Clock-directed Modular Code Generation for Synchronous Data-flow Languages

Dariusz Biernacki *
Institute of Computer Science
University of Wroclaw
Wroclaw, Poland
dabi@ii.uni.wroc.pl

Jean-Louis Colaço †
Prover Technology
Toulouse, France
Jean-louis.Colaco@prover.com

Grégoire Hamon
The MathWorks
Boston, USA
Gregoire.Hamon@mathworks.com

Marc Pouzet ‡
LRI, Univ. Paris-Sud 11
INRIA
Orsay, France
Marc.Pouzet@lri.fr

Abstract

The compilation of synchronous block diagrams into sequential imperative code has been addressed in the early eighties and can now be considered as folklore. However, separate, or *modular*, code generation, though largely used in existing compilers and particularly in industrial ones, has never been precisely described or entirely formalized. Such a formalization is now fundamental in the long-term goal to develop a mathematically certified compiler for a synchronous language as well as in simplifying existing implementations.

This article presents in full detail the modular compilation of synchronous block diagrams into sequential code. We consider a first-order functional language reminiscent of LUSTRE, which it extends with a general n-ary *merge* operator, a *reset* construct, and a richer notion of clocks. The clocks are used to express activation of computations in the program and are specifically taken into account during the compilation process to produce efficient imperative code. We introduce a generic machine-based intermediate language to represent transition functions, and we present a concise clock-directed translation from the source to this intermediate language. We address the target code generation phase by describing a translation from the intermediate language to JAVA and C.

Categories and Subject Descriptors C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.3.2 [*Language Classifications*]: Data-flow languages; D.3.4 [*Processors*]: Code generation, Compilers

General Terms Algorithms, Languages, Theory, Verification

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

1. Introduction

Block diagram formalisms as found in SCADE/LUSTRE [22] or SIMULINK [18] are widely used for embedded system design. Among these, synchronous block diagrams are based on a discrete model of time where signals are infinite streams and blocks define stream functions. The code generation from synchronous block diagrams into sequential imperative code is an old topic and has been addressed in the early years of LUSTRE [4]. The subject can now be considered as a part of the original folklore in synchronous programming [2].

Given a stream function $f : Stream(T) \rightarrow Stream(T')$ and a stream equation $y = f(x)$, the code generation consists in producing a pair (f_t, s_0) made of a transition function f_t of type $S \rightarrow T \rightarrow T' \times S$ and an initial state s_0 of type S, such that $\forall n \in I\!N. y_n, s_{n+1} = f_t\, s_n\, x_n$ if $x = (x_i)_{i \in I\!N}$ and $y = (y_i)_{i \in I\!N}$. The transition function takes a state and the current input, and it returns the current output with a new state. Its infinite repetition produces the sequence of outputs. In actual implementations, the transition function is written in imperative style with in-place modification of the state. Synchrony finds a very practical justification here: an infinite stream of type $Stream(T)$ is represented by a scalar value of type T and no intermediate memory nor complex buffering mechanism is needed. These principles are generalized to functions with multiple inputs and multiple outputs.

Code generation is obtained through a static scheduling of equations according to data dependencies. Separate, or modular, code generation aims at producing a transition function for each block definition and composing them together to produce the main transition function. As noticed by Gonthier [13], even in the absence of a causality loop, modular code generation is not always feasible. This is illustrated by the equation $(y, z) = copy(t, y)$ with $copy(x, y) = (x, y)$. This equation defines two perfectly valid streams y and z (since $y = t$ and $z = y = t$), but it cannot be scheduled independently of the way $copy$ is compiled. Said differently, it is not possible to produce only one transition function for

* This work started while the author was at INRIA Futurs in Orsay.

† This work started while the author was at ESTEREL-TECHNOLOGIES.

‡ This work was partially supported by ANR and INRIA.

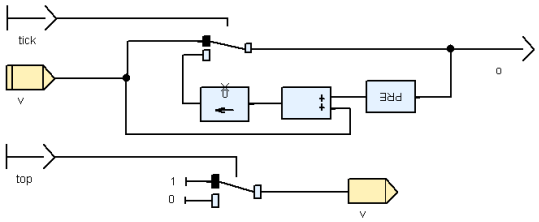
LCTES'08, June 12–13, 2008, Tucson, Arizona, USA.

copy which can be called in every context. This observation has led to two different approaches to the compilation problem. The first one aims to keep maximal expressiveness of the source language and consists in compiling the program after a full inlining of function calls. The resulting set of equations can then be translated into imperative code through simple scheduling techniques. Techniques for forward or backward enumeration of state variables can be used to generate an explicit finite state automaton leading to a very efficient code [4, 15]. Unfortunately, this efficiency gain comes at the price of modular compilation, and, moreover, the size of the generated code may explode in practice. For this reason the enumeration must be restricted to a selected set of state variables, as done in the academic LUSTRE compiler [16]. But finding the adequate variables which lead to efficient code is difficult both in time and size. Conversely, modular compilation is mandatory in industrial compilers like the one of SCADE. function is translated into one imperative function with no preliminary inlining unless requested by the programmer. Consequently, modular compilation imposes stronger causality constraints stating that every feedback loop must cross an explicit delay. Nonetheless, these constraints are well accepted by SCADE users. They are also justified by the need for *tracability* 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 consists in decomposing a function body into a minimal number of functions, each of them leading to one atomic transition function [21]. This solution is complementary to the modular compilation technique and can be included in a compilation chain as a pre-processing step.

Modular compilation of synchronous block diagrams, though largely used in the industrial compiler of LUSTRE has never been described precisely or formalized entirely. Such a formalization appears now as a fundamental need in the long-term goal to develop a mathematically certified compiler of a synchronous language inside a proof assistant such as COQ [10] as well as in simplifying existing implementations. This would give an opportunity to improve pure process-based certification as imposed by certification authorities by introducing stronger mathematical validation using proof techniques. To this aim, our contribution is to provide a minimal and complete description of the code generation step in order to serve as building block in a certified compilation chain.

This article presents in detail the modular compilation of synchronous block diagrams into sequential code. The source language we consider is a first-order declarative language reminiscent of LUSTRE, general enough to make a suitable intermediate language for the compilation of automata as introduced in [17, 6]. The language provides a n-ary *merge* operator as a way to combine complementary streams, a *reset* construct to restart a component in a modular way, and a generalized notion of *clocks* used to express various activation conditions. We introduce a generic object-based intermediate language to represent sequential transition functions, and we illustrate its versatility by giving a translation into JAVA and C. Synchronous programs are translated modularly into programs from the intermediate language. Clocks play a central role during the process of translation and are specifically treated to generate efficient control structures. This approach is in contrast to classical compilation methods based on enumeration techniques. The use of an intermediate language and the special treatment of clocks leads to a very concise description of the compilation process, yet produces efficient sequential code which competes with the one used in the new SCADE compiler.

This work is part of a long-term project to develop a certified LUSTRE compiler implemented in COQ. A reference compiler, based in particular on the material presented in this article, has been written in OCAML and in the programming language of COQ. Proofs of semantics equivalence in COQ are under way. For lack of space, we only describe the main steps in the compilation chain and do not give the formal semantics of the source and target languages.




```
-- count the number of top between two tick
node counting (tick:bool; top:bool)
returns (o: int)
var v: int;
let o = if tick then v else 0 -> pre o + v;
    v = if top then 1 else 0;
tel;
```

Figure 1. The counting node in SCADE and in LUSTRE

The article is organized as follows. In section 2, we present our input language, a synchronous data-flow kernel. In section 3, we address the issue of schedulability of a set of equations and lower our programs into a simpler normal form. Section 4 introduces the intermediate sequential language, and in section 5, we define the translation from the data-flow language to this intermediate language. In section 6, we describe code generation to JAVA and C . In section 7, we sketch the construction of the entire compiler. Finally, in section 8 and 9, we discuss related and future work.

2. A Clocked Data-flow Language

A program in LUSTRE consists of a number of node definitions, where each node computes its output from its input via a collection of parallel equations. In Figure 1 we present an example LUSTRE program along with its graphical representation programmed using the SCADE language. At each instant the output of the node `counting` (parameter `o`) is computed based on the inputs (`tick` and `top`) and its previous value (`pre o`), using a local variable (`v`), and it counts how many times the value of `top` has been true since the last time the value of `tick` was true.

In this section, we define a synchronous data-flow kernel considered as a basic calculus that is powerful enough to express any LUSTRE program. Additionally, this language contains some advanced features such as a means to *reset* a function application in a modular way, and *value constructors* belonging to enumerated types and filtering mechanisms. Since the code generation in our compiler is done after static analyzes, such as type and clock verification, we present the grammar of the language where every term is fully annotated with type and clock information.

2.1 Syntax and Intuitive Semantics

A program in the kernel language is made of a list of global type (td) and node (d) declarations. To simplify the presentation, only abstract and enumerated types are provided here. A global node declaration is of the form `node` $f(p) = p$ `with var` p `in` D, where p stands for a list of variable declarations, and D is a list of parallel equations. An equation ($pat = a$) defines the values of the variables mentioned in the pattern pat, where pat may be either a variable or a tuple of patterns $(pat, ..., pat)$. a stands for an annotated expression (e) with its clock (ct). Expressions

are values (v), variables (x), tuples ($a_1, ..., a_n$), initialized delays (v `fby` a), point-wise applications ($op\,(a_1, ..., a_n)$), node instantiations with a possible reset condition ($f\,(a_1, ..., a_n)$ `every` a), a sampling operation (a `when` $C(x)$), and a combination operation (`merge` x $(C_1 \rightarrow a_1)$ ... $(C_n \rightarrow a_n)$). The expression a `when` $C(x)$ is the sampled stream of a on the instants where x equals C. Symmetrically, `merge` is the combination operator: if a is a stream producing values belonging to a finite enumerated type $bt = C_1 + ... + C_n$ and $a_1, ..., a_n$ are complementary streams (i.e., at a given cycle, at most one stream is producing a value), then it combines them to form a faster stream. $f\,(a_1, ..., a_n)$ `every` a is the resetable function application: the internal state of the application of f is reset every time the boolean stream a is true. To simplify the presentation, we write $op\,(a_1, ..., a_n)$ for the point-wise application of an external function op (e.g., $+$, not) to its arguments and $f\,(a_1, ..., a_n)$ `every` `False` for the application of a stateful function. A value (v) can be a constructor (C) belonging to an enumerated type or any 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. Here is the grammar defining the syntax of the clock-annotated programs:

$$
\begin{array}{lcl}
td & ::= & \texttt{type}\, bt \mid \texttt{type}\, bt = C + ... + C \\
d & ::= & \texttt{node}\, f(p) = p\, \texttt{with var}\, p\, \texttt{in}\, D \\
p & ::= & x : bt; ...; x : bt \\
D & ::= & pat = a \mid D\, \texttt{and}\, D \\
pat & ::= & x \mid (pat, ..., pat) \\
a & ::= & e^{ct} \\
e & ::= & v \mid x \mid (a, ..., a) \mid v\, \texttt{fby}\, a \mid op\,(a, ..., a) \\
 & & \mid f\,(a, ..., a)\, \texttt{every}\, a \mid a\, \texttt{when}\, C(x) \\
 & & \mid \texttt{merge}\, x\, (C \rightarrow a)\, ...\, (C \rightarrow a) \\
v & ::= & C \mid i \\
ct & ::= & ck \mid ct \times ... \times ct \\
ck & ::= & \texttt{base} \mid ck\, \texttt{on}\, C(x)
\end{array}
$$

Clock annotations do not play any role in the data-flow semantics of the language, so we omit them in the examples below. It is assumed that the first parameter in the initialized delay v `fby` a is an immediate value. If op is a combinatorial function, $op\,(a_1, ..., a_n)$ applies it point-wise to its arguments (classical arithmetic operations are written in infix form). Here are some examples:

x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
v `fby` x	v	x_0	x_1	x_2	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	...

The kernel provides a general sampling mechanism based on enumerated types. This way, the classical sampling operation e `when` x of LUSTRE, where x is a boolean stream, is written e `when` `True`(x). In the same way, e `when not` x is written e `when` `False`(x). The conditional `if/then/else`, the delay `pre` and initialization operator `->` of LUSTRE can be encoded in the following way:

$$
\begin{array}{lcl}
\texttt{if}\, x\, \texttt{then}\, e_2\, \texttt{else}\, e_3 & = & \texttt{merge}\, x \\
 & & \quad (\texttt{True} \rightarrow e_2\, \texttt{when}\, \texttt{True}(x)) \\
 & & \quad (\texttt{False} \rightarrow e_3\, \texttt{when}\, \texttt{False}(x)) \\
e_1\, \texttt{->}\, e_2 & = & \texttt{if True fby False then}\, e_1 \\
 & & \texttt{else}\, e_2 \\
\texttt{pre}\,(e) & = & nil\, \texttt{fby}\, e
\end{array}
$$

The conditional `if/then/else` is built from the `merge` operator and the sampling operator `when`. The initialization operation e_1 `->` e_2 first returns the very first value of e_1 and then the current value of e_2. The uninitialized delay operation `pre` (e) is a shortcut for nil `fby` e where nil stands for any constant value which has the type of e. [1] The examples below illustrate these operators.

h	`True`	`False`	`True`	`False`	...
x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
x `->` y	x_0	y_1	y_2	y_3	...
`pre` (x)	nil	x_0	x_1	x_2	...
$z = x$ `when` `True`(h)	x_0		x_2		...
$t = y$ `when` `False`(h)		y_1		y_3	...
`merge` h (`True` $\rightarrow z$) (`False` $\rightarrow t$)	x_0	y_1	x_2	y_3	...

Forgetting clock annotations, the counting example of Figure 1 is written in the kernel language as follows:

```
node counting (tick:bool; top:bool) = (o:int) with
var v: int in
    o = if tick then v else 0 -> pre o + v
and v = if top then 1 else 0
```

2.2 Annotating Terms with Their Clocks

The code generation applies once type verification and clock calculus have been performed. At the end of these steps, every term is annotated with its type and clock. Typing is almost standard [20]. The purpose of the clock calculus is to reject programs which cannot be executed synchronously, and is also defined as a type inference system. Clocks do not have to be explicitly given in the source language, e.g., the programmer writes $(v$ `fby` $x) + y$ instead of $((v\, \texttt{fby}\, x^{ck})^{ck} + y^{ck})^{ck}$, if ck is the clock of x and y, and similarly he writes `merge` h (`True` $\rightarrow z$)(`False` $\rightarrow t$) instead of $(\texttt{merge}\, h\, (\texttt{True} \rightarrow z^{ck\ \texttt{on True}(h)})(\texttt{False} \rightarrow t^{ck\ \texttt{on False}(h)}))^{ck}$, if ck is the clock of h.

To make the article more self-contained, we present the associated clock conditions that must be verified by annotated terms (Figure 2). In a real implementation, the clock calculus applies to unannotated terms and produces annotated terms (the clock calculus shown here is based on [6]). We express that a program is well-clocked using several judgements. For instance, the judgement $H \vdash e : ct$ states that the expression e has a clock type ct under the clock environment H, $H \vdash D$ states that D is a set of well-clocked equations under the clock environment H. An environment H is of the form $[x_1 : ck_1, ..., x_n : ck_n]$, where $x_i \neq x_j$ for $i \neq j$.

In the rules for `when` and `merge`, it is assumed that the type correctness of the control variable and the type constructors has been verified by the type checker.

3. Toward Sequential Code

The language of Section 2 is declarative, with the evaluation of expressions controlled by the clock formalism. In order to generate sequential code from the source language, we first need to address the issues of finding a correct order for the equations, as well as dealing with fast, possibly stateful, computations inside slower ones.

3.1 Syntactic Dependencies and Scheduling

Following the definition introduced in [15], we say that an expression a *statically* depends on x if x appears free in a and not as an argument of a delay `fby`. $Left\,(a)$ returns the set of variables appearing this way in a (we overload the notation for $Left\,(e)$ and $Left\,(D)$). $Def\,(D)$ defines the set of variables defined in D.

[1] It is the purpose of the initialization analysis to check that the computation result does not depend on the actual *nil* value.

$$\frac{H \vdash e : ct}{H \vdash e^{ct} : ct} \qquad \frac{H \vdash a_1 : ck \; ... \; H \vdash a_n : ck}{H \vdash op\,(a_1, ..., a_n) : ck}$$

$$H \vdash v : \texttt{base}$$

$$H, x : ck \vdash x : ck$$

$$\frac{H \vdash a_1 : ck \; ... \; H \vdash a_n : ck \quad H \vdash a : ck}{H \vdash f\,(a_1, ..., a_n)\ \texttt{every}\ a : ck \times ... \times ck}$$

$$\frac{H \vdash a_1 : ct_1 \; ... \; H \vdash a_n : ct_n}{H \vdash (a_1, ..., a_n) : ct_1 \times ... \times ct_n}$$

$$\frac{H \vdash a : ck \quad H \vdash x : ck}{H \vdash a\ \texttt{when}\ C(x) : ck\ \texttt{on}\ C(x)} \qquad \frac{H \vdash a : ck}{H \vdash v\ \texttt{fby}\ a : ck}$$

$$\frac{H \vdash x : ck \quad H \vdash a_1 : ck\ \texttt{on}\ C_1(x) \; ... \; H \vdash a_n : ck\ \texttt{on}\ C_n(x)}{H \vdash \texttt{merge}\ x\ (C_1 \to a_1)\ ...\ (C_n \to a_n) : ck}$$

$$\frac{H \vdash pat : ct \quad H \vdash a : ct}{H \vdash pat = a} \qquad \frac{H \vdash D_1 \quad H \vdash D_2}{H \vdash D_1\ \texttt{and}\ D_2}$$

$$\frac{H \vdash pat_1 : ct_1 \; ... \; H \vdash pat_n : ct_n}{H \vdash (pat_1, ..., pat_n) : ct_1 \times ... \times ct_n}$$

$$\frac{\vdash_{\texttt{base}} p : H_p \quad \vdash_{\texttt{base}} q : H_q \quad \vdash r : H_r \quad H_p, H_q, H_r \vdash D}{\vdash \texttt{node}\ f(p) = q\ \texttt{with}\ \texttt{var}\ r\ \texttt{in}\ D}$$

$$\vdash x_1 : t_1, ..., x_n : t_n : [x_1 : ck_1, ..., x_n : ck_n]$$

$$\vdash_{\texttt{base}} x_1 : t_1, ..., x_n : t_n : [x_1 : \texttt{base}, ..., x_n : \texttt{base}]$$

Figure 2. Clock Constraints

$Vars(D)$ is the set of variables appearing in D. If $pat = a$ is an equation in D, every variable from pat immediately depends on variables from $Left\,(a)$. The transitive closure of this relation defines the notion of static dependency. A program is causal when for each node the corresponding graph of dependencies is acyclic. The corresponding definitions are given in figure 3.

In a set of equations D, an equation $pat = a$ is ready ($(pat = a) \in R\,(D)$) when it does not depend on any other equations. We make a particular treatment of equations of the form $x = (v\ \texttt{fby}\ a)^{ck}$. In this case, x corresponds to a memory so it will have to be scheduled after any other computation reading variable x.

$$(x = (v\ \texttt{fby}\ a)^{ck}) \in R\,(D) \text{ if } (Vars(a) \cap Def\,(D)) = \emptyset \wedge x \notin Vars(D)$$
$$(pat = a) \in R\,(D) \text{ if } Vars(pat) \cap (Left\,(D) \cup Left\,(a)) = \emptyset$$

We write $D|_{pat=e}$ for the exclusion of the equation $pat = e$ from D. A sequence of equations $l = pat_1 = e_1, ..., pat_n = e_n$ is a feasible schedule of D if $l \in Sch\,(D)$, where:

$$\begin{array}{lll} pat = a \in Sch\,(pat = a) & if & Left\,(a) \cap Vars(p) = \emptyset \\ pat = a, l \in Sch\,(D) & if & (pat = a) \in R\,(D|_{pat=a}) \\ & & \wedge\ l \in Sch\,(D|_{pat=a}) \end{array}$$

For the remainder, we assume that programs have passed a causality check that insure the existence of a schedule.

The data-flow nature of this language makes the implementation of classical graph-based optimizations (e.g., copy elimination, common-subexpression elimination) particularly easy. We do not detail them here.

$$\begin{array}{lcl}
Left\,(e^{ck}) & = & Left\,(e) \cup Vars(ck) \\
Left\,(v\ \texttt{fby}\ a) & = & \emptyset \\
Left\,(op\,(a_1, ..., a_n)) & = & \cup_{1 \le i \le n} Left\,(a_i) \\
Left\,(f\,(a_1, ..., a_n)\ \texttt{every}\ a) & = & \cup_{1 \le i \le n} Left\,(a_i) \cup Left\,(a) \\
Left\,(x) & = & \{x\} \\
Left\,(v) & = & \emptyset \\
Left\,(\texttt{merge}\ x\ \ (C_1 \to a_1) & = & \cup_{1 \le i \le n} Left\,(a_i) \cup \{x\} \\
\quad ... & & \\
\quad (C_n \to a_n)) & & \\
Left\,(a\ \texttt{when}\ C(x)) & = & \{x\} \cup Left\,(a) \\
 & & \\
Left\,(pat = a) & = & Left\,(a) \\
Left\,(D_1\ \texttt{and}\ D_2) & = & Left\,(D_1) \cup Left\,(D_2) \\
 & & \\
Def\,(x = (v\ \texttt{fby}\ a)^{ck}) & = & \emptyset \\
Def\,(pat = a) & = & Vars(pat) \\
Def\,(D_1\ \texttt{and}\ D_2) & = & Def\,(D_1) \cup Def\,(D_2) \\
 & & \\
Vars(x) & = & \{x\} \\
Vars((pat_1, ..., pat_n)) & = & \cup_{1 \le i \le n} Vars(pat_i)
\end{array}$$

Figure 3. Syntactic Dependencies

3.2 Putting Equations in Normal Form

We introduce a source-to-source transformation which consists in extracting stateful computations that appear inside expressions. This is a necessary step toward the translation into sequential code. For example, the following equation (omitting nested clock annotations for clarity):

$$\begin{array}{rl} & z = ((((4\ \texttt{fby}\ o) * 3)\ \texttt{when}\ \texttt{True}(c)) + k)^{ck\ \texttt{on}\ \texttt{True}(c)} \\ \texttt{and}\ & o = (\texttt{merge}\ c\ \ (\texttt{True} \to (5\ \texttt{fby}\ (z + 1)) + 2) \\ & \qquad\qquad (\texttt{False} \to ((6\ \texttt{fby}\ x))\ \texttt{when}\ \texttt{False}(c)))^{ck} \end{array}$$

is rewritten into:

$$\begin{array}{rl} & t_1 = (4\ \texttt{fby}\ o)^{ck} \\ \texttt{and}\ & z = (((t_1 * 3)\ \texttt{when}\ \texttt{True}(c)) + k)^{ck\ \texttt{on}\ \texttt{True}(c)} \\ \texttt{and}\ & t_2 = (5\ \texttt{fby}\ (z + 1))^{ck\ \texttt{on}\ \texttt{True}(c)} \\ \texttt{and}\ & t_3 = (6\ \texttt{fby}\ x)^{ck} \\ \texttt{and}\ & o = (\texttt{merge}\ c\ \ (\texttt{True} \to t_2 + 2) \\ & \qquad\qquad (\texttt{False} \to t_3\ \texttt{when}\ \texttt{False}(c)))^{ck} \end{array}$$

In the same way, node instances ($f\,(a_1, ..., a_n)\ \texttt{every}\ e$) are extracted from nested expressions. The extraction is made through a linear traversal, introducing equations for each stateful computation.

After the extraction, terms and equations can be characterized by the following grammar:

$$\begin{array}{lcl}
a & ::= & e^{ck} \\
e & ::= & a\ \texttt{when}\ C(x) \mid op\,(a, ..., a) \mid x \mid v \\
ce & ::= & \texttt{merge}\ x\ (C \to ca)\ ...\ (C \to ca) \mid e \\
ca & ::= & ce^{ck} \\
eq & ::= & x = ca \mid x = (v\ \texttt{fby}\ a)^{ck} \\
 & & \mid (x, ..., x) = (f\,(a, ..., a)\ \texttt{every}\ x)^{ck} \\
D & ::= & D\ \texttt{and}\ D \mid eq
\end{array}$$

The normalization functions are given in Figure 4. $NormE_D\,(a)$ returns a normalized expression for a; $NormCA_D\,(a)$ returns a normalized expression of the form ca for a and a new set of declarations; $NormD_D\,(D_1)$ normalizes the definitions D_1 and $NormV_D\,(a)$ returns a fresh variable storing the result of the normalized expression of a and a new set of declarations. To avoid

duplications, definitions of normalization functions are given in order.

Note that it would also be possible to introduce a new intermediate language instead of the source-to-source transformation. This is essentially a matter of taste, the main advantage of the present formulation being to save the redefinition of auxiliary notions.

4. A Simple Object-based Language

A classical way to encapsulate a state and a collection of functions that manipulate this state is given by the object-orientation paradigm. We are not interested in inheritance or object polymorphism aspects, but only in the capability to encapsulate a piece of memory managed exclusively by the methods of the class. We propose here to define a very simple object-based language that will be used as an intermediate language for the translation. Adopting this point of view has two main advantages compared to a direct translation into one 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 any target language like C or JAVA.

A stateful stream function or *node* can be considered as a simple class definition with instance variables and two methods `step` and `reset`. Variables are used to represent the internal state of the node (i.e., one for each delay). 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 block diagrams.

The syntax of the language is given below. A program is made of a sequence of global definitions (d) of classes. An instruction S may be an assignment of a local variable ($x := c$) or of a state variable ($\texttt{state}\,(x) := c$), a sequence ($S\,;\,S$), a reinitialization method invocation of an object o ($o.\texttt{reset}$), an invocation of the step method of object o ($o.\texttt{step}\,(e_1, \ldots, e_n)$), a void statement ($\texttt{skip}$), or a control structure ($\texttt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\}$). If x is of type $bt = C_1 + ... + C + ... + C_n$, we shall indifferently write $\texttt{case}\ (x)\ \{C_1 : \texttt{skip}; ...; C : S; ...; C_n : \texttt{skip}\}$ or $\texttt{case}\ (x)\ \{C : S\}$. An expression ($e$) can be either an access to a local variable (x) or to a state variable ($\texttt{state}\,(x)$), an immediate integer constant (i) or a value constructor (C), a tuple ($e_1, ..., e_n$), or a function call ($f\,(e_1, \ldots, e_n)$). A machine (f) defines a set of memories (m), a set of instances for objects used inside the body of the methods `step` or `reset` (j), and these two methods.

$$
\begin{array}{rcl}
d & ::= & \texttt{machine}\ f = \\
 & & \quad \texttt{memory}\ m \\
 & & \quad \texttt{instances}\ j \\
 & & \quad \texttt{reset}() = S \\
 & & \quad \texttt{step}\,(p)\,\texttt{returns}\,(p) = \texttt{var}\ p\ \texttt{in}\ S \\
S & ::= & x := c \mid \texttt{state}\,(x) := c \mid S\,;\,S \mid \texttt{skip} \\
 & & \quad \mid o.\texttt{reset} \mid (x, ..., x) = o.\texttt{step}\,(c, ..., c) \\
 & & \quad \mid \texttt{case}\ (x)\ \{C : S; ...; C : S\} \\
c & ::= & x \mid v \mid \texttt{state}\,(x) \mid op(c, ..., c) \\
v & ::= & C \mid i \\
j & ::= & o : f, ..., o : f \\
p, m & ::= & x : t, ..., x : t
\end{array}
$$

We only define the minimal intermediate language which is sufficient for the translation. One may consider a more general form with several methods, in particular to give access to the components of a structured output and avoid copying the output when calling a node. Nonetheless, this optimization is orthogonal to the translation and can be done afterwards.

5. The Translation

The translation closely follows the principle of the co-iterative semantics described in [5], restricted to the first-order language. The main differences are that absent values are not explicitly represented at run-time and states are modified in-place instead of being returned by transition functions.

We introduce the following notation. If $p = [x_1 : t_1; ...; x_n : t_n]$ and $p_2 = [x'_1 : t'_1; ...; x'_k : t'_k]$ then $p_1 + p_2 = [x_1 : t_1; ...; x_n : t_n; x'_1 : t'_1; ...; x'_k : t'_k]$ 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. 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. If $s_1 = S_1, ..., S_n$ and $s_2 = S'_1, ..., S'_k$ are two lists of instructions, we write $s_1 @ s_2$ for their concatenation $S_1, ..., S_n, S'_1, ..., S'_k$.

Clocks in the source language are transformed into control structures in the target language. Intuitively, a computation S on clock $\texttt{base}\ \texttt{on}\ C_1(x_1)\ \texttt{on}\ C'_1(x'_1)$ is transformed into the code: $\texttt{case}\ (x_1)\ \{C_1 : \texttt{case}\ (x'_1)\ \{C'_1 : S\}\}$. We define the function $Control(.,.)$ such that $Control(ck, S)$ returns a control structure so that S is executed only when ck is true:

$$
\begin{array}{lcl}
Control(\texttt{base}, S) & = & S \\
Control(ck\ \texttt{on}\ C(x), S) & = & Control(ck, \texttt{case}\ (x)\ \{C : S\})
\end{array}
$$

We also define the function $Join(.,.)$ which merges two control structures gathered by the same guards:

$$
\begin{array}{l}
Join(\ \texttt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\}, \\
\qquad \texttt{case}\ (x)\ \{C_1 : S'_1; ...; C_n : S'_n\}) \\
\quad = \texttt{case}\ (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\} \\
Join(S_1, S_2) = S_1; S_2
\end{array}
$$

$$
\begin{array}{l}
JoinList(S) = S \\
JoinList(S_1, ..., S_n) = Join(S_1, JoinList(S_2, ..., S_n))
\end{array}
$$

The translation is defined by a set of mutually recursive functions. $TE_{(m,si,j,d,s)}\,(e)$ defines the translation of an unannotated expression e in a context (m, si, j, d, s) and returns an expression from the target language c. We overload the notation for annotated expressions a. m stands for a memory environment, si stands for a list of instructions that initialize the memory, j is an environment for node instances, d is an environment for local variables and s is a list of instructions. $TA_{(m,si,j,d,s)}\,(x, ca)$ defines the translation of an expression which is stored into x and it returns a new context. $TEq_{(m,si,j,d,s)}\,(eq)$ defines the translation of an equation. We use two auxiliary functions: the operation $TEList_{(m,si,j,d,s)}\,(a_1, ..., a_n)$ translates a list of expressions and returns a list of expressions from the target language, whereas $TEqList_{(m,si,j,d,s)}\,(l)$ translates a list of equations.

The definitions of the translation functions are given in Figure 5. The first six rules apply to stateless expressions. The translation of a `merge` operator, whose result is stored into a pattern pat, is obtained by translating each branch and storing the corresponding result in pat. Note that since the result of each branch is annotated with its proper clock, the `merge` construction does not generate any code by itself. For a node instance $(f\,(a_1, ..., a_k)\ \texttt{every}\ x)^{ck}$, we introduce a fresh name o which is an object of the machine f. The initialization code consists in calling the `reset` method. The step function is essentially the result of calling the `reset` method when x is true and calling the step function associated to o. These two actions must be performed only when ck is true. A memory equation $x = (v\ \texttt{fby}\ a)^{ck}$ is translated into an assignment of the state variable x, executed when ck is true. Finally, the code generation of a node consists in first scheduling the set of equations and then translating them iteratively.

$$
\begin{array}{lcl}
NormE_D\,((v\ \texttt{fby}\ a)^{ck}) & = & let\ a, D = NormE_D\,(a)\ in\ x, x = (v\ \texttt{fby}\ a)^{ck}\ \texttt{and}\ D, \\
 & & \text{where } x \notin Vars(D) \\
NormE_D\,((a\ \texttt{when}\ C(x))^{ck}) & = & let\ a, D = NormE_D\,(a)\ in\ (a\ \texttt{when}\ C(x))^{ck}, D \\
NormE_D\,(x^{ck}) & = & x^{ck}, D \\
NormE_D\,(v^{ck}) & = & v^{ck}, D \\
NormE_D\,(op(a_1, ..., a_n)^{ck}) & = & let\ (a_1, ..., a_n), D = NormElist_D\,(a_1, ..., a_n)\ in\ op(a_1, ..., a_n)^{ck}, D \\
NormE_D\,((\texttt{merge}\ x\ (C_1 \to a_1)\ ...\ (C_n \to a_n))^{ck}) & = & let\ a_1, ..., a_n, D = NormCAlist_D\,(a_1, ..., a_n)\ in \\
 & & y, y = (\texttt{merge}\ x\ (C_1 \to a_1)\ ...\ (C_n \to a_n))^{ck}\ \texttt{and}\ D, \\
 & & \text{where } y \notin Vars(D) \\
NormE_D\,(a) & = & NormCA_D\,(a), \text{for the remaining forms of a} \\
 & & \\
NormV_D\,(a) & = & let\ a, D = NormE_D\,(a)\ in\ x, x = a\ \texttt{and}\ D, \text{where } x \notin Vars(D) \\
 & & \\
NormCA_D\,((f\,(a_1, ..., a_m)\ \texttt{every}\ a)^{ck}) & = & let\ (a_1, ..., a_m), D = NormElist_D\,(a_1, ..., a_m)\ in \\
 & & let\ x, D = NormV_D\,(a)\ in\ y, y = (f\,(a_1, ..., a_m)\ \texttt{every}\ x)^{ck}\ \texttt{and}\ D, \\
 & & \text{where } y \notin Vars(D) \\
 & & \\
NormCA_D\,((\texttt{merge}\ x\ (C_1 \to a_1)\ ...\ (C_n \to a_n))^{ck}) & = & let\ a_1, ..., a_n, D = NormCAlist_D\,(a_1, ..., a_n)\ in \\
 & & (\texttt{merge}\ x\ (C_1 \to a_1)\ ...\ (C_n \to a_n)), D \\
NormCA_D\,(a) & = & NormE_D\,(a), \text{for the remaining forms of a} \\
 & & \\
NormD_D\,(D_1\ \texttt{and}\ D_2) & = & NormD_{NormD_D\,(D_1)}\,(D_2) \\
NormD_D\,(px = (v\ \texttt{fby}\ a)^{ck}) & = & let\ x, D = NormV_D\,(a)\ in\ (px = (v\ \texttt{fby}\ x)^{ck})\ \texttt{and}\ D \\
NormD_D\,((x_1, ..., x_n) = (f\,(a_1, ..., a_m)\ \texttt{every}\ a)^{ck}) & = & let\ (a_1, ..., a_m), D = NormElist_D\,(a_1, ..., a_m)\ in \\
 & & let\ x, D = NormV_D\,(a)\ in \\
 & & (x_1, ..., x_n) = (f\,(a_1, ..., a_m)\ \texttt{every}\ x)^{ck}\ \texttt{and}\ D \\
NormD_D\,((pat_1, ..., pat_n) = (a_1, ..., a_n)) & = & let\ D = NormD_D\,(pat_1 = a_1)\ in \\
 & & ...let\ D = NormD_D\,(pat_n = a_n)\ in\ D \\
NormD_D\,(x = a) & = & let\ a, D = NormCA_D\,(a)\ in\ (x = a)\ \texttt{and}\ D, \\
 & & \text{for the remaining forms of a} \\
 & & \\
NormElist_D\,(a_1, ..., a_n) & = & let\ a_1, D = NormE_D\,(a_1)\ in \\
 & & ...let\ a_n, D = NormE_D\,(a_n)\ in\ (a_1, ..., a_n), D \\
 & & \\
NormCAlist_D\,(a_1, ..., a_n) & = & let\ a_1, D = NormCA_D\,(a_1)\ in \\
 & & ...let\ a_n, D = NormCA_D\,(a_n)\ in\ (a_1, ..., a_n), D
\end{array}
$$

Figure 4. The Normalization Function

Considering the set of normalized equations given in Section 3.2, it is first scheduled, obtaining, for example:

$$
\begin{array}{l}
\quad z = (((t_1 * 3)\ \texttt{when}\ \texttt{True}(c)) + k)^{ck\ \text{on}\ \texttt{True}(c)} \\
\texttt{and}\ o = (\texttt{merge}\ c\ \ (\texttt{True} \to t_2 + 2) \\
\qquad\qquad\qquad (\texttt{False} \to t_3\ \texttt{when}\ \texttt{False}(c)))^{ck} \\
\texttt{and}\ t_2 = (5\ \texttt{fby}\ (z + 1))^{ck\ \text{on}\ \texttt{True}(c)} \\
\texttt{and}\ t_1 = (4\ \texttt{fby}\ o)^{ck} \\
\texttt{and}\ t_3 = (6\ \texttt{fby}\ x)^{ck}
\end{array}
$$

Then, it is translated into:

$$
\begin{array}{l}
\texttt{case}\ (c)\{ \\
\ \texttt{True}:\ \ z := \texttt{state}\,(t_1) * 3 + k; \\
\qquad\quad o := \texttt{state}\,(t_2) + 2; \\
\qquad\quad \texttt{state}\,(t_2) := z + 1; \\
\ \texttt{False}: o := \texttt{state}\,(t_3); \\
\}; \\
\texttt{state}\,(t_1) := o; \\
\texttt{state}\,(t_3) := x
\end{array}
$$

This example illustrates the effect of scheduling decisions of equations on the resulting code. Control optimization, i.e., fusion of control structures, depends on the ability to put together equations on the same clocks, provided data dependencies are respected. Of course, one could do control optimization on the target code, but this would require rebuilding much of the data-flow information, which is already available in the source program.

We now illustrate modular compilation on the so-called *activation condition* of SCADE (or *enabled subsystem* in SIMULINK), instantiated on the counter. [2] The activation condition, implemented by the node `condact` below, takes an input `i` and a boolean sequence `c` and runs the node `count` on the sub-sequence of `i` on the instants where `c` is true. Otherwise, it keeps the previous value of `i`:

```
node count(i:int) = (o:int) with
  o = (0 fby o) + i
node condact(c:bool;i:int) = (o:int) with
  o = merge c (True -> count(i when True(c))
               (False -> (0 fby o) when False(c))
```

Then, the node `condact` is translated into the following code:

```
machine condact =
```

[2] The activation condition is an example of a higher-order primitive which takes a node as a parameter, so it cannot be expressed as is in our source language (unless treated specifically or first inlined).

$$
\begin{array}{lcl}
TE_{(m,si,j,d,s)}\,(e^{ct}) & = & TE_{(m,si,j,d,s)}\,(e) \\
TE_{(m,si,j,d,s)}\,(v) & = & v \\
TE_{(m,si,j,d+[x:t],s)}\,(x) & = & x \\
TE_{(m+[x:t],si,j,d,s)}\,(x) & = & \texttt{state}\,(x) \\
TE_{(m,si,j,d,s)}\,(op(a_1,...,a_n)) & = & \mathit{let}\ c_1,...,c_n = TEList_{(m,si,j,d,s)}\,(a_1,...,a_n)\ \mathit{in} \\
 & & op(c_1,...,c_n) \\
TE_{(m,si,j,d,s)}\,(a\ \texttt{when}\ C(x)) & = & TE_{(m,si,j,d,s)}\,(a) \\
TEList_{(m,si,j,d,s)}\,(a_1,...,a_n) & = & (TE_{(m,si,j,d,s)}\,(a_1),...,TE_{(m,si,j,d,s)}\,(a_n)) \\
TA_{(m,si,j,d,s)}\,(y,(\texttt{merge}\ x\ \begin{array}{l}(C_1 \to ca_1)\\ ...\\ (C_n \to ca_n)\end{array}\ )^{ck}) & = & \texttt{case}\,(x)\ \{\ \begin{array}{l}C_1 : TA_{(m,si,j,d,s)}\,(y,ca_1)\\ ...\\ C_n : TA_{(m,si,j,d,s)}\,(y,ca_n)\end{array}\ \} \\
TA_{(m,si,j,d,s)}\,(x,e^{ck}) & = & x := TE_{(m,si,j,d,s)}\,(e), \text{for the remaining forms of } e \\
TEq_{(m,si,j,d+[x:t],s)}\,(x = (v\ \texttt{fby}\ a)^{ck}) & = & \mathit{let}\ c = TE_{(m,si,j,d,s)}\,(a)\ \mathit{in} \\
 & & (m + [x:t], [\texttt{state}\,(x) := v]@si, j, d, \\
 & & [Control(ck, \texttt{state}\,(x) := c)]@s) \\
TEq_{(m,si,j,d,s)}\,(p = (f\,(a_1,...,a_k)\ \texttt{every}\ x)^{ck}) & = & \mathit{let}\ (c_1,...,c_k) = TEList_{(m,si,j,d,s)}\,(a_1,...,a_k)\ \mathit{in} \\
 & & (m, [o.\texttt{reset}]@si, [(o,f)] + j, d, \\
 & & Control(ck, \texttt{case}\ (x)\ \{(\texttt{True} : o.\texttt{reset})\})@ \\
 & & Control(ck, p = o.\texttt{step}\,(c_1,\ \ldots,\ c_k))@s) \\
 & & \text{where } o \notin Dom(j) \\
TEq_{(m,si,j,d,s)}\,(x = e^{ck}) & = & (m, si, j, d, Control(ck, TA_{(m,si,j,d,s)}\,(x, e^{ck}))@s) \\
TEqList_{(m,si,j,d,s)}\,(eq) & = & TEq_{(m,si,j,d,s)}\,(eq) \\
TEqList_{(m,si,j,d,s)}\,(eq,l) & = & TEq_{\,TEqList_{(m,si,j,d,s)}\,(l)}\,(eq) \\
TP\,(\texttt{node}\ f(p) = q\ \texttt{with}\ \texttt{var}\ r\ \texttt{in}\ D) & = & \mathit{let}\ m, si, j, d, s = TEqList_{([],[],[],r,[])}\,(l)\ \mathit{in} \\
 & & \texttt{machine}\ f = \\
 & & \ \texttt{memory}\ m \\
 & & \ \texttt{instances}\ j \\
 & & \ \texttt{reset}() = si \\
 & & \ \texttt{step}\,(p)\,\texttt{returns}\,(q) = \texttt{var}\ d\ \texttt{in}\ JoinList(s) \\
 & & \text{where } l \in Sch\,(D)
\end{array}
$$

Figure 5. The Translation Function

```
memory x2: int
instances x4: count
reset () =
  x4.reset ();
  state(x2) = 0;
step(c:bool;i:int) returns (o:int)
  var x3:int in
  case(c) { True: o = x4.step(i);
            False: o = state(x2) };
  state(x2) = o;
```

This example illustrates the memory model that is used for the generated code: it is essentially a tree structure, each machine allocating the memory for its sub-machines. There is no dynamic allocation of memory, thus conforming with what is practiced on most of safety critical embedded applications.

6. Target code generation

The intermediate language of Section 4 can be quite naturally translated into either a full-fledged object-oriented language or into a low-level imperative language. Our main interest lies in the generation of C code which is the traditional target of compilers of synchronous languages. Moreover, a compiler for C has been recently certified in COQ [3] which should make it possible to develop a complete certified compiler from LUSTRE to assembly

code. Nonetheless, in order to illustrate the versatility of the intermediate language, we also consider JAVA code generation.

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, e.g, 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 field 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 answer 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:

```
public class counting {
  boolean x_1;
  int x_2;

  public void reset() {
    x_1 = true;
    x_2 = 0; }

  public int step(boolean tick, boolean top) {
    int o; int x_3; int v; boolean b;
    b = x_1;
    x_1 = false;
    if (top) {v = 1;} else {v = 0;}
    if (b) {x_3 = 0;} else {x_3 = x_2 + v;}
    if (tick) {o = v;} else {o = x_3;}
    x_2 = o;
    return o; }}
```

6.2 Translation into C

The C code generator follows the principles already demonstrated by the RELUC compiler. [3] 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 for representing the internal state of each object. 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 answer type of the `step` function is again represented as a structure to allow tuples to be returned. [4] 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:

```
typedef struct {
  int x_1; int x_2; } counting_mem;

void counting_reset(counting_mem *self) {
  self->x_1 = 1;
  self->x_2 = 0; }

int counting_step(int tick, int top,
                  counting_mem *self) {
int o; int x_3; int v; int b;
b = self->x_1;
self->x_1 = 0;
if (top) {v = 1;} else {v = 0;}
if (b) {x_3 = 0;} else {x_3 = x_2 + v;}
if (tick) {o = v;} else {o = x_3;}
self->x_2 = o;
return o; }
```

[3] RELUC is a prototype compiler developed at *Esterel Technologies*; it is used as an implementation reference for the next SCADE generation.

[4] As a matter of fact, RELUC differs from our approach in the way multiple outputs are handled. In RELUC a memory structure is extended with an appropriate number of fields for storing the outputs.

7. The Complete Compiler

Based on the material presented in this article, we have built a compiler to serve as a reference implementation for a certified compiler. The goal was to make it as small as possible and mostly based on local program transformations. Figure 6 describes the various parts of the compiler with the corresponding numbers of lines of OCAML code. Another version has also been implemented in the programming language of COQ, but only up to the generation of the object-based intermediate language. We have also implemented three classical optimizations directly on the clocked-data flow language: inlining, dead-code removal, and automata minimization – the general form of common-subexpression elimination. They are all defined as source-to-source transformations and largely benefit from the data-flow nature of the language. Finally, language extensions proposed in [6] have been implemented as well.

The source language we have presented 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. The two specificities are the n-ary *merge* (instead of the *current* operator of LUSTRE) and a modular *reset* construct (also absent in LUSTRE). The *merge* is used to combine n complementary streams and introduces a general notion of clocks. The *reset* is used to restart the behavior of a node. These two constructions can be encoded in 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 [6], a conservative extension of LUSTRE with hierarchical state automata is proposed, based on a translation semantics into a clocked data-flow kernel similar to the one considered in this article. The *merge* and *reset* constructs are used extensively in this encoding. The authors advocate that such a translation not only gives the semantics of the whole language, but is an effective way to implement the compiler in the sense that the generated code is good in terms of size and efficiency. This solution has been integrated in the RELUC compiler of LUSTRE. Thus, the present article completes this work and highlights the missing part of the compilation chain. Altogether, these results serve as the basis of SCADE 6, the next version of SCADE.

The code generation is done after type checking, clock checking, and specific static analyzes such as causality or initialization analysis. If one of these steps fails, the compilation process stops. Type checking is almost standard [20]. The clock calculus rejects programs that cannot be executed synchronously and is defined as a type inference problem [7]. The causality analysis checks the absence of instantaneous loops in order to ensure that a static schedule is feasible. Finally, the initialization analysis checks that the behavior does not depend on the initial values of delays [8]. At the end of these analyzes, the program is annotated with type and clock information. Then, constructs that are not part of the data-flow kernel (e.g., control structures such as activation conditions or state machines) are translated into the clocked data-flow kernel.

In Section 1, we have stressed the importance of modular compilation for separate compilation, code tracability, as well as to keep

		Ocaml (LOC)
Administrative code	abstract syntax + printers	546
	lexer & parser	335
	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)	arround 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 6. MiniLustre in Numbers

the size of the generated code linear in the size of the source program. The price to pay is an extra constraint on feedback loops that must explicitly cross a delay (not nested inside nodes). Thus, in practice, modular compilation affects the causality analysis which has to reject semantically correct programs because they cannot be compiled modularly. To avoid this restriction, an industrial compiler such as the one present in the SCADE-Suite proposes to inline, on user demand, specific nodes of the model. This feature can also be used to find a good compromise in terms of *program size/program speed* (as any compiler optimizer silently does). This explains why it is important to complement a synchronous compiler with an inliner. Note that such an inliner is a trivial task in LUSTRE thanks to its substitution principle. The other solution in order to avoid the restriction on feedback loops would rely on the decomposition mechanism proposed by Raymond [21]. Taking dependences between inputs and outputs, a stream function is decomposed into a minimal number of atomic functions, each of them leading to a single transition function. Then, every call to a non atomic function is statically inlined when it appears in a feedback loop. Such a solution is complementary to what is considered in the present paper and would be inserted as a pre-processing step in the whole compilation chain.

In Section 5 we have presented a control optimization which gathers two consecutive control structures on the same guard. There are other optimizations that can be implemented in this translation, particularly around the scheduling policy. The role of scheduling is to transform a partially ordered set of equations into a sequence of assignments. The solution is not unique in general, and we can take advantage of the freedom to favor certain optimizations. For instance, the scheduling can contain heuristics that try to schedule consecutively equations that are guarded by the same clock. Then, the merging of consecutive control structures will be able to factorize more control conditions. Another classical optimization is related to the reuse of variables (which corresponds to removing *copy* variables in classical compilation terminology [19]). As mentioned in [15], 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`. The RELUC compiler as well as the reference compiler we have developed to support the present article implement a scheduling heuristics for that purpose.

8. Related Work

This article is related to the work done on the code generation of synchronous languages and in particular LUSTRE and SIGNAL. We have already pointed out the differences with the academic compiler of LUSTRE. The distinction with SIGNAL comes from the different expressiveness of our source language and its associated clock calculus. For example, the language does not allow the expression of relations as SIGNAL does, but only functions, and is based on a Kahn semantics. Moreover, we use a simpler clock calculus based on ML-type inference whereas the clock calculus of SIGNAL calls for boolean resolution [1, 11]. In our language, it is for example not possible to express the disjunctive clocks of the form $ck_1 \vee ck_2$ (stating that a value is present if one of the two clocks is true) as is SIGNAL. Clocks are only of the form `base on` c_1 `on` ... `on` c_n, and they correspond directly to nested control structures. 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. It would be interesting to know if the resulting code would coincide with the one obtained here with simpler but dedicated techniques.

This work is connected also with the works on the DC format [14] and its extension DC+ [9] introduced for the compilation of synchronous languages. The DC format allows for similar control properties as the source language which we consider. However, as the author in [14] 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 often related to code distribution (see [12] 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.

9. Conclusion and Future Work

This article has presented the code generation of a synchronous data-flow language into imperative code. This code generation is modular in the sense that each node definition is translated into an independent pair of imperative functions. The principles presented in this article are implemented in the RELUC compiler of SCADE/LUSTRE and experimented on industrial real-size examples. However, their precise description has never been published or described before. Such a formalization now appears as a fundamental need in order to develop a certified compiler for a synchronous language in a proof assistant, as well as to simplify existing implementations. Moreover, it offers an opportunity to improve process-based certification as used today by SCADE customer with a stronger mathematical argument of certification using proof techniques.

Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[3] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

[4] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.

[5] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VERIMAG tech. report no. 97–07 at `www.lri.fr/~pouzet`.

[6] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[7] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[8] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):245–255, August 2004.

[9] Sacres consortium. The declarative code dc+ , version 1.4. Technical report, Esprit project EP 20897 : Sacres, 1997.

[10] The coq proof assistant, 2007. http://coq.inria.fr.

[11] Thierry Gautier and Paul Le Guernic. Code generation in the sacres project. In *Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99*, pages 127–149, Huntingdon, UK, Feb 1999. Springer.

[12] Alain Girault. A survey of automatic distribution method for synchronous programs. In *International Workshop on Synchronous Languages, Applications and Programs (SLAP)*, Edinburg, UK, April 2005. ENTCS.

[13] Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones*. PhD thesis, Université Paris VI, Paris, 1988.

[14] N. Halbwachs. *The declarative code DC, version 1.2a*. Vérimag, Grenoble, France, October 1995. unpublished report.

[15] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[16] N. Halbwachs and Pascal Raymond. A tutorial of lustre. `http://www-verimag.imag.fr/SYNCHRONE/`, 2002.

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

[18] The MathWorks. http://www.mathworks.com/products/simulink.

[19] Steven S Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[20] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[21] Pascal Raymond. Compilation séparée de programmes lustre. Technical report, Projet SPECTRE, IMAG, juillet 1988.

[22] SCADE. `http://www.esterel-technologies.com/scade/`, 2007.