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 SCADE/LUSTRE [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].

[*]This work started while the author was at ESTEREL-TECHNOLOGIES.

[†]This work was partially supported by the French ACI Sécurité Alidecs.

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 are not made or distributed for profit or commercial advantage and 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.

Given a stream function $f : \mathit{Stream}(T) \rightarrow \mathit{Stream}(T')$ 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 $\forall n \in \mathbb{N}.y_n, s_{n+1} = f_t\, s_n\, x_n$ 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 $\mathit{Stream}(T)$ 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 SCADE. 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 loop must cross an explicit delay. These constraints are, nonetheless, well accepted by SCADE users. They are also justified by the need for *tracability* 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 SYNCHRONE [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

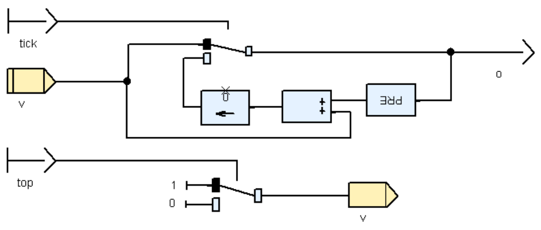
A program is made of a list of global node declarations (d) and type declarations (td). A global node declaration is of the form $\texttt{node}\ f(p) = p\ \texttt{with var}\ p\ \texttt{in}\ D$. To simplify the presentation, only abstract and enumerated types are provided here. a stands for annotated expressions (e) with their clock (ct). Expressions are made of values (v), tuples $(a_1, ..., a_n)$, initialized delays ($v\ \texttt{fby}\ a$), variables (x), point-wise applications ($op\,(a_1, ..., a_n)$), node instantiations with a possible reset condition ($f\,(a_1, ..., a_n)\ \texttt{every}\, a$), a combination operation ($\texttt{merge}\ x\ (C_1 \rightarrow a_1)\ ...\ (C_n \rightarrow a_n)$) or a sampling operation ($a\ \texttt{when}\ C(x)$). The expression $a\ \texttt{when}\ C(x)$ is the sampled stream of a on the instants where x equals C. Symmetrically, $\texttt{merge}$ 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. $f\,(a_1, ..., a_n)\ \texttt{every}\, a$ 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 argument and $f\,(a_1, ..., a_n)\ \texttt{every}\,\texttt{False}$ 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 ($pat = a$). To simplify the presentation, the boolean type is not explicitly given and we assume the existence of an initial environment defining $\texttt{bool} = \texttt{False} + \texttt{True}$. In the same way, combinatorial functions are provided externally.

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

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

x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
$v\ \texttt{fby}\ x$	v	x_0	x_1	x_2	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	...



```
-- 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

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). 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:

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

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]

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 annotation, the counting example of Figure 1 is written:

```
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.

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

Clocks do not have to be explicitly given in the source language, e.g., the programmer writes $(v \texttt{ fby } x) + y$ 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 $(\texttt{True} \rightarrow z)(\texttt{False} \rightarrow t)$ 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 self-contained, we present the associated clock conditions which must be verified by annotated terms. In a real implementation, the clock calculus apply to unannotated terms and produces annotated terms (the clock calculus shown here is based on [6]). We define several judgements to express that a program is well clocked. For instance, the judgement $H \vdash e : ct$ states that the expression e has a clock type ct under the clock environment H, whereas the judgement $H \vdash D$ 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 $i \neq j$.

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

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

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

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

$$\frac{H \vdash a_1 : ct_1 \ \dots \ H \vdash a_n : ct_n}{H \vdash (a_1, ..., a_n) : ct_1 \times \dots \times ct_n} \qquad \frac{H \vdash v : ck}{H, x : ck \vdash x : ck}$$

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

$$H, x : ck \vdash x : ck \qquad \frac{H \vdash pat_1 : ct_1 \ \dots \ H \vdash pat_n : ct_n}{H \vdash (pat_1, ..., pat_n) : ct_1 \times \dots \times ct_n}$$

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

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

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

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. 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 *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. 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 dependence. A program is causal when for each node the corresponding graph of dependencies is acyclic.

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

An equation $pat = a$ from a set of equations D 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 $pat = (v\ \mathtt{fby}\ a)^{ck}$. Indeed, in this case, pat corresponds to a memory so it will have to be scheduled after any other computation reading variables from pat.

$$\begin{array}{lll}
(pat = v\ \mathtt{fby}\ a^{ck}) \in R\,(D) & \mathit{if} & \mathit{Vars}(pat) \cap \mathit{Left}\,(D) = \emptyset \\
(pat = a) \in R\,(D) & \mathit{if} & \mathit{Left}\,(a) \cap \mathit{Def}\,(D) = \emptyset
\end{array}$$

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 $l \in \mathit{Sch}\,(D)$, where:

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

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):

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

is rewritten into:

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

In the same way, node instances ($f\,(a_1, ..., a_n)\ \mathtt{every}\,e$) 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.

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

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 redefinition 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 ($\texttt{state}\,(x) := c$), a sequence ($S\,;\,S$), the re-initialization method invocation of an object o ($o.\texttt{reset}$), the 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 $a = C_1 + ... + C + ... + C_n$, we shall write $\texttt{case}\ (x)\ \{C : S\}$ or $\texttt{case}\ (x)\ \{C_1 : \texttt{skip}; ...; C : S; ...; C_n : \texttt{skip}\}$ indifferently. An expression (e) can be either the 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 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.

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

5. THE TRANSLATION

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

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 for all i, j such that $1 \leq i \leq n$, $1 \leq j \leq k$, $x_i \neq x'_j$. $[\,]$ denotes the empty substitution. 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 $s_1 @ s_2 = S_1, ..., S_n, S'_1, ..., S'_k$ for their concatenation.

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

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

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

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

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 2. 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 $(f\ (a_1, ..., a_n)\ \texttt{every}\ x)^{ck}$, we introduce a fresh name o which 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 $x = (v\ \texttt{fby}\ a)^{ck}$ 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 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

$$
\begin{array}{lcl}
\mathit{TE}_{(m,si,j,d,s)}\,(e^{ct}) & = & \mathit{TE}_{(m,si,j,d,s)}\,(e) \\
\mathit{TE}_{(m,si,j,d,s)}\,(v) & = & v \\
\mathit{TE}_{(m,si,j,d+[x:t],s)}\,(x) & = & x \\
\mathit{TE}_{(m+[x:t],si,j,d,s)}\,(x) & = & \texttt{state}\,(x) \\
\mathit{TE}_{(m,si,j,d,s)}\,(\mathit{op}(a_1,...,a_n)) & = & \mathit{let}\ c_1,...,c_n = \mathit{TEList}_{(m,si,j,d,s)}\,(a_1,...,a_n)\ \mathit{in}\ \mathit{op}(c_1,...,c_n) \\
\mathit{TE}_{(m,si,j,d,s)}\,(a\ \texttt{when}\ C(x)) & = & \mathit{TE}_{(m,si,j,d,s)}\,(a) \\
 & & \\
\mathit{TEList}_{(m,si,j,d,s)}\,(a_1,...,a_n) & = & (\mathit{TE}_{(m,si,j,d,s)}\,(a_1),...,\ \mathit{TE}_{(m,si,j,d,s)}\,(a_n)) \\
 & & \\
\mathit{TAList}_{(m,si,j,d,s)}\,(x_1,...,x_n)(ca_1,...,ca_n) & = & \mathit{let}\ (m_1,si_1,j_1,d_1,s_1) = \mathit{TA}_{(m,si,j,d,s)}\,(x_1,ca_1)\ \mathit{in} \\
 & & ...\,\mathit{TA}_{(m_{n-1},si_{n-1},j_{n-1}d_{n-1},s_{n-1})}\,(x_n,ca_n) \\
\mathit{TA}_{(m,si,j,d,s)}\,(y,(\texttt{merge}\ x\ (C_1 \rightarrow ca_1)...(C_n \rightarrow ca_n))^{ck}) & = & \mathit{TAList}_{(m,si,j,d,s)}\,(y,...,y)(ca_1,...,ca_n) \\
\mathit{TA}_{(m,si,j,d,s)}\,(x,e^{ck}) & = & (m,si,j,d,\mathit{Control}(ck,x\ :=\ \mathit{TE}_{(m,si,j,d,s)}\,(e))) \\
 & & \\
\mathit{TEq}_{(m,si,j,d,s)}\,(x = ca) & = & \mathit{TA}_{(m,si,j,d,s)}\,(x,ca) \\
\mathit{TEq}_{(m,si,j,d+[x:t],s)}\,(x = (v\ \texttt{fby}\ a)^{ck}) & = & \mathit{let}\ c = \mathit{TE}_{(m,si,j,d,s)}\,(a)\ \mathit{in} \\
 & & (m+[x:t],[\texttt{state}\,(x)\ :=\ v]@si,j,d, \\
 & & [\mathit{Control}(ck,\texttt{state}\,(x)\ :=\ c)]@s) \\
\mathit{TEq}_{(m,si,j,d,s)}\,((x_1,...,x_k) = (f\ (a_1,...,a_n)\ \texttt{every}\,x)^{ck}) & = & \mathit{let}\ (c_1,...,c_n) = \mathit{TEList}_{(m,si,j,d,s)}\,(a_1,...,a_n)\ \mathit{in} \\
 & & (m,[o.\texttt{reset}]@si,[(o,f)]+j,d, \\
 & & \mathit{Control}(ck,\texttt{case}\ (x)\ \{(\texttt{True} : o.\texttt{reset})\})@ \\
 & & \mathit{Control}(ck,(x_1,...,x_n) = o.\texttt{step}\,(c_1,\ \ldots,\ c_n))@s) \\
 & & \text{where } o \notin \mathit{Dom}(j) \\
 & & \\
\mathit{TEqList}_{(m,si,j,d,s)}\,(eq) & = & \mathit{TEq}_{(m,si,j,d,s)}\,(eq) \\
\mathit{TEqList}_{(m,si,j,d,s)}\,(eq,l) & = & \mathit{TEq}_{\mathit{TEqList}_{(m,si,j,d,s)}\,(l)}\,(eq) \\
 & & \\
\mathit{TP}\,(\texttt{node}\ f(p) = q\ \texttt{with}\ \texttt{var}\ r\ \texttt{in}\ D) & = & \mathit{let}\ m,si,j,d,s = \mathit{TEq}_{([],[],[],r,[])}\,(l)\ \mathit{in} \\
 & & \texttt{class}\ f = \\
 & & \quad \texttt{memory}\ m \\
 & & \quad \texttt{instances}\ j \\
 & & \quad \texttt{reset}()\,\texttt{returns}\,()\ =\ si \\
 & & \quad \texttt{step}\,(p)\,\texttt{returns}\,(q) = \texttt{var}\ d\ \texttt{in}\ \mathit{JoinList}(s) \\
 & & \text{where } l \in \mathit{Sch}\,(D)
\end{array}
$$

Figure 2: The Translation Function

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. [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. [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:

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

void counting_reset(counting_mem *self) {
```

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

[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.

```
    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; }
```

7. TOWARDS A COMPLETE COMPILER

In this section, we discuss the organization of the entire compiler as well as possible extensions of the technique proposed in this article.

The source language we have presented is a first-order data-flow language similar to LUSTRE. Nonetheless, it exhibits specific constructions which make it both a good target for implementing extensions of LUSTRE as well as a good input language to generate 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 which 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 then the generated code is inefficient or calls for complex optimization techniques to cancel the effect of the encoding. Providing *merge* and *reset* as basic primitives allows for a more direct and efficient compilation.

In [6], we have proposed a conservative extension of LUSTRE with hierarchical state automata, basing it on a translation semantics into the clocked data-flow kernel considered in the present article. The *merge* and *reset* constructs were used extensively in this encoding. We advocated that this translation not only gives the semantics of the whole language but appears to be an effective way to implement the compiler in the sense that the generated code is reasonably good in terms of size and efficiency. This solution has been integrated in the ReLuC compiler of LUSTRE and the LUCID SYNCHRONE compiler. 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 which 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 and to keep 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 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 `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. 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 $ck_1 \vee ck_2$ (stating that a value is present if one of the two clocks is true). 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. 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, 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 SCADE 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 SCADE customer with a stronger mathematical argument of certification using proof techniques.

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

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

[14] 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.

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

[16] Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[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] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006. Distribution available at: `www.lri.fr/∼pouzet/lucid-synchrone`.

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