A Conservative Extension of Synchronous Data-flow with State Machines [*]

Jean-Louis Colaço
Esterel-Technologies
France

Bruno Pagano
Esterel-Technologies
France

Marc Pouzet
LRI, Université Paris-Sud
France

ABSTRACT

This paper presents an extension of a synchronous data-flow language such as LUSTRE with imperative features expressed in terms of powerful state machine *à la* SYNCCHART. This extension is fully *conservative* in the sense that *all the programs* from the basic language still make sense in the extended language and their semantics is preserved.

From a syntactical point of view this extension consists in hierarchical state machines that may carry at each hierarchy level a bunch of equations. This proposition is an alternative to the joint use of SIMULINK and STATEFLOW but improves it by allowing a fine grain mix of both styles.

The central idea of the paper is to base this extension on the use of *clocks*, translating imperative constructs into well clocked data-flow programs from the basic language. This clock directed approach is an easy way to define a semantics for the extension, it is light to implement in an existing compiler and experiments show that the generated code compete favorably with *ad-hoc* techniques. The proposed extension has been implemented in the ReLuC compiler of SCADE/LUSTRE and in the LUCID SYNCHRONE compiler.

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

General Terms

Design, Languages, Theory

Keywords

Synchronous languages. Heterogeneous systems. Typing. Clock calculus. Compilation.

[*]This work was carried out in the context of the IMCAD (Improving the Cockpit Application Development Process) project. IMCAD is a shared-cost RTD project (Contract No G4RD-CT-2001-00632) funded by the European Community Under the Competitive and Sustainable Growth Programme.

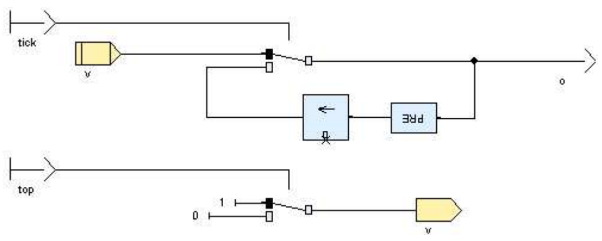
1. INTRODUCTION

When implementing a critical real-time embedded system, the designer has to chose its favorite programming or design tools. When the application is *data-flow dominated* (e.g., regulation systems), he will naturally go for block diagram formalisms as provided by SIMULINK [20], SCADE/LUSTRE [11] or SIGNAL [2]. On the contrary, when the application is more *control dominated* (e.g., drivers or protocols), imperative or automata based formalisms as provided by STATEFLOW [20], STATECHARTS [13], the SYNCCHART [1] or ESTEREL [3] will certainly be better choices. Nonetheless, real systems rarely fall into one category and are often a mix of both styles. A typical example is a flight by wire application where control laws are established in a data-flow style for each flight phase (take-off, landing, etc), transitions from one law to the other being specified in term of an automaton.

This has conducted people to propose multi-paradigm solutions allowing to use a dedicated language or formalism for each aspect [14, 18, 6, 5, 4] and relying on a linking phase to obtain the final application. This is typically what provides commercial tools like SIMULINK and STATEFLOW: a SIMULINK block-diagram may contain operators specified in STATEFLOW and which compute some flow which control the active parts of the system. ESTEREL-TECHNOLOGIES proposes similar solutions in its SCADE Suite, using SYNCCHART as the language for describing state machines. PTOLEMYII also provides mean to describe mixed systems made of data-flow equations and finite state machines. It goes even further by allowing various models of computations and communications (e.g., Kahn process networks, communicating sequential processes).

Nonetheless, this approach often lacks a unified semantics which applied for the complete application. It forces a strong separation of the two parts of the system at very early stages of the design. Moreover, it works well when small automata switch big control laws but is less satisfactory on more mixed designs. When using different code generation tools for each part, the ability to produce good software (readable *and* efficient) is reduced and finally, on a safety critical project where certification is required, the number of formalisms and tools makes the job harder.

Our purpose is to define a unique language able to go from a pure data-flow application to a pure control one in an integrated way. Compared to the above mentioned approaches, our solution simplifies the work of the designer by allowing to directly write data-flow equations into the states where they are active. Moreover, the presence of an automaton is



```
-- count the number of top between two tick
node counting (tick:bool; top:bool)
returns (o: bool);
  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

not limited to the leafs of the description but can appear at any level of the model hierarchy. This is a *key* feature when designing complex systems.

Following the pioneering work by Maraninchi & Rémond on *Mode-automata* [15, 16], we propose to provide imperative constructs at a *language level* by extending a data-flow language with imperative constructs. Our contribution is the following. We propose an extension of a synchronous data-flow language such as Lustre with a rich set of imperative features by means of state machines. These state machines allows to express various kinds of transitions (weak and strong transitions with possible reset of states). Moreover, this extension if fully conservative in the sense that *all the programs* from the basic language still make sense in the extended language and their semantics is preserved. This is an essential feature for an integration into an industrial tool such as Scade. In comparison, *Mode-automata* allows a limited form of transitions (essentially weak transitions corresponding to Moore automata) and impose restrictions on the form of data-flow programs appearing into states (e.g., clocks and node instances are forbidden). The central idea is to base the extension on the use of *clocks*, translating imperative constructs into well clocked data-flow programs. This clock directed approach enjoys several nice properties. It forces to give a very precise semantics of the extension and is of great help when adapting existing static analysis (e.g., type and clock calculus); it is light to implement in an existing compilers allowing to reuse the existing code generator; practical experiments show that the generated code compete favorably with hand-written code or *ad-hoc* compilation techniques [17]. Finally, the existing compilation techniques for data-flow is understood enough to be accepted by the certification authorities. Extending the language in a way that preserves this feature is a key point for us. Thus, this work is the basis for the evolution of the existing Scade Suite solution.

The paper is organized as follows. Section 2 gives the main intuitions of the proposed extension. Section 3 introduces a basic synchronous data-flow language and reminds its type and clock systems. Then it defines a programming language by adding two kinds of imperative constructs, a switch-like statement allowing to switch between sets of equations according to some enumerated value and an automaton construction. We define a *translation semantics* into the basic language and extend the type and clock systems such that any well typed (and clocked) program from the extended language is translated into a well clocked program from the basic language. Section 4 discuss other program analysis and we conclude in section 5.

2. OVERVIEW

Consider a purely synchronous data-flow language such as Lustre, taken as a basic language. This language allows for the description of block diagrams such as a simple counter as given in figure 1 (the left part show its graphical Scade representation). This program counts the number of `top` between two `tick`. Nonetheless, for the purpose of an integration of automata constructs, we make this basic language a little more general. We equip it with a mean to *reset* a function application in a modular way following [12], we provide *value constructors* belonging to some enumerated types and a merging operator for combining exclusive flows. We can provide synchronous semantics and type systems for such a language. Moreover, efficient compilation techniques exist [19] and this last point is not addressed in the present paper.

Based on this language we propose to extend it with hierarchical state machines. The example in figure 2 illustrates this extension on the well known case of the chronometer. This is a two button device: `StSt` stands for start/stop button and `Rst` is the reset one. It is supposed to be a cyclic application with a 10ms cycle-time. The flows `disp_1` and `disp_2` are the two fields of the display. Time counting starts and stops using the button `StSt`. When the chronometer is stopped, pressing the `Rst` button resets it. When the counter is running, pressing `Rst` freezes the display, giving a lap time. The solution we implement here contains two state machines in parallel, one has two hierarchy levels. The transitions are of two kinds: *strong* (arrow starting with a red bullet) or *weak* (arrows terminated by a blue bullet). These transitions can directly enter a state, which means that all its content must be *reset* when firing it or enter through an *history* connector H^* meaning that the target state is resumed instead of being reset. When a strong transition is fired, its target state is activated in the same synchronous tick; we use it to start and stop the chronometer. In the case of a weak one, the active state is the source one, which allow to have a condition that depends on internal computations without introducing a causality cycle. This example also introduces *shared* variables. For instance `s` is defined in states `STOP` and `START`. The `last s` notation allows to refer to the latest past value of a shared variable. The `d` flow is purely local and is delayed using the regular `pre` of Lustre. When no value is defined for a shared variable (`disp_1` and `disp_2` in `LAP`), the default action consists in maintaining the latest computed value (that is, `disp_1 = last disp_1`).

The semantics of the extension is obtained by translating automata construction into a purely data-flow kernel. Intuitively this translation is based on the following mapping:

- activation of a state is represented by a clock;
- exclusivity of the states of an automaton is encoded with exclusive clocks based on the use of an enumerated type;
- states hierarchy is represented by clocks hierarchy.

In LUSTRE, the clock of a flow characterizes the instants where the flow provides new values. Clocks give a powerful and safe way to control the activation of several block diagrams: a piece of data-flow networks can be controled by sampling its inputs according to a condition, that is, to set its input on some particular clock. Thanks to the clock calculus, the resulting description is guaranted to be synchronous.

3. FORMALIZATION

3.1 A Purely Data-flow Internal Language

We define a synchronous data-flow kernel considered as a basic calculus in which any LUSTRE program can be translated. Nonetheless, we make it a little more general. We equip it with a mean to *reset* a function application in a modular way following [12] and we provide *value constructors* belonging to some enumerated types. Finally, we do not impose to declare types nor clocks which are computed automatically.

A program is made of a sequence of global value declarations (d) and type declarations (td). A global value declaration may define either a constant stream ($\texttt{let}\ x = e$), a combinatorial function ($\texttt{let fun}\ x(p) = e$) or a node ($\texttt{let node}\ x(p) = e$). To simplify the presentation, only abstract types and enumerated types are provided here. Expressions (e) are made of value constructors (C) belonging to an enumerated type, initialised delays ($e_1\ \texttt{fby}\ e_2$), variables (x), applications ($x(e)$), pairs (e, e), local definitions ($\texttt{let}\ D\ \texttt{in}\ e$), sampling functions and a reset construct. $e_1\ \texttt{when}\ C(e_2)$ is the sampled stream of e_1 on the instants where e_2 equals C. Symmetrically, $\texttt{merge}$ is the combination operator: if e is a stream producing values belonging to a finite enumerated type $t = C_1 + ... + C_n$ and $e_1, ..., e_n$ are complementary streams, then it combines them to form a longer stream. $x(e)\ \texttt{every}\ c$ is the reset function application: the internal state of the application $x(e)$ is reset every time the boolean stream c is true.

A pattern p may be a variable or a pair pattern (p, p). A declaration (D) can be a collection of parallel equations. An equation can define some instantaneous values ($p = e$), *activation* (or *decision*) variables ($\texttt{clock}\ x = e$).

$$\begin{array}{lll}
e & ::= & C \mid x \mid e\ \texttt{fby}\ e \mid (e, e) \mid x(e) \\
 & & \mid \texttt{let}\ D\ \texttt{in}\ e \mid x(e)\ \texttt{every}\ e \\
 & & \mid e\ \texttt{when}\ C(e) \mid \texttt{merge}\ e\ (C \rightarrow e)\ ...\ (C \rightarrow e) \\
D & ::= & D\ \texttt{and}\ D \mid p = e \mid \texttt{clock}\ x = e \\
p & ::= & x \mid (p, p) \\
d & ::= & \texttt{let}\ x = e \mid \texttt{let fun}\ x(pat) = e \\
 & & \mid \texttt{let node}\ x(pat) = e \mid d; d \\
td & ::= & \texttt{type}\ t \mid \texttt{type}\ t = C_1 + ... + C_n \mid td; td
\end{array}$$

We note $fv(e)$ the set of free variables from an expression e. Its definition is straightforward and is not given here.

$e_1\ \texttt{fby}\ e_2$ stands for the initialised delay as depicted in the following diagram. Moreover, when x stands for a combinatorial function (typically an external function such as $\texttt{+}$, $\texttt{not}$), it applies point-wise to its arguments. If x and y are two streams, $+(x, y)$ stands for the point-wise addition that we write here in infix position.

x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
$x\ \texttt{fby}\ y$	x_0	y_0	y_1	y_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 $e\ \texttt{when}\ c$ of LUSTRE and LUCID SYNCHRONE where c is a boolean stream is written $e\ \texttt{when}\ \texttt{True}(c)$. In the same way, $e\ \texttt{when not}\ c$ is now written $e\ \texttt{when}\ \texttt{False}(c)$. The conditional $\texttt{if/then/else}$, the delay $\texttt{pre}$ and initialization operator $\texttt{->}$ of LUSTRE can be encoded in the following way:

$$\begin{array}{l}
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = \\
\quad \texttt{let clock}\ c = e_1\ \texttt{in} \\
\quad \texttt{merge}\ c\ \ (\texttt{True} \rightarrow e_2\ \texttt{when}\ \texttt{True}(c)) \\
\quad \qquad\quad (\texttt{False} \rightarrow e_3\ \texttt{when}\ \texttt{False}(c)) \\
\quad \text{where}\ c \notin FV(e_1) \cup FV(e_2) \\
e_1\ \texttt{->}\ e_2 = \texttt{if True fby False then}\ e_1\ \texttt{else}\ e_2 \\
\texttt{pre}\,(e) = nil\ \texttt{fby}\ e
\end{array}$$

The conditional $\texttt{if/then/else}$ is built from the $\texttt{merge}$ operator and the sampling operator $\texttt{when}$ provided we define $\texttt{type bool} = \texttt{True} + \texttt{False}$. The uninitialized delay operation $\texttt{pre}\,(e)$ is a shortcut for $nil\ \texttt{fby}\ e$ where nil stands for any constant value which has the type of e. We should rely on some initialization analysis to know whether the computation depends on the actual nil value [9]. The $\texttt{let/clock}$ construction is used for introducing the *decision variable* c which is used for sampling streams.

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

A program is a collection of stream functions defined globally. In doing this, we make a distinction between *combinatorial* functions whose definitions begins with a $\texttt{let/fun}$ and *stateful* functions ($\texttt{let/node}$). These two type of functions exist in synchronous data-flow languages such as LUSTRE, SCADE and LUCID SYNCHRONE: a combinatorial function is a function typically imported from a host language (e.g., C) and applies point-wise to its argument whereas a node has an internal state and must thus be compiled in a special way. These two kind of stream functions will receive different types as we shall see later.

For example, the node counting the number of `top` between two `tick` shall be written:

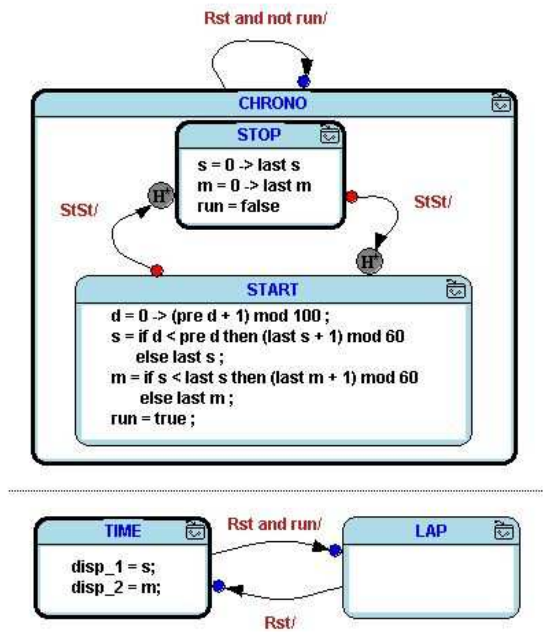
```
let node counting (tick, top) =
  let o = if tick then v else 0 -> pre o + v
  and v = if top then 1 else 0 in o
```

Figure 1 shows how this simple example can be represented in SCADE and gives the equivalent LUSTRE text.

The semantics of this kernel can be defined precisely following classical formulations and we do not come back on it here. See [8] for a denotational Kahn semantics. The semantics of the reset follows the proposal in [12]. The only novelties come from the sampling mechanism on enumerated types and these semantics can be adapted accordingly.

3.1.1 The Type System

This kernel is statically typed and we give it a ML-like type system. We distinguish type schemes (σ) which can be quantified from regular types (t). A regular type is made of



```
let node chrono (StSt, Rst) = (disp_1, disp_2) where
  automaton
    CHRONO ->
      do automaton
           STOP ->
             do  s = 0 -> last s
             and m = 0 -> last m
             and run = false
             unless StSt continue START
         | START ->
             let d = 0 -> (pre d + 1) mod 100 in
             do s = if d < pre d
                    then (last s + 1) mod 60
                    else last s
             and m = if s < last s
                     then (last m + 1) mod 60
                     else last m
             and run = true
             unless StSt continue STOP
         end
      until Rst and not run then CHRONO
  end and
  automaton
    TIME ->
      do disp_1 = s
      and disp_2 = m
      until Rst and run then LAP
  | LAP ->
      do until Rst then TIME
  end
```

Figure 2: A Chronometer

basic atomic types (B), combinatorial function types ($t \xrightarrow{0} t$) and sequential function types ($t \xrightarrow{1} t$), product types ($t \times t$) and type variables (α).

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, ..., \alpha_n . t \\ t & ::= & B \mid t \xrightarrow{k} t \mid t \times t \mid \alpha \\ B & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \\ k & ::= & 0 \mid 1 \\ H & ::= & [x_1 : \sigma_1, ..., x_n : \sigma_n] \end{array}$$

Typing is obtained by asserting judgments of the form $H \overset{k}{\vdash} e : t$ and $H \overset{k}{\vdash} D : H_0$. The first one states that "the expression e has type t in environment H". The second one states that "the definition D is well typed in H and produces the typing environment H_0". $k = 0$ means that the expression is combinatorial (no internal state is modified during the computation) whereas $k = 1$ stands for a statefull expression.

Typing is made in an initial environment H_0 such that:

$$\begin{array}{rl} H_0 = [& . \texttt{fby} . : \forall \alpha . \alpha \times \alpha \xrightarrow{1} \alpha, \\ & \texttt{pre}\,(.) : \forall \alpha . \alpha \xrightarrow{1} \alpha, \\ & \texttt{if} . \texttt{then} . \texttt{else} . : \forall \alpha . \texttt{bool} \times \alpha \times \alpha \xrightarrow{0} \alpha] \end{array}$$

Due to lack of space, the typing rules for expressions and declarations are not reproduced in this paper. They can be easily deduced from usual presentation [7]. For the node `counting`, the compiler automatically compute type $\texttt{bool} \times \texttt{bool} \xrightarrow{1} \texttt{int}$.

3.1.2 The Clock System

The type system checks the data consistency of a program, that is, what is transported on a channel. The purpose of the clock calculus is to check the time consistency, that is, when a value is available. The clock calculus produces judgments of the form $H \vdash e : cl$ for expressions and $H \vdash D : H_0$ for definitions. $H \vdash e : cl$ means that "the expression e has clock cl in the environment H". $H \vdash D : H_0$ means that "the definition D defines the local environment H_0 under the environment H".

$$\begin{array}{lcl} \rho & ::= & \forall \alpha_1, ..., \alpha_n . \forall X_1, ..., X_m . cl \mid cl \\ cl & ::= & cl \rightarrow cl \mid cl \times cl \mid (c : ck) \mid ck \\ ck & ::= & \texttt{base} \mid \alpha \mid ck \ \texttt{on} \ C(c) \\ c & ::= & X \mid m \\ H & ::= & [x_1 : \rho_1, ..., x_n : \rho_n] \end{array}$$

We distinguish clock schemes (ρ) that can be quantified, from regular clocks types (cl). A regular clock type is made of function clocks ($cl \rightarrow cl$), product clocks ($cl \times cl$), a dependence ($c : ck$), and stream clocks (ck). A stream clock may be the base clock (`base`), a clock variable (α), a sampled clock (ck `on` $C(c)$) on a condition c. Here, c can be either a skolem name (m) or a meta-variable(X).

$FV(cl)$ defines the set of free variables (α) of cl. $Dom(H)$ is the domain of H. $FN(cl)$ stands for the set of free variables (X). $N(cl)$ is the set of names (x) of cl. Their definitions are straightforward and not given here.

An expression e_1 `when` $C(e_2)$ is well clocked if e_1 and e_2 have the same clock α. In that case, the clock of the result is a sub-clock of α that we write α `on` $C(c)$ if c stands for the value of e_2. The clock type $(c : ck)$ given to an expression e says two things: e is present on clock ck and has the value c.

Clocks can be instantiated or generalized in the following way:

$$\begin{array}{lcl} cl[\vec{ck}/\vec{\alpha}][\vec{c}/\vec{X}] & \leq & \forall \vec{\alpha} . \vec{X} . cl \\ gen_H(cl) & = & \forall \alpha_1, ..., \alpha_n . \forall X_1, ..., X_m . cl \text{ where} \\ & & \{\alpha_1, ..., \alpha_n\} = FV(cl) - FV(H) \text{ and} \\ & & \{X_1, ..., X_m\} = FN(cl) - FN(H) \\ genall(cl) & = & gen_{\emptyset}(cl) \end{array}$$

The clock calculus is done in an initial environment H_0. As for the typing, we give clock types to the basic primitives:

$$\begin{array}{rl} H_0 = [& . \texttt{fby} . : \forall \alpha . \alpha \times \alpha \rightarrow \alpha, \\ & \texttt{pre}\,(.) : \forall \alpha . \alpha \rightarrow \alpha, \\ & \texttt{if} . \texttt{then} . \texttt{else} . : \forall \alpha . \alpha \times \alpha \times \alpha \rightarrow \alpha] \end{array}$$

If d is a definition, then $H \vdash d : H'$ builds a global clocking environment H' from the definition d. Its definition is the following.

$$\frac{H_0 \vdash p : cl_1 \quad H + H_0 \vdash e : cl_2 \quad Dom(H_0) = fv(p)}{H \vdash \texttt{let fun}\ f(p) = e : [genall(cl_1 \to cl_2)/f]}$$

$$\frac{H_0 \vdash p : cl_1 \quad H + H_0 \vdash e : cl_2 \quad Dom(H_0) = fv(p)}{H \vdash \texttt{let node}\ f(p) = e : [genall(cl_1 \to cl_2)/f]}$$

$$\frac{H \vdash e : cl}{H \vdash \texttt{let}\ x = e : [genall(cl)/f]}$$

$$\frac{H \vdash d_1 : H_1 \quad H + H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1 + H_2}$$

The clocking rules for expressions and declarations are defined in figure 3.

The system states that a constructor C may receive any stream clock ck. The clock of a variable can be instantiated. In this calculus, variables which serve as *activation variables* are introduced with a special keyword `clock`, following the presentation of [8]. These variables can in turn be used to down-sample an other stream, and we give a unique symbolic value m so that only streams down-sampled with the same source name m can be considered to be synchronous. The clock rules for definitions, applications and functions are straitforward. An expression e_1 `when` $C(e_2)$ produces a stream which is on some clock ck `on` $C(c)$ if e_1 has some clock type ck and e_2 has clock type $(c : ck)$. The `merge` construction expect streams which are on opposite clocks ck `on` $C_i(c)$. Note that when `bool` = `False` + `True`, we obtain the classical rule for the `merge` operator given in [8], replacing the original notation `merge` $e\ e_1\ e_2$ by `merge` e (`True` $\to e_1$) (`False` $\to e_2$).

3.2 The Programming Language

In this section, we introduce a programming language which extends the basic one with control structures.

Expressions (e) are extended with a mean to access the last value of a shared variable (`last` x). A declaration may now introduce local names (`let` D_1 `in` D_2), a case statement (`match` x `with` $C_1 \to D$ ... $C_n \to D$), a reset definition (`reset` D `every` e) or an automaton. Every handler $S \to u\ s$ in an automaton is made of two parts. (u) defines a set of *shared* variables (`do` $D\ w$) which may use some auxiliary local names (`let` D `in` u) and a set of weak conditions (w) to escape from the handler. (s) stands for the set of strong escape conditions.

$$\begin{array}{lll}
e & ::= & C \mid x \mid e\ \texttt{fby}\ e \mid (e,e) \mid x(e) \mid \texttt{last}\ x \\
 & & \mid \texttt{let}\ D\ \texttt{in}\ e \mid x(e)\ \texttt{every}\ e \\
 & & \mid e\ \texttt{when}\ C(e) \mid \texttt{merge}\ e\ (C \to e)\ ...\ (C \to e) \\
D & ::= & D\ \texttt{and}\ D \mid p = e \mid \texttt{clock}\ x = e \mid \texttt{let}\ D\ \texttt{in}\ D \\
 & & \mid \texttt{match}\ e\ \texttt{with}\ C \to D\ ...\ C \to D \\
 & & \mid \texttt{reset}\ D\ \texttt{every}\ e \\
 & & \mid \texttt{automaton}\ S \to u\ s\ ...\ S \to u\ s \\
u & ::= & \texttt{let}\ D\ \texttt{in}\ u \mid \texttt{do}\ D\ w \\
s & ::= & \texttt{unless}\ e\ \texttt{then}\ S\ s \mid \texttt{unless}\ e\ \texttt{continue}\ S\ s \mid \epsilon \\
w & ::= & \texttt{until}\ e\ \texttt{then}\ S\ w \mid \texttt{until}\ e\ \texttt{continue}\ S\ w \mid \epsilon
\end{array}$$

We define some auxiliary definitions. $Def(D)$ stands for the defined names from D. The function $fv(.)$ is now lifted to definitions such that $fv(D)$ stands for the set of free variables from D. Their definitions are given in appendix B.

3.2.1 Translation Semantics

In this section, we introduce a *translation semantics* for the extended language, that is, a semantics where new programming constructs are expressed in terms of the basic ones. For this purpose, we define a translation function $T(.)$ which applies recursively to expressions and declarations. In order to keep the notation light, we overload the notation. The actual interpretation of $T(.)$ shall be clear from context. The general structure of the translation is given in figure 4. It is essentially a morphism translating every new programming constructs (i.e., `match/with`, `automaton`, `reset/every`, `last`) in terms of the basic ones and leaving the other constructs unchanged. For the translation, we introduce three *translation combinators* whose definition is detailed below.

3.2.1.1 Reseting a Behavior

The translation of the `reset/every` construction consists in propagating the reset construct recursively. When arriving at an application, it transforms it into a reset application. Let $CReset_x\ D$ be the result of this translation applied to a definition D reset on some boolean condition x and $CResE_x\ e$, the result of the translation applied to expressions. We have:

$$\begin{array}{lll}
CReset_x\ (D_1\ \texttt{and}\ D_2) & = & D'_1\ \texttt{and}\ D'_2 \\
 & & \text{where } D'_1 = CReset_x\ D_1 \\
 & & \text{and } D'_2 = CReset_x\ D_2 \\
CReset_x\ (p = e) & = & p = CResE_x\ e \\
CReset_x\ (\texttt{clock}\ y = e) & = & \texttt{clock}\ y = CResE_x\ e \\
CResE_x\ (y(e)) & = & y(e')\ \texttt{every}\ x \\
 & & \text{where } e' = CResE_x\ e \\
CResE_x\ (y(e_1)\ \texttt{every}\ e_2) & = & y(e'_1)\ \texttt{every}\ x\ \texttt{or}\ e'_2 \\
 & & \text{where } e'_1 = CResE_x\ e_1 \\
 & & \text{and } e'_2 = CResE_x\ e_2 \\
CResE_x\ (e_1\ \texttt{fby}\ e_2) & = & \texttt{let}\ y = CResE_x\ e_1\ \texttt{in} \\
 & & \texttt{if}\ x\ \texttt{then}\ y \\
 & & \texttt{else}\ y\ \texttt{fby}\ (CReset_x\ e_2) \\
 & & \text{where } y \notin fv(e_1) \cup fv(e_2) \\
CResE_x\ (\texttt{let}\ D\ \texttt{in}\ e) & = & \texttt{let}\ D'\ \texttt{in}\ CReset_x\ e \\
 & & \text{where } D' = CReset_x\ D \\
CResE_x\ C & = & C \\
CResE_x\ y & = & y \\
CResE_x\ (e_1\ \texttt{when}\ C(e_2)) & = & e'_1\ \texttt{when}\ C(e'_2) \\
 & & \text{where } e'_1 = CResE_x\ e_1 \\
 & & \text{and } e'_2 = CResE_x\ e_2 \\
CResE_x\ (e_1, e_2) & = & (CResE_x\ e_1, CResE_x\ e_2)
\end{array}$$

$$\begin{array}{l}
CResE_x\ \texttt{merge}\ e\ (C_1 \to e_1)(C_n \to e_n) = \\
\qquad \texttt{merge}\ e'\ (C_1 \to e'_1)(C_n \to e'_n) \\
\qquad \text{where } e' = CResE_x\ e \text{ and } e'_i = CResE_x\ e_i
\end{array}$$

The reset condition x is distributed recursively in every definition or sub-expression. The interesting cases appear with applications and initialized delays. In the former case, an application is transformed into a reset application. Moreover, an already reset application on some expression e_2 can now be also reset on the condition x. In the later, an initialised delay e_1 `fby` e_2 is transformed into a conditional which emits the initial value when x is true.

3.2.1.2 Activation Conditions over Enumerated Types

The `match/with` construction is expressed as a combination of `merge` and `when` constructions, associating an equation for every variable it defines. For this purpose, we define $proj_y^{C(x)}(D)$ as the projection of D according to y and on

$$H \vdash C : s \qquad \frac{H + H_0 \vdash D : H_0 \quad H + H_0 \vdash e : cl}{H \vdash \texttt{let}\, D \,\texttt{in}\, e : cl} \qquad \frac{H \vdash e : ck \quad m \notin N(H)}{H \vdash \texttt{clock}\; x = e : [(m : ck)/x]} \qquad \frac{H \vdash p : cl \quad H \vdash e : cl}{H \vdash p = e : [cl/p]}$$

$$\frac{cl \leq H(x)}{H \vdash x : cl} \qquad \frac{H \vdash D_1 : H_1 \quad H \vdash D_2 : H_2}{H \vdash D_1 \,\texttt{and}\, D_2 : H_1 + H_2} \qquad \frac{H \vdash e_1 : cl_2 \rightarrow cl_1 \quad H \vdash e_2 : cl_2}{H \vdash e_1(e_2) : cl_1} \qquad \frac{H \vdash e_1 : cl_1 \quad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2}$$

$$\frac{H \vdash e_1 : ck \quad H \vdash e_2 : (c : ck)}{H \vdash e_1 \;\texttt{when}\; C(e_2) : ck \;\texttt{on}\; C(c)} \qquad \frac{H \vdash e : (c : ck) \quad \forall i \quad 1 \leq i \leq n \quad H \vdash e_i : ck \;\texttt{on}\; C_i(c)}{H \vdash \texttt{merge}\; e\; (C_1 \rightarrow e_1) ... (C_n \rightarrow e_n) : ck} \qquad \frac{H \vdash x(e_1) : cl \quad H \vdash e_2 : ck}{H \vdash x(e_1) \;\texttt{every}\; e_2 : cl}$$

Figure 3: The kernel Clocking rules

$$\begin{array}{lcl}
T(x(e)) & = & x(T(e)) \\
T(e_1 \;\texttt{fby}\; e_2) & = & T(e_1) \;\texttt{fby}\; T(e_2) \\
T(x(e_1) \;\texttt{every}\; e_2) & = & x((T(e_1))) \;\texttt{every}\; (T(e_2)) \\
T(e_1 \;\texttt{when}\; C(e_2)) & = & T(e_1) \;\texttt{when}\; C(T(e_2)) \\
T(\texttt{merge}\; e\; (C_1 \rightarrow e_1) \;...\; (C_n \rightarrow e_n)) & = & \texttt{merge}\; T(e)\; (C_1 \rightarrow T(e_1)) \;...\; (C_n \rightarrow T(e_n)) \\
T(D_1 \;\texttt{and}\; D_2) & = & T(D_1) \;\texttt{and}\; T(D_2) \\
T(\texttt{clock}\; x = e) & = & \texttt{clock}\; x = T(e) \\
T(\texttt{let}\; D_1 \;\texttt{in}\; D_2) & = & T(D_1) \;\texttt{and}\; T(D_2) \text{ if } \mathit{Def}(D_2) \cap (\mathit{fv}(D_1) \cup \mathit{Def}(D_1)) = \emptyset \\
 & & \\
T_S(\texttt{let}\; D \;\texttt{in}\; u) & = & (D \;\texttt{and}\; D', se, re) \text{ where } se, re, D' = T_S(u) \\
 & & \\
T_S(\epsilon) & = & \emptyset, S, \texttt{False} \\
T_{S_0}(\texttt{until}\; e\;\; \texttt{then}\; S\; w) & = & (x = e \;\texttt{and}\; D, \texttt{if}\; x \;\texttt{then}\; S \;\texttt{else}\; se, \texttt{if}\; x \;\texttt{then}\; \texttt{True} \;\texttt{else}\; re) \\
 & & \text{where } x \notin \mathit{fv}(u) \cup \mathit{fv}(e) \text{ and } D, se, re = T_{S_0}(w) \\
T_{S_0}(\texttt{until}\; e\;\; \texttt{continue}\; S\; w) & = & (x = e \;\texttt{and}\; D, \texttt{if}\; x \;\texttt{then}\; S \;\texttt{else}\; se, \texttt{if}\; x \;\texttt{then}\; \texttt{False} \;\texttt{else}\; re) \\
 & & \text{where } x \notin \mathit{fv}(u) \cup \mathit{fv}(e) \text{ and } D, se, re = T_{S_0}(w) \\
T_{S_0}(\texttt{unless}\; e\;\; \texttt{then}\; S\; s) & = & (x = e \;\texttt{and}\; D, \texttt{if}\; x \;\texttt{then}\; S \;\texttt{else}\; se, \texttt{if}\; x \;\texttt{then}\; \texttt{True} \;\texttt{else}\; re) \\
 & & \text{where } x \notin \mathit{fv}(u) \cup \mathit{fv}(e) \text{ and } D, se, re = T_{S_0}(s) \\
T_{S_0}(\texttt{unless}\; e\;\; \texttt{continue}\; S\; s) & = & (x = e \;\texttt{and}\; D, \texttt{if}\; x \;\texttt{then}\; S \;\texttt{else}\; se, \texttt{if}\; x \;\texttt{then}\; \texttt{False} \;\texttt{else}\; re) \\
 & & \text{where } x \notin \mathit{fv}(u) \cup \mathit{fv}(e) \text{ and } D, se, re = T_{S_0}(s) \\
 & & \\
T(\texttt{reset}\; D \;\texttt{every}\; e) & = & \texttt{let}\; x = T(e) \;\texttt{in}\; \mathit{CReset}_x\;\; T(D) \text{ where } x \notin \mathit{fv}(D) \cup \mathit{fv}(e) \\
T(\texttt{match}\; e \;\texttt{with}\; C_1 \rightarrow D_1 \;...\; C_n \rightarrow D_n) & = & \mathit{CMatch}\; (T(e))\; (C_1 \rightarrow (T(D_1), \mathit{Def}(D_1))) \;...\; (C_n \rightarrow (T(D_n), \mathit{Def}(D_n))) \\
T(\texttt{automaton}\; S_1 \rightarrow u_1\; s_1 \;...\; S_n \rightarrow u_n\; s_n) & = & \mathit{CAutomaton}\; (S_1 \rightarrow (T_{S_1}(u_1), T_{S_1}(s_1))) \;...\; (S_n \rightarrow (T_{S_n}(u_n), T_{S_n}(s_n)))
\end{array}$$

Figure 4: The Translation Semantics

clock condition $C(x)$ such that:

$$\begin{array}{lcll}
\mathit{proj}_y^{C(x)}(D) & = & e & \text{if } (y = e) \in D \\
 & = & (\texttt{pre}\,(y)) \;\texttt{when}\; C(x) & \text{otherwise}
\end{array}$$

If N is a set of names, we write $\mathit{Split}_N(D)$ as the partition of the declarations D into two sets D_1, D_2 such that $\mathit{Def}(D_2) = N$ and $D_1 \cup D_2 = D$. Finally, we define the function:

$$\begin{array}{lcl}
\mathit{COn}\; D\; C(x) & = & D\,[x_1 \;\texttt{when}\; C(x)/x_1, ..., \\
 & & \;x_n \;\texttt{when}\; C(x)/x_n \\
 & & \;(\texttt{pre}\,(x_1)) \;\texttt{when}\; C(x)/\texttt{last}\; x_1, ..., \\
 & & \;(\texttt{pre}\,(x_n)) \;\texttt{when}\; C(x)/\texttt{last}\; x_n] \\
 & & \text{where } \{x_1, ..., x_n\} = \mathit{fv}(D)
\end{array}$$

This operation consists in filtering all the free variables appearing in D as well as their last value. In figure 4, the translation of a `match/with` statement consists in the translation of its components and a call to the compilation combinator *CMatch* given in figure 5.

A free variable y read in a handler D_i is transformed into y `when` $C_i(x)$. Thus, it is observed on some clock ck `on` $C(x)$ provided that y and x have the same clock ck. An expression `last` y is transformed into $(\texttt{pre}\,(y))$ `when` $C(x)$ meaning that the expression $\texttt{pre}\,(y)$ has clock ck, that is a faster clock than the local clock ck `on` $C(x)$ of the handler. `last` y really means the previous value of y, the last time y has been *computed*. On the contrary, an expression $\texttt{pre}\,(y)$ inside a handler is transformed into an expression $\texttt{pre}\,(y \;\texttt{when}\; C(x))$. Thus, $\texttt{pre}\,(y)$ which stands for a *local memory* denotes the previous value of y on the clock of the handler, *not* on the clock x is defined. This is why we say that $\texttt{pre}\,(y)$ stands for the previous value of y, the last time y has been *observed*.

Note that the proposed translation only deals with `last` appearing inside a `match`; for the others present at top level this translation can be extended by replacing `last` by `pre`. Another choice is to define an analysis that reject the programs containing a `last` in a context where it behaves like a `pre`.

To illustrate the translation process, consider the following piece of code (given in concrete syntax where `do/done` separate shared variables from local declarations).

```
match x with
  Left -> let cpt = 1 -> last o1 + 1 in
          do o1 = 2 * cpt done
| Right -> do o2 = 1 -> pre o2 + 1
           and o1 = 0 done
end
```

`cpt` is a local variable whereas `o1` and `o2` are shared vari-

$$
\begin{array}{l}
\mathit{CMatch}\ (e)\ (C_1 \rightarrow (D_1, N_1))...(C_n \rightarrow (D_n, N_n)) = \\
\quad D'_1 \texttt{ and } ... \texttt{ and } D'_2 \texttt{ and} \\
\quad \texttt{clock } x = e \texttt{ and} \\
\quad y_1 = \texttt{merge } x \\
\qquad\qquad (C_1 \rightarrow \mathit{proj}_{y_1}^{C_1(x)}(G_1)) \\
\qquad\qquad ... \\
\qquad\qquad (C_n \rightarrow \mathit{proj}_{y_1}^{C_n(x)}(G_n)) \\
\quad \texttt{and } ... \texttt{ and} \\
\quad y_k = \texttt{merge } x \\
\qquad\qquad (C_1 \rightarrow \mathit{proj}_{y_k}^{C_1(x)}(G_k)) \\
\qquad\qquad ... \\
\qquad\qquad (C_n \rightarrow \mathit{proj}_{y_k}^{C_n(x)}(G_k)) \\
\text{where } \forall i, x \notin \mathit{fv}(e) \cup \mathit{fv}(D_i) \\
\text{and } \{y_1, ..., y_k\} = N_1 \cup ... \cup N_n \\
\text{and } \forall i D'_i, G_i = \mathit{Split}_{N_i}(\mathit{COn}\ D_i\ C_i(x))
\end{array}
$$

Figure 5: The translation of match

ables. This code is translated into:

```
    clock c = x
and cpt = 1 -> ((pre o1) when Left(c)) + 1
and o1 = merge c (Left -> 2 * cpt) (Right -> 0)
and o2 = merge c
          (Left -> (pre o2) when Right(c))
          (Right -> (1 -> pre (o2 when Right(c)) + 1))
```

This translation highlights the fact that `last o1` in the source program refers to the previous value of `o1` whereas `pre o2` refers to the value `o2` had, the last time `x` was equal to `Right`.

3.2.1.3 Automata

The automaton construction is translated by applying recursively the translation function to its components as described in figure 4. For this purpose, we have introduced the translation function for handlers. We write $T_C(u) = (D, es, er)$, $T_C(s) = (D, es, er)$ and $T_C(w) = (D, es, er)$ for translating the constructions u and escape conditions s and w. D stands for a set of declarations, es denotes an expression computing a state according to an escape condition and er denotes an expression computing a reset condition for the next state. The translation function is parameterized by the state name S of the handler. In case no escape occur, the state expression must return S.

The translation function is given in figure 6. An automaton is translated into two case statements: the first one computes what is the current state to be executed according to strong preemptions whereas the second one computes the equations in the current state and where to go at the next state. We introduce several auxiliary variables. s defines what is the current *state*; ns stands for the next state whereas pns stands for its previous value. r stands for a boolean value which is true if the current state must be *reset* on entry; nr is true if the state to be executed at the next reaction will have to be reset; pnr stands for its previous value. Moreover, we must add a new type definition of the form `type` $t = S_1 + ... + S_n$ provided there is no name conflict neither with existing type names nor with enumerated values (otherwise, some renaming should apply).

Observe that, as a consequence of this semantics, in an automaton only one set of equations is executed during a reaction. It is moreover possible to enter *strongly* in a state and leave it *weakly* (or conversely). Nonetheless, it is not

$$
\begin{array}{l}
\mathit{CAutomaton}\quad S_1 \rightarrow (D_1, es_1, er_1)\ (D'_1, es'_1, er_1)\ ... \quad = \\
\qquad\qquad\qquad S_n \rightarrow (D_n, es_n, er_n)\ (D'_n, es'_n, er_n) \\
\quad \texttt{match } \mathit{pns} \texttt{ with} \\
\qquad S_1 \rightarrow \texttt{reset } s = es'_1 \texttt{ and } r = er'_1 \texttt{ and } D'_1 \texttt{ every } \mathit{pnr} \\
\qquad ... \\
\qquad S_n \rightarrow \texttt{reset } s = es'_n \texttt{ and } r = er'_n \texttt{ and } D'_n \texttt{ every } \mathit{pnr} \\
\quad \texttt{and} \\
\quad \texttt{match } s \texttt{ with} \\
\qquad S_1 \rightarrow \texttt{reset } ns = sw_1 \texttt{ and } nr = rw_1 \texttt{ and } D_1 \texttt{ every } r \\
\qquad ... \\
\qquad S_n \rightarrow \texttt{reset } ns = sw_n \texttt{ and } nr = rw_n \texttt{ and } D_n \texttt{ every } r \\
\quad \texttt{and clock } \mathit{pns} = S_1 \texttt{ fby } ns \\
\quad \texttt{and clock } \mathit{pnr} = \texttt{False fby } nr \\
\text{where } \forall i.s, ns, r, nr \notin FV(es_i) \cup FV(er_i) \\
\qquad\qquad\qquad \cup FV(D_i) \cup FV(D'_i)
\end{array}
$$

Figure 6: The translation of automata

possible to enter and leave strongly during one reaction, that is, to cross more than two transitions. This is a *key* difference with the SyncChart or StateCharts, and largely simplifies program understanding and analysis.

3.2.2 The Type System

We should first extend the typing rule for the new programming constructs. The typing rule should mimic the translation semantics such that it gives the same types as the typing of the translation. These rules state in particular that newly introduced constructions are only allowed in a node (they are considered as state-full constructions). For the node `chrono`, the compiler automatically computes the type $\texttt{bool} \times \texttt{bool} \xrightarrow{1} \texttt{int} \times \texttt{int}$.

Typing does not raise any particular difficulty and we do not detail it here. When typing a program, it is possible to restrict the use of `last` x such that `last` x is only accepted when x is a shared variable, that is, a variable which is defined either in a case statement or an automaton. This way, there is no possible confusion between `last` x and `pre` (x) (it is not possible to write `last` x in a context where it behaves like `pre` (x)). There is technically no reason to restrict the use of `last` x since a pending `last` x can always be replaced by `pre` (x) once control structures have been translated into the basic language. We have experimented both solutions in our compilers. Experimenting real-size designs will help in choosing the more appropriate solution.

3.2.3 The Clock System

The clock calculus must be extended such that translated program can be accepted by the basic clock calculus and can thus be safely compiled. Remember that we have introduced the notation $\mathit{COn}\ D\ C(c)$ to say that every free variable in a block is observed on the local clock defined by the block. We now define $H\ \mathit{on}_{ck}\ C(c)$ to apply on clocking environment in order to simulate this process during the clock calculus. Consider for example a `match/with` statement which is itself executed on some clock ck. When entering in a branch, a free variable x with defined clock ck will be read on the sub-clock ck `on` $C(c)$ of ck.

$$(H\ \mathit{on}_{ck}\ C(c))(x) = H(x) \texttt{ on } C(c) \text{ provided } H(x) = ck$$

For example, if $H = [\alpha/x_1, \alpha/x_2]$ then $H\ \mathit{on}_\alpha\ (C(c) : \alpha)$ is an environment H' such that the clock information associated to x_1 in H' is α `on` $C(c)$. As a consequence, if a free variable has some sub-clock ck `on` $C'(c')$ instead of ck, then the translation of the program will result in a badly clocked program. This is why we impose that free variables be on the clock ck.

We define $merge(H_1, ..., H_n)$ as the merge of several environments returning a environment H such that:

- $Dom(H) = Dom(H_1) \cup ... \cup Dom(H_n)$
- $\forall x, \forall i, j.i \neq j, H_i(x) = cl_1 \wedge H_j(x) = cl_2 \Rightarrow cl_1 = cl_2$

We extend the previous system with the rules given in figure 7. The predicate $H \overset{ck}{\vdash} u : H'$ states that the automaton handler u produces the clock environment H' and has escape conditions on clock ck. The predicates $H \vdash w : ck$ and $H \vdash s : ck$ state that the weak and strong escape conditions are on clock ck.

Let us explain the clocking rule for `match/with` statements (the one for automata being similar). We first introduce a new symbol m which abstract the value of the test e which has itself clock ck. We say that shared variables in the handler must all be on clock ck `on` $C_i(m)$ provided that free variables are all red on clock ck `on` $C_i(m)$ also.

3.2.4 Type and Clock Preservation

We can state now that the program transformation preserves types and clocks. Precisely, any well typed (clocked) program in the extended language receive the same type (clock) as its translated version.

THEOREM 1 (CORRECTION). *For any expression e, declaration D and environment H we have:*

- $H \overset{k}{\vdash} e : ty$ *iff* $H \overset{k}{\vdash} T(e) : t$ *and* $H \overset{k}{\vdash} D : H'$ *iff* $H \overset{k}{\vdash} T(D) : H'$
- $H \vdash e : cl$ *iff* $H \vdash T(e) : cl$ *and* $H \vdash D : H'$ *iff* $H \vdash T(D) : H'$

The proof is made by induction on the proof derivations for types and clocks.

4. DISCUSSION

The proposed extension has been fully implemented both in the RELUC compiler at ESTEREL-TECHNOLOGIES and in the LUCID SYNCHRONE compiler. In both compilers, a program is first typed, clocked, then two other program analysis are applied. The causality analysis [10] first rejects programs which cannot be statically scheduled. Then, the initialization analysis rejects programs containing un-initialized delays [9]. Whereas we did not described these two analysis in the present paper, they have been modified according to the introduction of control structure. Once programs have passed these four analysis, the imperative constructs are translated into a basic language close to the one used in this paper and the resulting program is passed to the existing code generation phase.

We have experimented the proposed extension and approach on several examples ranging from simple ones to more complex ones with the following results. First of all, the generated code for programs combining data-flow equations *and* automata constructs is very good and compete favorably with hand-written code or *ad-hoc* compilation techniques as proposed in [17]: when `C` is the target language, the code generator of RELUC is able to generate the minimal number of `switch` statements such that only the instructions from the active state of an automaton are executed at every instant (the `C` code generated by the RELUC compiler for the chronometer is given in appendix A) . Thus, as far as code generation is concerned, no pertinent information has been lost during the programming transformation. This result comes from the fact that our transformation relies deeply on the use of *clocks* and the existing compilation techniques developed in both compilers are specifically tuned to optimize clocks. Second, the modification of both compilers was light and an important part of the compiler was left unchanged. The typing, clock calculus, causality analysis and initialization analysis have been slightly modified and a new pass was added to the compiler. This pass follows closely the proposal described in section 3.2.1.

The causality and initialization analysis has not been discussed in the present paper due to lack of space. These two analysis raise some interesting problems which are informally discussed here. In [9], we have proposed a modular analysis expressed as a type system for a language close to the basic language. Thus, one way to check the correct initialization of a complete program could be to apply the analysis once the translation has been performed. Nonetheless, this solution does not work for, at least, three reasons. First, it gives poor diagnostic in case of error. Second, it does not fit well in a graphical programming environment where systems must be analyzed on the fly interactively, thus rejecting complex program transformation. Finally, in the case of automata, precious information has been lost during the translation so that the existing initialization analysis may produce *false negative* when applied to the translated program. Consider for example:

```
let node two x = o where
  automaton
    S1 -> do o = 0 -> last o + 1 until x continue S2
  | S2 -> do o = last o - 1 until x continue S1
  end
```

`o` is always defined. Nonetheless, if we look at its translation (after a few simplifications), the information that `S1` is an initial state and thus, that `o` do have a value at the very first instant is no more clearly apparent.

```
let node two x = o where
      o = merge s
            (S1 -> 0 -> (pre o) when S1(s) + 1)
            (S2 -> (pre o) when S2(s) - 1)
  and ns = merge s
            (S1 -> if x when S1(s) then S2 else S1)
            (S2 -> if x when S2(s) then S1 else S2)
  and clock s = S1 -> pre ns
```

On such a program, the analysis presented in [9] states that `o` is not initialised because clock informations are not taken into account.

Nonetheless, checking that the initial program is correctly initialized comes from a simple deduction. The initial value of `o` is necessarily defined since the initial state is only weakly preempted. Thus, when the control enters in state `S2`, `o` do have a previous value. On the contrary, the following program must be statically rejected since `x` must be true at the very first instant:

```
let node two x = o where
  automaton
    S1 -> do o = 0 -> last o + 1 unless x continue S2
  | S2 -> do o = last o - 1 until x continue S1 end
```

The extension of the initialization analysis given in [9] has been implemented in both compilers. This is relatively simple because at most two successive transitions can be taken during one reaction (a strong preemption cannot be

$$\frac{cl \le \sigma}{H[x:\sigma] \vdash \texttt{last}\ x : cl} \qquad \frac{H \vdash e : ck \quad m \notin N(H) \quad H\ on_{ck}\ C_i(m) \vdash D_i : H_i\ on_{ck}\ C_i(m)}{H \vdash \texttt{match}\ e\ \texttt{with}\ C_1 \rightarrow D_1 ... C_n \rightarrow D_n : merge(H_1,...,H_n)} \qquad \frac{H \vdash e : ck \quad H \vdash D : H'}{H \vdash \texttt{reset}\ D\ \texttt{every}\ e : H'}$$

$$\frac{m \notin N(H) \quad H\ on_{ck}\ S_i(m) \overset{ck\ \texttt{on}\ S_i(m)}{\vdash} u_i : H_i\ on_{ck}\ S_i(m) \quad H\ on_{ck}\ S_i(m) \vdash s_i : ck\ \texttt{on}\ S_i(m)}{H \vdash \texttt{automaton}\ S_1 \rightarrow u_1\ s_1 \ldots S_n \rightarrow u_n\ s_n : merge(H_1,...,H_n)} \qquad \frac{H \vdash D : H_0 \quad H \vdash w : ck}{H \overset{ck}{\vdash} \texttt{do}\ D\ w : H_0}$$

$$\frac{H \vdash D_1 : H_1 \quad H + H_1 \vdash D_2 : H_2}{H \vdash \texttt{let}\ D_1\ \texttt{in}\ D_2 : H_2} \qquad \frac{H \vdash D_1 : H_1 \quad H + H_1 \overset{ck}{\vdash} u : H_2}{H \overset{ck}{\vdash} \texttt{let}\ D_1\ \texttt{in}\ u : H_2} \qquad \frac{H \vdash e : ck \quad H \vdash w : ck}{H \vdash \texttt{until}\ e\ \texttt{then}\ S\ w : ck}$$

$$H \vdash \epsilon : ck \qquad \frac{H \vdash e : ck \quad H \vdash w : s}{H \vdash \texttt{until}\ e\ \texttt{continue}\ S\ w : s} \qquad \frac{H \vdash e : ck \quad H \vdash w : ck}{H \vdash \texttt{unless}\ e\ \texttt{then}\ S\ w : ck} \qquad \frac{H \vdash e : ck \quad H \vdash w : ck}{H \vdash \texttt{unless}\ e\ \texttt{continue}\ S\ w : ck}$$

Figure 7: The Extended Clock System

followed by an other strong preemption). Intuitively, any shared variable `x` from an initial state with no strong preemption or belonging to the initial state and its immediate successor by a strong preemption is well initialised such that `last x` has a defined value in the remaining states.

We believe that the clock based approach proposed in this paper is general enough to apply to any data-flow language providing a clock mechanism and associated compilation methods. This is, in particular the case for SIGNAL. Such an experiment would be interesting because SIGNAL provides a richer clock calculus than the one used in the present paper. This may give the ability to relax the constraint imposed on free variables, namely that all the free variables accessed in an automaton must be on the same clock. Nonetheless, this restriction did not appear to be a limitation in all the real applications we have encountered so far.

5. CONCLUSION

In this paper, we have presented a conservative extension of a synchronous data-flow language such as LUSTRE with imperative constructs by means of state machines. This extension is *conservative* in the sense that it applies to the whole LUSTRE language without any restriction and this is a central point for being used in an industrial tool.

The proposed extension provides a rich way to mix data-flow equations and automata with strong or weak escape conditions. Yet, we made it simple enough to be usable in a *qualified compiler* (as it is the case for SCADE). This is why we have adopted a *clock-directed* approach defined as a source-to-source transformation. Starting from a basic LUSTRE-like kernel, we have extended it with control structures which are in turn translated into the basic kernel. This way, it is possible to use the existing code generation phase without modification. The resulting extension has been fully implemented in two compilers, the ReLuC compiler of SCADE at ESTEREL-TECHNOLOGIES and in the one of LUCID SYNCHRONE. Experimental results show that this approach leads to a very efficient code. The proposed extension will be integrated into SCADE-V6, the new version of SCADE.

6. REFERENCES

[1] Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *CESA*, Lille, july 1996. IEEE-SMC. Available at: `www-mips.unice.fr/∼andre/synccharts.html`.

[2] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. *Heterogeneous Concurrent Modeling and Design in Java*. Memorandum UCB/ERL M04/27, EECS, University of California, Berkeley, CA USA 94720, July 2004.

[5] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 1994. special issue on Simulation Software Development.

[6] Reinhard Budde, G. Michele Pinna, and Axel Poigné. Coordination of synchronous programs. In *International Conference on Coordination Languages and Models*, number 1594 in Lecture Notes in Computer Science, 1999.

[7] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[8] 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.

[9] 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.

[10] Pascal Cuoq and Marc Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] 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.

[13] D. Harel. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3:231–275, 1987.

[14] M. Jourdan, F. Lagnier, P. Raymond, and F. Maraninchi. A multiparadigm language for reactive systems. In *5th IEEE International Conference on Computer Languages*, Toulouse, May 1994. IEEE Computer Society Press.

[15] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.

[16] 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.

[17] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.

[18] Axel Poigné and Leszek Holenderski. On the combination of synchronous languages. In W.P.de Roever, editor, *Workshop on Compositionality: The Significant Difference*, volume LNCS 1536, pages 490–514, Malente, September 8-12 1997. Springer Verlag.

[19] Pascal Raymond. *Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-v3*. PhD thesis, Institut National Polytechnique de Grenoble, 1991.

[20] www.mathworks.com/.

APPENDIX

A. C CODE FOR THE CHRONOMETER

```
#define true (bool)1
#define false (bool)0
#define not true ^
typedef unsigned char bool;

typedef enum {__t1_none, __tSTOP_s1, __tSTART_s1} __Tt1;
typedef enum {__t3_none, __tTIME_w1, __tLAP_w1} __Tt3;
typedef enum {STOP, START} __TA1;
typedef enum {TIME, LAP} __TA3;
typedef enum {__t2_none, __tCHRONO_w1} __Tt2;

/* context */
typedef struct {
  /* outputs */
  int disp_1, disp_2;
  /* inits */
  bool _M_init3, _M_init2, _M_init1, _M_init0;
  /* memories */
  __TA3 __Next_SA3;
  __TA1 __SA1;
  __Tt2 __fired_w2;
  int m, _M_pre_d00, _M_pre_s1;
} C_CHRONO;

void CHRONO_init (C_CHRONO *C){
  C->_M_init3 = true; C->_M_init2 = true;
  C->_M_init1 = true; C->_M_init0 = true;
}

void CHRONO_cycle(const bool StSt, const bool Rst, C_CHRONO *C){
  bool __reset_CHRONO;
  int s, d0;
  __Tt1 __fired_s1; __TA3 __SA3;
  __TA1 __preNext_SA1; __Tt3 __fired_w3;

  __reset_CHRONO = (C->__fired_w2 == __tCHRONO_w1);
  if (C->_M_init0) {
    C->_M_init0 = false;
    __SA3 = TIME;
  } else __SA3 = C->__Next_SA3;
  if (__reset_CHRONO) {
    C->_M_init3 = true;
    C->_M_init2 = true;
    C->_M_init1 = true;
  }
  if (C->_M_init1) {
    C->_M_init1 = false;
    __preNext_SA1 = STOP;
  } else __preNext_SA1 = C->__SA1;
  switch (__preNext_SA1) {
  case START:
    if (StSt) __fired_s1 = __tSTART_s1;
    else __fired_s1 = __t1_none;
    switch (__fired_s1) {
    case __tSTART_s1:
      C->__SA1 = STOP;
      break;
    default:
      C->__SA1 = __preNext_SA1;
    }
    break;
  case STOP:
    if (StSt) __fired_s1 = __tSTOP_s1;
    else __fired_s1 = __t1_none;
    switch (__fired_s1) {
    case __tSTOP_s1:
      C->__SA1 = START;
      break;
    default: C->__SA1 = __preNext_SA1;
    }
  }
  switch (C->__SA1) {
  case START:
    __reset_CHRONO = true;
    if (C->_M_init3) {
      C->_M_init3 = false;
      d0 = 0;
    } else d0 = ((C->_M_pre_d00 + 1) % 100);
    if ((d0 < C->_M_pre_d00)) s = ((C->_M_pre_s1 + 1) % 60);
    else s = C->_M_pre_s1;
    if ((s < C->_M_pre_s1)) C->m = ((C->m + 1) % 60);
    C->_M_pre_d00 = d0;
    break;
  case STOP:
    __reset_CHRONO = false;
    if (C->_M_init2) {
      s = 0; C->_M_init2 = false; C->m = 0;
    } else s = C->_M_pre_s1;
    break;
  }
  if ((Rst & (not __reset_CHRONO))) C->__fired_w2 = __tCHRONO_w1;
  else C->__fired_w2 = __t2_none;
  switch (__SA3) {
  case LAP:
    if (Rst) __fired_w3 = __tLAP_w1;
    else __fired_w3 = __t3_none;
    switch (__fired_w3) {
    case __tLAP_w1:
      C->__Next_SA3 = TIME;
      break;
    default: C->__Next_SA3 = __SA3;
    }
    break;
  case TIME:
    if ((Rst & __reset_CHRONO)) __fired_w3 = __tTIME_w1;
    else __fired_w3 = __t3_none;
    switch (__fired_w3) {
    case __tTIME_w1:
      C->__Next_SA3 = LAP;
      break;
    default: C->__Next_SA3 = __SA3;
    }
    C->disp_1 = s; C->disp_2 = C->m;
  }
  C->_M_pre_s1 = s;
}
```

B. AUXILIARY FUNCTIONS

$$
\begin{array}{lcl}
Def(D_1 \,\mathsf{and}\, D_2) & = & Def(D_1) \cup Def(D_2) \\
Def(\mathtt{let}\, D_1 \,\mathtt{in}\, D_2) & = & Def(D_2) \\
Def(p = e) & = & fv(p) \\
Def(\mathtt{clock}\; x = e) & = & \{x\} \\
Def(\mathtt{reset}\; D \;\mathtt{every}\; e) & = & Def(D) \\
Def(\;\mathtt{match}\; e \;\mathtt{with}\;\;) & = & \cup_{1 \le i \le n} Def(D_i) \\
\quad\quad C_1 \rightarrow D_1 & & \\
\quad\quad \ldots & & \\
\quad\quad C_n \rightarrow D_n & & \\
Def(\;\mathtt{automaton}\quad\quad) & = & \cup_{1 \le i \le n} Def(u_i) \\
\quad\quad S_1 \rightarrow u_1\; s_1 & & \\
\quad\quad \ldots & & \\
\quad\quad S_n \rightarrow u_n\; s_n & & \\
Def(\mathtt{let}\, D \,\mathtt{in}\, u) & = & Def(u) \\
Def(\mathtt{do}\; D\; w) & = & Def(D)
\end{array}
$$

$$
\begin{array}{lcl}
fv(D_1 \,\mathsf{and}\, D_2) & = & fv(D_1) \cup fv(D_2) \\
fv(\mathtt{let}\, D_1 \,\mathtt{in}\, D_2) & = & fv(D_1) \cup (fv(D_2) - Def(D_1)) \\
fv(x = e) & = & fv(e) \\
fv(\mathtt{clock}\; x = e) & = & fv(e)
\end{array}
$$