Clock-directed Modular Code Generation
from Synchronous Block Diagrams

Dariusz Biernacki
INRIA Futurs
Orsay, France

Jean-Louis Colaco
Siemens VDO
Toulouse, France

Marc Pouzet
LRI, Univ. Paris-Sud 11
Orsay, France

ABSTRACT

The compilation of synchronous block diagrams into sequential imperative code has been addressed in the early eighties and can be considered now as folklore. However, separate or modular code generation, though largely used in existing compilers and particularly in industrial ones, has been neither precisely described nor entirely formalized. Such a formalization appears now as a fundamental need 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 object-based intermediate language to represent transition functions and we present a concise clock-directed translation function from the source to the intermediate language. We also address the target code generation phase by describing a translation from the intermediate language to JAVA and C.

1. INTRODUCTION

Block diagram formalisms as found in SIMULINK [18] or SCADELUSTRE [22] are widely used for embedded system design. Among them, 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 be considered now as a part of the original folklore in synchronous programming [2].

Given a stream function \( f : \text{Stream}(T) \rightarrow \text{Stream}(T) \) and a stream equation \( y = f(x) \), the code generation consists in producing a pair \((f_t, s_0)\) made of a transition function of type \( S \rightarrow T \rightarrow T \times S \) and an initial state \( s_0 \) of type \( S \) such that \( \forall n \in \mathbb{N}, s_{n+1} = f_t s_n x_n \) if \( x = (x_i)_{i\in\mathbb{N}} \) and \( y = (y_i)_{i\in\mathbb{N}} \). The transition function takes a state and the current input and 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 \( \text{Stream}(T) \) is represented by a scalar value of type \( T \) and no intermediate memory nor complex buffering mechanism is needed. These principles generalize to functions with multiple inputs and multiple outputs.

Code generation is obtained through a static scheduling of equations according to data dependences. 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. Nonetheless, modular code generation is not always feasible, even in the absence of causality loop, as illustrated by the equation \((y, z) = f(t, y) \) with \( f(x, y) = (x, y) \). 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 \( f \) is compiled. This observation has led to two different approaches to the compilation problem. One consists in compiling the program after a full inlining of function calls has been performed in order to keep the maximal expressiveness of the source language. 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, 14]. Unfortunately, this efficiency gain is 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 [15]) but finding the adequate variables which lead to efficient code in both time and size is difficult. Conversely, modular compilation is mandatory in industrial compilers like the one of SCAD/E. Each stream function is translated into an imperative function with no preliminary inlining unless requested by the programmer. Consequently, modular compilation imposes stronger causality constraints stating that every feedback

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright the authors, APGES 2007, Oct. 4th 2007, Salzburg, Austria.
loop must cross an explicit delay. These constraints are, nonetheless, well accepted by SCADE users. They are also justified by the need for traceability of the generated code, as required by certification authorities in the context of critical software.

Modular compilation of synchronous block diagrams, though largely used in the LUCID SYNCHRON [21] compiler or 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. Additionally, it complements previous work done on the formalization of static analysis (such as clock calculus [7] and initialization analysis [8]), general principles of compilation [5] and language extensions [17, 6].

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 [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. (Clocks express various activation conditions in the program). 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 good control structures. This approach is in contrast with 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.

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. Also, the implementation and proofs 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.

The article is organized as follows. In Section 2, we present a synchronous data-flow kernel. In Section 3, we address the issue of schedulability of a set of equations and of transforming it into a normal form. In Section 4, we define an intermediate sequential language for representing transition functions. In Section 5, we define the translation from the data-flow language to the intermediate language. In Section 6, we describe JAVA and C code generation from the intermediate language. In Section 7, we sketch the construction of the entire compiler. In Sections 8 and 9, we discuss related and future work and we conclude.

2. A Clocked DATA-FLOW LANGUAGE

We define a synchronous data-flow kernel considered as a basic calculus into which any Lustre program can be translated. Actually, we make it a little more general by equipping it with a means to reset a function application in a modular way, following [16] and we provide value constructors belonging to some enumerated types and filtering mechanisms as introduced in [6]. Moreover, the code generation being done after type and clock verification, we assume that every term is annotated with its proper type and clock information.

2.1 Syntax and Intuitive Semantics

A program is made of a list of global node declarations (d) and type declarations (td). A global node declaration is of the form node f(p) = p with var p in D. To simplify the presentation, only abstract and enumerated types are provided here. a stands for annotated expressions (e) with their clock (ct). Expressions are made of values (v), tuples (a_1, ..., a_n), initialized delays (v fby a), variables (x), pointwise applications (op(a_1, ..., a_n)), node instantiations with a possible reset condition (f(a_1, ..., a_n) every a), a combination operation (merge x (C_1 -> a_1) ... (C_n -> a_n)) or a sampling operation (a when C(x)). 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 (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 resettable 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., \_, \_, \_) to its argument 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).

A pattern pat may be a variable or a tuple of patterns (pat,...,pat). A declaration (D) can be a collection of parallel equations. An equation defines a value, and we assume the existence of an initial environment defining bool = False + True. In the same way, combinatorial functions are provided externally.

Clock annotations do not play any role in the data-flow semantics of the language so we omit them in the examples below. v fby a stands for the initialized delay. It is assumed that the first parameter of fby is an immediate value.

<table>
<thead>
<tr>
<th>x</th>
<th>x_0</th>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>y_0</td>
<td>y_1</td>
<td>y_2</td>
<td>y_3</td>
</tr>
<tr>
<td>v fby x</td>
<td>x_0</td>
<td>x_1</td>
<td>x_2</td>
<td></td>
</tr>
<tr>
<td>x + y</td>
<td>x_0 + y_0</td>
<td>x_1 + y_1</td>
<td>x_2 + y_2</td>
<td>x_3 + y_3</td>
</tr>
</tbody>
</table>
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 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 now 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:

\[
\begin{align*}
if \ x \ then \ e_2 \ else \ e_3 &= \text{merge } x \\
& \quad (\text{True } \rightarrow \ e_2 \ when \ True(x)) \\
& \quad (\text{False } \rightarrow \ e_3 \ when \ False(x)) \\
\end{align*}
\]

\[
\begin{align*}
c_1 \rightarrow e_2 &= \text{if } True \ then \ fby \ False \ then \ e_1 \\
& \quad \text{else } e_2 \\
\text{pre}(e) &= \text{nil} \ fby \ e \\
\end{align*}
\]

The conditional \( if/then/else \) is built from the merge operator and the sampling operator \( when \). The initialization operation \( pre(e) \) is a shortcut for \( \text{nil} \ fby \ e \) where \( \text{nil} \) stands for any constant value which has the type of \( e \). \(^1\)

\[
\begin{array}{c|ccc}
\emptyset & \text{True} & \text{False} & \ldots \\
\hline
x & x_0 & x_1 & x_2 & x_3 & \ldots \\
y & y_0 & y_1 & y_2 & y_3 & \ldots \\
x \rightarrow y & x_0 & y_1 & y_2 & y_3 & \ldots \\
\text{pre}(x) & \text{nil} & x_0 & x_1 & x_2 & \ldots \\
z = x \ when \ True(h) & x_0 & \ldots & \ldots \\
t = y \ when \ False(h) & y_1 & y_2 & \ldots \\
\text{merge } h & x_0 & x_1 & x_2 & y_3 & \ldots \\
& \quad (\text{True } \rightarrow \ z) \\
& \quad (\text{False } \rightarrow t)
\end{array}
\]

Forgetting clock annotation, the counting example of Figure 1 is written:

\[
\begin{align*}
\text{node counting} \ (\text{tick:bool; top:bool}) &= (o:\text{int}) \text{ with } \\
\text{var } v \ : \ \text{int in} \\
o &= \text{if tick then } v \ else \ 0 \rightarrow \text{pre } o + v \\
\text{and } v &= \text{if top then } 1 \ else \ 0;
\end{align*}
\]

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.

\(^1\)Then, it is the purpose of the initialization analysis to check that the computation result does not depend on the actual \( \text{nil} \) value.
3. TOWARDS SEQUENTIAL CODE

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

3.1 Syntactic Dependences and Scheduling

Following the definition introduced in [14], we say that an expression \( a \) is **statically** depends on \( x \) if \( x \) appears free in \( a \) and not as an argument of a delay \( \text{fby} \). \( \text{Left}(a) \) returns the set of variables appearing this way in \( a \) (we overload the notation for \( \text{Left}(e) \) and \( \text{Left}(D) \)). \( \text{Def}(D) \) defines the set of variables defined in \( D \). If \( \text{pat} = a \) is an equation in \( D \), every variable from \( \text{pat} \) immediately depends on variables from \( \text{Left}(a) \). The transitive closure of this relation defines the notion of static dependence. A program is causal when for each node the corresponding graph of dependencies is acyclic.

\[
\begin{align*}
\text{Left}(e^{ck}) &= \text{Left}(e) \cup \text{Vars}(ck) \\
\text{Left}(v \text{ fby} a) &= \emptyset \\
\text{Def}(pat = v \text{ fby} a) &= \text{Vars}(pat) \\
\text{Def}(\text{Left}(a)) &= \text{Def}(D_1) \cup \text{Left}(D_2) \\
\text{Vars}(x) &= \{x\} \\
\text{Vars}((pat_1, ..., pat_n)) &= \bigcup_{1 \leq i \leq n} \text{Vars}(pat_i)
\end{align*}
\]

An equation \( pat = a \) from a set of equations \( D \) is ready ((\( \text{pat} = a \in \text{Right}(D) \)) when it does not depend on any other equations. We make a particular treatment of equations of the form \( \text{pat} = (v \text{ fby} a)^{ck} \). Indeed, in this case, \( \text{pat} \) corresponds to a memory so it will have to be scheduled after any other computation reading variables from \( \text{pat} \).

\[
\begin{align*}
\text{pat} = v \text{ fby} a^{ck} &\in \text{Right}(D) \\
\text{pat} = a &\in \text{Right}(D)
\end{align*}
\]

We write \( D|_{\text{pat}=e} \) for the exclusion of the equation \( \text{pat} = e \) from \( D \). A sequence of equations \( \{ i = \text{pat}_1 = e_1, ..., \text{pat}_n = e_n \} \) is a feasible schedule of \( D \) if \( i \in \text{Sch}(D) \), where:

\[
\begin{align*}
\text{pat} = a &\in \text{Sch} \quad \text{pat} = a \\
\text{pat} = a, l &\in \text{Sch} \quad \text{pat} = a \quad \text{pat} = a
\end{align*}
\]

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

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

3.2 Putting Equations in Normal Form

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

\[
z = (((4 \text{ fby} a) * 3) \text{ when True}(c)) \text{ if True}(c) + k)^{ck} \text{ on True}(c)
\]

and \( o = (\text{merge} c \text{ when } 5 \text{ fby } (z + 1) + 2) \text{ on True}(c) \)

is rewritten into:

\[
z = (((t_1 + 3) \text{ when True}(c)) + k)^{ck} \text{ on True}(c)
\]

and \( o = (\text{merge} c \text{ when } 5 \text{ fby } (z + 1))^{ck} \text{ on True}(c) \)

and \( t_1 = (4 \text{ fby} a)^{ck} \)

and \( t_2 = (5 \text{ fby } (z + 1))^{ck} \text{ on True}(c) \)

and \( t_3 = (6 \text{ fby} x)^{ck} \text{ on False}(c) \)

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

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

\[
\begin{align*}
a &::= e^{ck} \\
e &::= a \text{ when } C(x) \mid op(a, ..., a) \mid x \mid v \\
ce &::= \text{merge} x (C \rightarrow ca) \mid (C \rightarrow ca) \mid e \\
ca &::= ce^{ck} \\
eq &::= x = (v \text{ fby} a)^{ck} \\
D &::= D \text{ and } D \mid eq
\end{align*}
\]

The extraction is straightforward and not detailed here. In the remainder we assume that equations have been normalized.

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 re-definition of auxiliary definitions.

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 paradigm. We are not interested in inheritance and 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 (in the sense of encapsulation) 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 like C or JAVA. 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 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 state variables. One difference with respect to object orientation is the absence of dynamic object creation since block diagrams we have considered are not recursive.

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 the assignment of a local variable (x := c) or of a state variable (state (x) := c), a sequence (S; S), the re-initialization method invocation of an object (o.reset), the invocation of the step method of object o (o.step (e1, ..., en)), a void statement (skip) or a control structure (case (x) {C1 : S1; ...; Cn : Sn}). If x is of type a = C1 + ... + C + ... + Cn, we shall write case (x) {C : S} or case (x) {C1 : skip ...; C : S; ...; Cn : skip} indifferently. An expression (e) can be either the access to a local variable (x) or to a state variable (state (x)), an immediate integer constant (i) or a value constructor (C), a tuple (e1, ..., en) or a function call (f(e1, ..., en)). A class (f) defines a set of memories (m), a set of instances for objects used inside the body of the methods step or reset and these two methods.

\[ d ::= \text{fun } f(p) \text{ returns } (p) = \text{var } p \text{ in } S | \text{class } f = \text{memory } m \text{ instances } j \text{ reset() returns () = } S \text{ step(p) returns (p) = var } p \text{ in } S \]

\[ S ::= x ::= c | \text{state (x) ::= c | } S \text{ | skip | o.reset (} (x,...,x) = o.step (e,...,e) | \text{case (x) { } C : S; } \ldots; C : S) \]

\[ e ::= x \text{ | v | state (x) | op(e,...,e) } \]

\[ v ::= C | i \]

\[ j ::= o : f ; \ldots ; o : f \]

\[ p, m ::= x : t ; \ldots ; x : t \]

5. THE TRANSLATION

The translation closely follows the principle of the iterative semantics described in [5]. 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. Moreover, we restrict it to the first-order case.

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

\[ \text{Join (case } (x) \{ C1 : S1; \ldots; Cn : Sn \}, \]

\[ \text{case } (x) \{ C1 : S1; \ldots; Cn : Sn \} \]

\[ = \text{case } (x) \{ \text{Join (S1, S1′); ...; } Cn : \text{Join (Sn, Sn′)} \} \]

\[ \text{Join (S1, S2) = S1; S2} \]

\[ \text{JoinList (S)} = S \]

\[ \text{JoinList (S1, ..., Sn) = Join (S1, JoinList (S2, ..., Sn))} \]

The translation is defined by a set of mutually recursive functions. \( TE(m, s, i, j, d, s) \) defines the translation of an un-annotated expression in a context \((m, s, i, j, d, s)\) and returns an expression from the target language \( c \). We overload the notation for annotated expressions. \( m \) stands for a memory environment, \( s \) 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. \( TA(m, s, i, j, d, s) \) defines the translation of an expression which is stored into \( x \) and it returns a new context. \( TEq(m, s, i, j, d, s) \) (eq) defines the translation of an equation. We use two auxiliary functions: the operation \( TELIST(m, s, i, j, d, s) \) (l) translates a list of expressions and returns a list of expressions from the target language, whereas \( TELIST(m, s, i, j, d, s) \) (l) translates a list of equations.

The definitions of the translation functions are given in Figure 2. The first six rules apply to stateless expressions. The translation of the 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 \((f(a1, ..., an)) \) every \( x \) is an object of class \( f \). 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 \( x = (v \text{ for } y) \) 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 to translate them iteratively.

6. TARGET CODE GENERATION

The intermediate language of Section 4 can be quite naturally translated into either a fully-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 consider also 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 class definition is translated into a JAVA class definition with two methods step and reset. The state variables...
6.2 Translation into C

The C code generator follows the principles already demonstrated by the ReLuC compiler.\textsuperscript{2} For each class, 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 again is represented as a structure that allows for tuples to be returned.\textsuperscript{3} 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 x_1; int x_2;
} counting_mem;

typedef struct {
    int o; int x_3; int v; boolean b;
    boolean x_1;
} step

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

void void counting_reset(counting_mem *self) {
    ...
}
```

\textsuperscript{2}ReLuC is a prototype compiler developed at Estrel Technologies; it is used as an implementation reference for the next SCADGE generation.

\textsuperscript{3}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.
In Section 1, we have stressed the importance of modular compilation for separate compilation, code traceability and to keep the size of the source program. The price to pay is an extra constraint on feedback loops which 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.

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 bunch 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 which 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 [14], a stream \( x \) and its previous value \( \text{pre}_x \) can be stored in the same variable if the computation of \( x \) is not followed by a use of \( \text{pre}_x \). 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. DC AND DC+

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 to express relations as SIGNAL does but only functions. Moreover, we use a simpler clock calculus based on ML-type inference whereas the clock calculus of SIGNAL calls for boolean resolution [1, 11]. It is not possible, for example, to express in our language the disjunctive clocks of SIGNAL of the form \( c_k \lor c_k \) (stating that a value is present if one of the two clocks is true). Clocks are only of the form \( \text{base} \) on \( c_1 \) on \( c_2 \) on \( c_n \) and they correspond directly to nested control-structures. The introduction of an n-ary \( \text{merge} \) and the general form of clocks presented here does not seem to have been considered in SIGNAL. Whereas this construction could be encoded in SIGNAL, obtaining good code would call for the full expressiveness of its clock calculus. 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 [13] and its extension DC+ [9] introduced for the compilation of synchronous languages. The DC format allows for the control properties that the source language, we consider, does. However, as the author in [13] points out,

```c
self->x_1 = 1;
self->x_2 = 0;

int counting_step(int tick, int top,
    counting_mem *self) {
    int s; int x_3; int v; int b;
    b = self->x_3;
    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;
}
```
DC was not considered 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 SCADÉ as a development tool.

Finally, code generation is often related to code distribution (see [12] for a survey and most recent references). It does not seem, however, 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 have been in use for several years in the compilers of LUCID SYNCHRONE and the ReLuC compiler of Scade/Lustre and have been experimented with on various real-size examples. However, their precise description has never been published or described before. Such a formalization appears now 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 replace process-based certification as used today by SCADÉ customer with a stronger mathematical argument of certification using proof techniques.

Acknowledgements: We thank Alexandre Bertails, Malgorzata Biernacka, Florence Plateau and the anonymous reviewers for useful comments on the presentation of this work.

10. REFERENCES


