))
$$

This is a very general notion of compatibility, we define a stronger notion of well-formed compatibility that applies to code obtained from the translation of a normalized and scheduled declaration:

- the code is executed in a context in which the memory is built by the instantiation of a machine, and is consistent with the reset function of that machine.
- the set of variables in the memory and in the scheduled set of equations is the same.

**Definition 12 (Well-Formed Compatibility).** A declaration $D$ is well-formed compatible with $\langle m, S, j, L \rangle$ when:

$$
D \triangleright \langle m, S, j, L \rangle \stackrel{\text{def}}{=} (\text{WFReset}(m,j)(S)) \land (D \triangleright \langle \text{Alloc}(m,j), L \rangle)
$$

**Definition 13 (Semantics Preservation for Expressions).** An annotated expression $e_{bt}^k$ is compatible with $c$ when:

$$
\begin{align*}
e_{bt}^k \triangleright m c & \stackrel{\text{def}}{=} \forall H, s, h, hm. (H \upharpoonright e_{bt}^k \downarrow s) \land (h + hm \sim H) \land (\text{Dom}(m) \subseteq \text{Dom}(hm))
\Rightarrow \forall i \leq \text{length}(h). (H(e^k)^i = tt) \Rightarrow (hm^i, h^i \uparrow c \downarrow s^i)
\end{align*}
$$

If $\vec{a} = (a_1, ..., a_n)$ and $\vec{c} = (c_1, ..., c_n)$, $\vec{a} \triangleright \vec{c}$ holds iff for all $1 \leq i \leq n$, $a_i \triangleright c_i$. 30
The intuition is that $e^{ck}_{bt}$ is compatible with $c$ if executed in compatible input environment (for the relation $\sim$), they produce the same sequences.

**Definition 14 (Semantics Preservation for Nodes).** The body of a node definition is equivalent to a class when they have the same runs, that is:

$$\langle p, q, d, D \rangle \sim \langle m, j, \text{reset} = S_1, \text{step} = \{ p, q, d \setminus m, S_2 \} \rangle \overset{\text{def}}{=} D \triangleright \langle m, S_1, j, \{ S_2 \} \rangle$$

Note that this definition is very restrictive in that patterns $(p, q, d)$ are the same on both sides. The fact that the semantics of OBC programs is preserved by renaming is not necessary and it is orthogonal to the correctness proof of the translation from MINILS to OBC.

**Definition 15 (Semantics Preservation for Environments).** Two global environments $G$ and $G_c$ are equivalent, written $G \sim G_c$ when the following holds:

$$G \sim G_c \overset{\text{def}}{=} (\text{Dom}(G) = \text{Dom}(G_c)) \land (\forall f. G(f) \sim G_c(f))$$

9.3.2. Semantics Preservation

In this section, we assume that we are in a context containing the environments $G$ (for the nodes) and $G_c$ (for the classes) and that we are under the following hypothesis.

**Hypothesis 1 ($H_{env}$).** $G \sim G_c$

We are now ready to prove the correctness of the translation. To this end, we show that every function used by the translation preserves the semantics. We first consider expressions: we show that an expression $e^{ck}_{bt}$ is compatible with its translated version $c$.

**Lemma 7 (Correctness of $TE$).** The following property holds:

$$\forall e^{ck}_{bt}, m. NA(e^{ck}_{bt}) \Rightarrow (e^{ck}_{bt} \triangleright TE_m(e^{ck}_{bt}))$$

**Proof:** By induction on the structure of normalized expressions. The proof is detailed in Appendix A.1.

Next, we consider controlled expressions. Controlled expressions are essentially nested `merge` constructs whose leaves are expressions.

**Lemma 8 (Correctness of $TCA$).**

$$\forall e^{ck}_{bt}, y, m. (SCH(y = e^{ck}_{bt}) \Rightarrow (y = e^{ck}_{bt}) \triangleright \langle m, \text{skip}, [], \text{Control}(ck, TCA_m(y, e^{ck}_{bt}) \rangle)$$

**Proof:** The full proof is available in Appendix A.2.

We now consider the compilation of an equation. As previously mentioned, only well-formed equations are considered. As this proof is longer than the others, we simplify it with the following lemma.

**Lemma 9 (Node instantiation).** Let $f$ be a global name so that $G(f) \sim G_c(f)$. Then:

$$\forall \vec{a}, e^{ck}_{bt}, c', \vec{c}, (\vec{a} \triangleright c', \vec{c} \triangleright c) \Rightarrow (\vec{y} = f(\vec{a}) \text{ every } e^{ck}_{bt} \triangleright \langle [\vec{y}], \text{o.reset}, [f/o] \rangle \text{ Control}(ck, \text{case}(c') \{(\text{true : o.reset})\}); \text{ Control}(ck, \vec{y} = \text{o.step}(c_1, ..., c_n))}$$

The lemma states that the node instantiation of $f$ is compatible with the instantiation of its corresponding class.

**Proof:** The detailed proof is given in Appendix A.3.

We now consider the compilation of an equation. As before, only well-formed equations are considered.
Lemma 10 (Correctness of $TEq(.)$).

$\forall D, eq, m, j, S, L. (D \triangleright \langle m, S, j, L \rangle) \land (SCH(eq and D)) \Rightarrow (eq and D \triangleright TEq_{m,S,j,L}(eq))$

PROOF: The full proof is available in Appendix A.4.

Lemma 11 (Correctness of $TEqList(.)$).

$\forall D. SCH(D) \Rightarrow D \triangleright TEqList(D)$

PROOF: The full proof is available in Appendix A.5.

The following property states that $JoinList(L)$ is equivalent to $L$.

Lemma 12 (Join). The following two properties hold:

$\forall \rho, M, L. (M, \rho \vdash L \Downarrow M', \rho + \rho') \Rightarrow (M, \rho \vdash JoinList(L) \Downarrow M, \rho + \rho')$

PROOF: The proof is available in Appendix A.6.

These lemmas serve to establish the main property of the translation method: when the translation succeeds, running the target program under the same input environment as that given to the source program produces the same output environment returned by the source program.

Theorem 2 (Correctness of $TP(.)$). The following property holds:

$SCH(D) \land$

$TP(node f(p) returns (q) var d in D) =$

$\text{class } f = \text{memory = } m; \text{ instances = } j;$

$\text{reset = } S_1; \text{ step(p) returns(q) var d\{m in S_2\}}$

$\Rightarrow \langle p, q, d, D \rangle \triangleright \langle m, j, \text{reset = } S_1; \text{ step = } \langle p, q, d\{m, [S_2]\}\rangle \rangle$

PROOF: The full proof is available in Appendix A.7.

This last result establishes that the translation from MINILS to object-based sequential code is correct.

10. Discussion and Related Work

10.1. Historical perspective

This work began in the year 2000 when the extension of SCADE and the design of a new compiler were considered at ESTEREL-TECHNOLOGIES, basing them on principles introduced in LUCID SYNCHRONE [18, 19]. The ReLuC\textsuperscript{6} prototype concretized these ideas. The kernel language provided the same basic programming constructs as the language kernel MINILS and the compilation was organized in a similar way (though the present formulation is simpler). The prototype was able to compile large existing users models (up to 100,000 lines of generated C code) and to produce efficient sequential code. The compiler included several dedicated type systems (clock inference [16], causality analysis and initialization analysis [14]) and supported hierarchical automata through a compilation into the basic language [13]. Several principles initiated in the ReLuC prototype are now integrated into the new language SCADE 6 and its compiler is commercially available since 2008.

The present formalization work continues this line of work by exploring the feasibility of a formally verified synchronous compiler. The present work is a first but essential step towards that development.

\textsuperscript{6}ReLuC stands for (Retargetable Lustre Compiler). It was developed as an implementation reference for the next generation of SCADE.
10.2. Code-generation for Synchronous Languages

The differences with the academic compiler of Lustre were described in the introduction. The distinction with Signal comes from the different expressiveness of our source language and its associated clock calculus. It is for example not possible to express disjunctive clocks of the form $ck_1 \lor ck_2$ (stating that a value is present if one of the two clocks is true) as in Signal. Clocks are only of the form base on $c_1$ on ... on $c_n$, and they correspond directly to nested control structures. Moreover, we use a simpler clock calculus based on ML-type constraints whereas the clock calculus of Signal calls for boolean resolution [20, 9]. We deliberately avoid complexity in the clock verification so as to focus on the code generation problem. The introduction of an $n$-ary merge and the general form of clocks presented here does not seem to have been considered in Signal. Even though this construction could be encoded in Signal, obtaining good code would call for the full expressiveness of its clock calculus and a more complex code generation step. An interesting question is to know whether the resulting code generated by the Signal compiler coincides with that obtained here with simpler but dedicated techniques.

This work is also related to the works on the DC format [21] and its extension DC+ [22] introduced for the compilation of synchronous languages. The DC format allows for similar control properties as the source language which we consider. However, as [21] points out, DC was not considered as a programming language, whereas the language we consider does have a static and dynamic semantics. This means that the result of all steps in the compilation chain can be statically typed or clock checked. This feature is important in compilers used for critical software and has already been used in the qualification process of industrial projects that use Scade as a development tool.

Finally, code generation is related to code distribution (see [23] for a survey and most recent references). However, it does not seem that the description of the modular compilation of a language such as the one treated here has been considered in this context.

10.3. Formal Certification of Synchronous Compilers

The formalization of compilers for general-purpose languages is now an active topic, after the foundational work by Leroy [24]. Nonetheless, only a few experiments on the certification of synchronous compilers have been done in the past: Gimenez and Ledinot [25] on a compiler for Scade V3, Terrasse [26] on a compiler for Esterel and Pnueli et al. [27] for the translation validation on the code generated by the Signal compiler. Ryabtsev et al. [28] have recently proposed a similar technique for Simulink.

In 1999, Gimenez and Ledinot [25] considered the formalization of a compiler for Scade V3 inside the proof-assistant Coq. A partial specification of the semantics for the source language and a target C language were developed (e.g., clocking rules, type-system and dynamic semantics). Unfortunately, the specification was only partial and abandoned. The present paper thus pursues this effort but it targets the existing Scade language with its rich control-structures.

Terrasse [26] considered the formalization in Coq of a subset of the Esterel constructive semantics. The language is pure-Esterel (Esterel with only pure signals) and restricted to its combinatorial subset. In comparison, we consider the full set of synchronous programs, that is, comprising combinatorial and sequential functions as well as control-optimization through the use of clocks.

The translation validation of the code generated by the Signal compiler has been proposed by Pnueli et al. [27]. The tool CVT checks, for every run of the compiler, that the generated code is equivalent to the source code (with a model checker). To succeed, this technique requires traceability information from the compiler (in particular how state variables are translated). Ryabtsev et al. [28] have followed the same approach to validate the RealTime Workshop compiler for a subset of Simulink. They rely on equivalence checking using the SMT-based model checker Yices. The model checker itself is not however formally verified. We have chosen a different approach, exposing and proving correct every elementary step of the compiler.
More recently, the Gene-Auto project\(^7\) addressed the development of a DO-178 certified Simulink to C code generator. In [29], the authors present a formalization in Coq of the scheduling function of a partially ordered set of nodes and prove it correct, i.e., the scheduling function returns a total order compatible with the partial order. A OCaml implementation is then extracted from the Coq specification. This work is indeed very close to what we are describing in the present paper, but it addresses a small part of the overall compilation process.

Finally, there have been numerous axiomatic semantics of synchronous data-flow languages using proof-assistants. Among others, Coupet-Grimal et al. [30] define a co-inductive semantics for a subset of Lustre; Novak et al. [31] define a co-inductive semantics for Signal; Boulmé et al. [32] define a shallow embedding of Lucid Synchrone in Coq, also using co-inductive definitions for operators. Unfortunately, these axiomatizations were of limited interest for our purpose. The main difficulty we faced was in finding the right compiler organisation that leads to both an efficient and formalizable compiler, the invariants used for proving the code generation correctness and the treatment of the reset construct. None of the previous references addressed these issues.

11. Conclusion and Future Work

This article presented a compiler for a synchronous data-flow language into sequential imperative code. It is based on a modular code generation technique where every node definition is translated into one transition function, thus following the approach taken in industrial compilers such as the one of SCADE. The clock-directed technique presented in this article has been implemented in the ReLuC compiler of SCADE/Lustre and evaluated on industrial examples of realistic size [14]. It is now used in the compiler of SCADE 6. Nonetheless, its precise description has never been published or described before. The paper presents an even simpler organisation where almost all steps (transformations and optimisations) are expressed as source-to-source transformations. Yet, the efficiency of the generated code is preserved, with respect to the original formulation used in the ReLuC compiler.

This paper is accompanied with a precise formalization of the source language, the intermediate sequential language and the essential steps of the compiler. The most important result of this formalization is the proof of correctness of the code generation process: when the compilation succeeds, the target code is proved to be equivalent to the source code. To our knowledge, this is the first synchronous realistic compiler with such a property.

This result is a first but essential step toward the development of a formally verified compiler for a synchronous language using a proof assistant. A first experiment, called VeLus (for Verified Lustre) is under way. Several parts of the presented material (specifications of semantics, types and clocks and their correctness with respect to the semantics, well formation rules, and the correctness of source-to-source transformations) has been done. The organisation of the compiler contains several steps (type and clock inference, causality analysis, scheduling and normalization) implemented in OCaml and only verified, thus following the approach of CVT. On the contrary, the translation function is simple and programmed directly in the programming language of Coq. The most interesting perspective is the connection of VeLus to the CompCert compiler of Leroy et al [33].

Several stages in the compilation of a synchronous language have been deliberately omitted in this paper so as to focus on the code generation problem. This concerns in particular the clock calculus whose expressiveness has been reduced with respect to that of Lustre and Lucid Synchrone [16]. Moreover, we only give a semantics to expressions annotated with their type and clock. This calls for a formalization of the translation from un-annotated to annotated expressions together with a proof of semantics preservation. Finally, an important direction is the use of MiniLS as a target language for a richer input language and a formalization of the translation proposed in [13].

---

\(^7\)www.geneauto.org
12. Acknowledgments

Dariusz Biernacki implemented the emission of C and Java code from the intermediate object-based language in the first OCAML implementation of the MINILS compiler. Xun Gong proved the correctness of the clock calculus in COQ.

References


Appendix A. Semantics Preservation

Appendix A.1. Correctness of TE(·) (lemma 7)

The following property holds:

\[ \forall c^k, c, m. NA(c^k) \Rightarrow (c^k \triangleright_m TE_m(c^k)) \]

**Proof:** Let \( H, h, hm \) such that \( h + hm \sim H \) and \( \text{Dom}(hm) = \text{Dom}(m) \). Recall that \( h + hm \sim H \)
means that both \( h \), \( hm \) and \( H \) have the same length. The proof is done by induction on the structure of normalized expressions.

**Case** \((a = v^k)\)

1. Let \( s \) be such that \( H \vdash v^k \downarrow s \).
2. By definition, \( s = \text{const}(H, v) \) that is, for all \( i \leq \text{length}(H) \), \( \text{if } H(c^k)^i = \text{tt} \text{ then } s^i = v \).
3. According to the operational semantics, for all \( i \leq \text{length}(h) \) \( \text{hm}^i, h^i \vdash v \downarrow s^i \). This is in particular true at any instant for which \( H(c^k)^i \) = \( \text{tt} \). Thus, the property holds.

**Case** \((a = x^k)\)

1. Let \( s \) be such that \( H \vdash x^k \downarrow s \).

   **Case** \((x \notin m)\)
   
   (a) We have to prove that \( \forall i \leq \text{length}(h), (H(c^k)^i = \text{tt}) \Rightarrow \text{hm}^i, h^i \vdash x \downarrow s^i \). According to theorem 1, at every instant \( i \), \( s^i \) is present iff \( H(c^k)^i = \text{tt} \).
   
   (b) The property holds since \( \forall i \leq \text{length}(h), h^i \vdash s^i \).

   **Case** \((x \in m)\)

   (a) We have to prove that \( \forall i \leq \text{length}(h), (H(c^k)^i = \text{tt}) \Rightarrow \text{hm}^i, h^i \vdash \text{state}(x) \downarrow s^i \).
   
   (b) According to theorem 1, at every instant \( i \), \( s^i \) is present iff \( H(c^k)^i = \text{tt} \).
   
   (c) \( x \in \text{Dom}(m) \) and \( \forall i \leq \text{length}(h), h^i + \text{hm}^i \sim H^i \). We have \( \text{hm}^i(x) = s^i \). Thus, the property holds.

**Case** \((a = \text{op}(a_1, \ldots, a_n)^k)\)

1. Let \( s \) such that \( H \vdash \text{op}(a_1, \ldots, a_n)^k \downarrow s \). According to the data-flow semantics we have a bunch of flows \( s_1, \ldots, s_n \) such that \( H \vdash a_k \downarrow s_k \) for \( k \in [1..n] \).
2. By induction, we have for any \( k \in [1..n] \) and \( i \) such that \( s_k^i \) is present:

   \[ \text{hm}^i, h^i \vdash TE_m(a_k) \downarrow s_k^i \]

3. According to the synchronous semantics, \( s = \text{lift}(op)(s_1, \ldots, s_n) \). That is, for every \( i \) such that \( H(c^k)^i \) = \( \text{tt} \), \( s^i = \text{op}(s_1^i, \ldots, s_n^i) \).
4. According to the operational semantics, \( \text{hm}^i, h^i \vdash \text{op}(TE_m(a_1), \ldots, TE_m(a_k)) \downarrow s^i \). Thus, the property holds.

**Case** \((a = (a' \text{ when } C(x))^k)\)

1. Because \( a \) is well clocked, \( c^k \) is of the form \( ck = ck' \text{ on } C(x) \) and \( a' = \epsilon^k \).
2. Suppose that \( H \vdash a \downarrow s \). According to the data-flow semantics, there exists \( s' \) such that \( H \vdash a' \downarrow s' \) and \( s = \text{when}_C(s', H(x)) \).
3. Take an \( i \) such that \( i \leq \text{length}(H) \). Suppose that \( H(c^k)^i \) = \( \text{tt} \). According to the semantics of the \text{on} operator, \( H'(c^k') = \text{tt} \). Applying the induction hypothesis, \( \text{hm}^i, h^i \vdash TE_m(a') \downarrow s'^i \). So the property holds. If \( H(c^k)^i = \text{ff} \), the property trivially holds too as the relation \( \sim \) only compares values at present instants.
Appendix A.2. Correctness of TCA (., .) (lemma 8)

The following property holds:

\[ \forall \epsilon^{ck}_{bt}, y, m. \text{SCH}(y = \epsilon^{ck}_{bt}) \Rightarrow (y = \epsilon^{ck}_{bt}) \triangleright (\langle [], \text{skip} [], \text{Control}(ck, \text{TCA}_m(y, \epsilon^{ck}_{bt})) \rangle) \]

PROOF: The proof is done by induction on the structure of normalized control expressions.

CASE \((a = \epsilon^{ck}_{bt})\)

1. Our goal becomes: \((y = \epsilon^{ck}_{bt}) \triangleright (\langle [], \text{skip} [], \text{Control}(ck, y := \text{TE}_m(\epsilon^{ck}_{bt})) \rangle)\).
2. Suppose that \(H + [s/y] \vdash y = \epsilon^{ck}_{bt} \triangleleft [s/y] \). According to the data-flow semantics, this is true if \(H + [s/y] \vdash \epsilon^{ck}_{bt} \triangleleft s \).
3. Because \(\text{SCH}(y = \epsilon^{ck}_{bt})\), \(y\) is not read in \(\epsilon^{ck}_{bt}\). We can simplify our hypothesis into \(H \vdash \epsilon^{ck}_{bt} \triangleleft s \).
4. Let \(h, hm\) such that \(h + hm \sim H\). By lemma 7, \(\epsilon^{ck}_{bt} \triangleright_m \text{TE}_m(\epsilon^{ck}_{bt})\).
5. Thus, for all \(i \leq \text{length}(h)\), if \(H^i(ck) = tt\) then \(hm^i, h^i \vdash \text{TE}_m(\epsilon^{ck}_{bt}) \triangleleft s^i\).
6. Let \(i \leq \text{length}(h)\). Either \(H^i(ck) = tt\) or \(H^i(ck) = ff\). In the first case:

\[ hm^i, h^i \vdash \text{Control}(ck, y := \text{TE}_m(\epsilon^{ck}_{bt})) \triangleleft hm^i, (h^i + [s^i/y]) \]

Taking \(hm^i\) as the sequence of empty environments, we have:

\[ hm^i + hm'^i, h^i \vdash \text{Control}(ck, y := \text{TE}_m(\epsilon^{ck}_{bt})) \triangleleft hm^i + hm'^{i+1}, (h^i + [s^i/y]) \]

and this is the expected result.

7. If \(H^i(ck) = ff\), then:

\[ hm^i, h^i \vdash \text{Control}(ck, y := \text{TE}_m(\epsilon^{ck}_{bt})) \triangleleft hm^i, h^i \]

Taking \(hm^i\) as a sequence of empty environments:

\[ hm^i + hm'^i, h^i \vdash \text{Control}(ck, y := \text{TE}_m(\epsilon^{ck}_{bt})) \triangleleft hm^i + hm'^{i+1}, h^i \]

Since \(s^i = \text{abs}\), we have \([s^i/y] \sim h^i\). Thus, the property holds.

CASE \((a = \text{merge} x (C_1 \rightarrow a_1) \ldots (C_n \rightarrow a_n))^{ck}_{bt}\)

1. By definition, \(\text{TCA}_m(y, a) = S\) where \(S = \text{case} (x) \{C_1 : S_1; \ldots; C_n : S_n\}\ and for all \(k \in [1..n]\), \(\text{TCA}_m(y, a_k) = S_k\).
2. Let \(H + [s/y] \vdash y = \text{merge} x (C_1 \rightarrow a_1) \ldots (C_n \rightarrow a_n))^{ck}_{bt} \triangleleft s\). According to the data-flow semantics, this is true if for all \(k \in [1..n]\), there exists \(s_k\) and \(sx\) such that \(H(x) = sx\) and \(H + [s/y] \vdash a_k \triangleleft s_k\).
3. Because \(a\) is well-clocked and well scheduled, \(a_k = e^{ck}_{bt}\) with \(c_k = c_k' \text{ on } C_k(x)\), the clock of \(x\) is \(c_k\) and \(\text{SCH}(y = a_k)\).
4. Applying the recurrence hypothesis, for all \(k \in [1..n]\), \((y = a_k) \triangleright (m, \text{skip} [], S_k)\ where S_k is of the form \(S_k = \text{Control}(c_k' \text{ on } C_k(x), S_k')\).
5. Let \(S' = \text{Control}(ck, \text{case} (x) \{C_1 : \text{Control}(ck_1, S_1); \ldots; C_n : \text{Control}(ck_n, S_n)\})\).
6. Let \(h, hm\) such that \(h + hm \sim H\). Let \(i \leq \text{length}(h + hm)\). Either \(H^i(ck) = tt\) or \(H^i(ck) = ff\). Suppose that \(H^i(ck) = tt\). According to the semantics of \text{merge}, it exists \(k \in [1..n]\) such that \(H(x)^i = C_k\), \(H^i(ck' \text{ on } C_k(x)) = tt\). Thus \(H^i(ck') = tt\) and so \(s^i = s_k\). Thus:

\[ hm^i + hm'^i, h^i \vdash S' \triangleleft hm^i + hm'^{i+1}, h^i + h'^i + [s^i/x] \]

and the property holds.

According to the operational semantics, \(hm^i, h^i \vdash \text{Control}(ck' \text{ on } C_k(x), S_k') \triangleleft hm^i, h^i + [s_k'/x]\).
Appendix A.3. Node instantiation (lemma 9)

Let $f$ be a global name so that $G(f) \sim G_e(f)$. Then:

$$ \forall \bar{a}, e^k_{bt}, c, c'. (\bar{a} \triangleright c) \land (c^k_{bt} \triangleright c') \Rightarrow (\bar{y} = f(\bar{a}) \text{ every } e^k_{bt}) \models (\langle \langle \rangle \rangle, \text{o.reset}, [f/o])$$

Control(ck, case (c')) {\{true : o.reset\}};

Control(ck, \bar{y} = o.step(c_1, ..., c_n))

PROOF:

1. Let $\bar{a} = (a_1, ..., a_n)$, $\bar{c} = (c_1, ..., c_n)$ and $\bar{y} = (y_1, ..., y_k)$, $e^k_{bt}$ and $c'$ such that $(\bar{a} \triangleright \bar{c}) \land (e^k_{bt} \triangleright c')$.

2. Let $S_0 = \text{Control}(ck, \text{case } (c') \{\{\text{true : o.reset}\}\})$.

Control(ck, \bar{y} = o.step(c_1, ..., c_n))

3. We must prove that:

$$\begin{align*}
(P_1) & \quad \text{WFReset}([f/o])\langle o.reset \rangle \\
(P_2) & \quad \forall H, H', h, hm.(h + hm \triangleright H) \land (H + H' \triangleright \bar{y} = f(\bar{a}) \text{ every } e^k_{bt} \downarrow H') \Rightarrow \\
& \quad \exists h', hm', \text{hmo.} \forall i \leq \text{length}(h). P
\end{align*}$$

with

$$P = (hm^i + hm'^i + [\text{hmo/o}]^i, h^i + s_0 \downarrow hm^i + hm'^i + [\text{hmo/o}]^{i+1}, h^i + h'^i) \land (h'^i + hm'^i + [\text{hmo/o}]^i \triangleright H')$$

Taking the empty environment for $hm'$ and simplifying the second part of $P[i]$, we get:

$$P = (hm^i + [\text{hmo/o}]^i, h^i + s_0 \downarrow hm^i + [\text{hmo/o}]^{i+1}, h^i + h'^i) \land (h'^i \triangleright H')$$

4. Property $P_1$:

(a) Let $G_e(f) = \langle m, S_1, j, S_2 \rangle$. Because $G(f) \sim G_e(f)$, WFReset(m)(S_1). Thus, executing o.reset reproduces the initial state, that is: $\forall M, \langle M/o \rangle, \rho \models \text{o.reset} \downarrow \langle \text{Alloc}(m, j)/o \rangle, \rho$.

(b) This is the expected property.

5. Property $P_2$ is proved by induction on the length of $H'$. According to the data-flow semantics, $H' = [\bar{s}/\bar{y}]$ with $\bar{s} = (s_1, ..., s_k)$.

(a) If $H' = \epsilon$, the property trivially holds as $i \leq \text{length}(h)$ is false.

(b) Otherwise, suppose that $H'$ is not empty. Let us develop the definition of the synchronous semantics. First of all:

$$H + H' \triangleright \bar{a} \downarrow s \bar{a} \quad H + H' \triangleright e^k_{bt} \downarrow r \quad H_0 + [s\bar{a}/p] \vdash_r D \downarrow H_0$$

with $H_0 = [sd/d] + [s/q]$.

(c) The sequence of reset $r$ is not empty. We split it according to the position of the first occurrence of a true value, that is, the definition of seq(). Let:

$$r = \text{seq}(r)(r_1)(r_2) \quad sd = \text{seq}(r)(sd_1)(sd_2)$$

and

$$s = \text{seq}(r)(s_1)(s_2) \quad sa = \text{seq}(r)(sa_1)(sa_2)$$

In the same way:

$$H = \text{seq}(r)(H_1)(H_2) \quad H' = \text{seq}(r)(H'_1)(H'_2)$$

and:

$$H_{11} = \text{seq}(r)(H_1)(\epsilon) \quad H'_{11} = \text{seq}(r)(H'_1)(\epsilon)$$
Appendix A.4. Correctness of TEq (. ) (lemma 10)

The following property holds:

\[ \forall D, eq, m, j, S, L. (D \triangleright (m, S, j, L)) \land (SCH(eq \land D)) \Rightarrow (eq \land D \triangleright TEq_{(m,S,j,L)}(eq)) \]

**Proof:** The proof is done by induction on the structure of normalized equations.

**Case (eq = (y = e_{bt}^{ck})**

(d) We now develop the definition of the data-flow semantics. Since:

\[ H + H' \vdash \equiv = f (\bar{a})\ every\ e_{bt}^{ck} \Downarrow H' \]

holds, we both have:

\[ (Q_1) \quad H_{11} + H'_{11} \vdash \equiv = f (\bar{a})\ every\ e_{bt}^{ck} \Downarrow H'_{11} \]

\[ (Q_2) \quad H_2 + H'_2 \vdash \equiv = f (\bar{a})\ every\ e_{bt}^{ck} \Downarrow H'_2 \]

\[ (Q_3) \quad H = H_{11} \bullet H_2 \quad \text{and} \quad H' = H'_{11} \bullet H'_2 \]

(e) Developing Q1, according to the data-flow semantics, we have:

\[
[sd_1/d] + [s_1/q] + [sa_1/p] \vdash D \Downarrow [sd_1/d] + [s_1/q]
\]

(f) Let ho1 such that ho1 \sim [sa_1/p]. By environment compatibility, it exists ho'1 and hmo1 such that:

\[ \forall i \leq \text{length}(ho_1), hmo_1, ho_1' \vdash S_2 \Downarrow \text{hmo}_1^{i+1}, ho_1' + ho_1'' \]

with ho_1'' + hmo_1 \sim [sd_1/d] + [s_1/q].

(g) According to Q3, H and H' can be split into two parts. h can be written h = h_{11} \bullet h_2 taking h_{11} such that h_{11} \sim H_{11} and h_2 \sim H_2. h can be written h' = h'_{11} \bullet h'_2 taking h'_{11} such that h'_{11} \sim H'_{11} and h'_2 \sim H'_2.

(h) We now define hmo_{11} so that hmo_{11} = \text{hold}([\text{Alloc}(m, j)/o])(r_{11})(hmo_1) where:

\[
\begin{align*}
\text{hold}(v)(s_1)(s_2)^1 &= v \\
\text{hold}(v)(s_1)(s_2)^i &= \text{hold}(v)(s_1)(s_2)^{i-1} \quad \text{if} \ s^i = \text{abs} \\
\text{hold}(v)(s_1)(s_2)^i &= s^i \quad \text{otherwise}
\end{align*}
\]

That is, the i-th value of the sequence s_2 is kept between instants where s_{11} is absent.

(i) Let h_{11} and h_{m1} such that h_{11} + h_{m1} \sim H_1. We have:

\[ \forall i \leq \text{length}(h_{11}), (h_{m1}^i + [\text{hmo}_{11}/o]^i, h_{11} \vdash S_0 \Downarrow h_{m1}^i + [\text{hmo}_{11}/o]^{i+1}, h_{11} + h_{m1}^i) \land (h_{m1}^i \sim H_{11}^i) \]

(j) We now apply the recurrence hypothesis for P2. Let h_2 and h_{m2} such that h_2 + h_{m2} \sim H_2. There exists h_2 and h_{m2} such that:

\[ \forall i \leq \text{length}(h_2), (h_{m2}^i + [\text{hmo}_2/o]^i, h_2 \vdash S_0 \Downarrow h_{m2}^i + [\text{hmo}_2/o]^{i+1}, h_2 + h_{m2}^i) \land (h_{m2}^i + h_{m2}^i \sim H_{22}^i) \]

(k) As hm = h_{m1} \bullet h_{m2}, we thus have:

\[ \forall i \leq \text{length}(h), (h^i + [\text{hmo}/o]^i, h^i \vdash S_0 \Downarrow h^i + [\text{hmo}/o]^{i+1}, h^i) \land (h^i + h^i \sim H^i) \]

and this concludes the proof. \[ \square \]
1. Let $\text{Cont} = \langle m, j, S, L \rangle$ and $\text{Cont}' = \langle m', j', S', L' \rangle$ such that $\text{TEq}_{\text{Cont}}(eq) = \text{Cont}'$. Let $D$ such that $D \triangleright \text{Cont}$ and $\text{SCH}(eq \text{ and } D)$.

2. By definition of the translation function, $\text{Cont}' = \langle m, j, S, L' \rangle$ with $L' = \text{Control}(ck, S_y); L$.

3. Let $H + H' + [s/y] \vdash eq \text{ and } D \triangleright H' + [s/y]$.

4. According to the data-flow semantics, this holds provided we have $H + H' + [s/y] \vdash eq \triangleright [s/y]$ and $H + H' + [s/y] \vdash D \triangleright H'$.

5. Since $D \triangleright \text{Cont}$, for all $h, hm$ such that $h + hm \sim H + [s/y]$, it exists $h'$ and $hm'$ such that $hm'^n = \text{Alloc}(m, j)$ and:

$$\forall i \leq \text{length}(h), (hm^i + hm'^i, h^i \triangleright L \downarrow hm^i + hm'^{i+1}, h^i + h'^i) \land (h^n + hm^i \sim H^n)$$

6. By lemma 8:

$$\forall i \leq \text{length}(h), (hm^i, h^i \triangleright \text{Control}(ck, S_y) \downarrow hm^i, h^i + h'^i) \land (h'_y \sim [s/y])$$

7. Let $i \leq \text{length}(h)$. By definition of the operational semantics:

$$hm^i + hm'^i, h^i \triangleright L \downarrow hm^i + hm'^{i+1}, h^i + h'^i + h^i$$

This is the first part for $eq \text{ and } D \triangleright \text{Cont}'$ to hold.

8. The second part is to check that $\text{WFReset}(m', j')(S')$. Since $m' = m$, $j' = j$ and $S' = S$, the result holds by definition since $\text{WFReset}(m, j)(S)$.

**Case** $eq = (y = (v \text{ fby } a)^{ck})$

1. Let $\text{Cont} = \langle m, j, S, L \rangle$ and $\text{Cont}' = \langle m', j', S', L' \rangle$.

2. According to the definition of the translation function, $m' = m + [y : bt = v]$, $j' = j$, $S' = \text{state}(y) := v; S$ and $L' = \text{Control}(ck, \text{state}(y) := c); L$ if $\text{TE}_{mv}(a) = c$.

3. Let $H + H' + [s/y] \vdash eq \text{ and } D \triangleright H' + [s/y]$.

4. According to the data-flow semantics, this holds provided we have:

$$H + H' + [s/y] \vdash eq \triangleright [s/y] \quad \text{and} \quad H + H' + [s/y] \vdash D \triangleright H'$$

5. The first case holds provided $H + H' + [s/y] \vdash a \triangleright s_a$ with $s = \text{fby}_v^{\#}(s_a)$.

6. Let $i \leq \text{length}(h)$.

7. Since $eq \text{ and } D$ is scheduled, $y$ does not appear in $D$ and thus in $L$. Moreover, $D \triangleright \text{Cont}$. It exists $h'$, $hm$ and $h'^i$ such that:

$$(hm^i + h'^i) + hm'^i, h^i \triangleright L \downarrow (hm^i + h'^i) + hm'^{i+1}, h^i + h'^i$$

with $hm'^i = \text{Alloc}(m, j)$ and $h^n + hm'^i \sim H'$ and $h'^i \sim [s'^i/y]$.

8. Let $hm'^i = \text{Alloc}(m + [y : bt = v], j)$. There are now two cases to consider, $H^i(ck) = ff$ and $H^i(ck) = tt$.

9. **Case** $H^i(ck) = ff$

   a. According to the operational semantics, we have:

   $$hm^i + [v/y], h^i \vdash \text{Control}(ck, \text{state}(y) := c) \downarrow hm^i + [v/y], h^i$$

   for any value $v$. This is in particular true when taking $v = s^n_a$.

   b. By composing the operational semantics of $\text{Control}(ck, \text{state}(x) := c)$ and that of $L$, we can state that:

   $$hm^i, h^i \vdash L \downarrow hm'^{i+1}, h^i + h'^i$$

   with $hm'^i = hm + [s'^i/y]$. $hm'^i$ is essentially equal to $hm'$, the value of $y$ being unchanged.

   c. This ends the first part of the proof.

10. **Case** $H^i(ck) = tt$
11. (a) According to the operational semantics, we have:

\[ hm^i + [s^i_a/y], h^i ▼ Control(ck; state(y) := c) ▼ hm^i + [s^{i+1}_a/y], h^i \]

(b) By composing the operational semantics of Control(ck; state(x) := c) and that of L, we can state that:

\[ hm^i_y, h^i ▼ L ▼ hm^i_y + [s^i_a/y], h^i \]

with \( hm^i_y = hm^i + [s^i_a/y] \).

(c) This ends the second part of the proof.

12. To get the final property, we must also prove that \( S' \) properly resets the state. This is an immediate consequence of the definition of \( S' = \text{state}(y) := v; S \) since \( y \) does not appear in \( S \) and \( S \) properly resets \( m \).

Case \((eq = (p = (f(a_1, ..., a_n) \text{ every } x)^c))\)

1. The proof is essentially that for the case \((eq = (y = c_{bk}))\), using lemma 9 instead of lemma 8.

\[ \square \]

Appendix A.5. Correctness of \( \text{TEqList(.) (lemma 11)} \)

\[ ∀D. \text{SCH(D)} \Rightarrow D ▼ \text{TEqList(D)} \]

\[ \text{PROOF :} \]

1. By induction on \( D \).
2. Since \( D \) is scheduled, \( D \) is either of the form \( eq \) or \( eq \text{ and } D' \).
3. In the first case, we apply lemma 10.
4. In the second one, we apply the recurrence hypothesis on \( D' \) and lemma 10.

\[ \square \]

Appendix A.6. Correctness of Join (lemma 12)

The following two properties hold:

\[ ∀ρ, M, L. (M, ρ ▼ L ▼ M', ρ + ρ') \Rightarrow (M, ρ ▼ \text{JoinList}(L) ▼ M', ρ + ρ') \]

\[ \text{PROOF :} \]

1. The first one is proved by induction on \( L \). The only interesting case is when \( L = [S_1; S_2]@L' \).
   If \( M, ρ ▼ S_1; S_2 ▼ M', ρ + ρ'_1 + ρ'_2 \) then \( M, ρ ▼ \text{Join}(S_1, S_2) ▼ M', ρ + ρ'_1 + ρ'_2 \). Thus, the property holds.

\[ \square \]

Appendix A.7. Correctness of \( \text{TP (.)(theorem 2)} \)

The following property holds:

\[ \text{SCH(D)} \land \text{TP(node f(p) returns(q) var d in D)} = \]

\[ \text{class f = \{memory = m; instances = j; reset = S_1; step(p) returns(q) var d\in m in S_2\}} \]

\[ \Rightarrow D ▼ \{m, j, \text{reset = S_1; step = \{p, q, d\in m, [S_2]\}}\} \]

\[ \text{PROOF :} \]

1. By application of Lemma 11, \( D ▼ \{m, j, S_1, L\} \) where \( \text{TEqList(D)} = \{m, j, S_1, L\} \)
2. By application of Lemma 12, \( D ▼ \{m, j, S_1, [\text{JoinList(L)}]\} \).
3. Thus, \( (p, q, d, D) ▼ \{m, j, \text{reset = S_1; step = \{p, q, d\in m, \text{JoinList(L)}\}} \) and this is the expected result.

\[ \square \]