Mixing Signals and Modes in Synchronous Data-flow Systems

Jean-Louis Colaço
Esterel-Technologies
France

Grégoire Hamon*
The MathWorks
USA

Marc Pouzet
LRI, Université Paris-Sud 11
France

ABSTRACT

Synchronous data-flow languages such as SCADE/LUSTRE manage infinite sequences, or *streams*, as primitive values making them naturally adapted to the description of data-dominated systems. Their conservative extension with means to define control-structures or *modes* has been a long-term research topic through which several solutions have emerged.

In this paper, we pursue this effort and generalize existing solutions by providing two constructs: a general form of state machines called *parameterized state machines*, and valued signals, as can be found in ESTEREL. Parameterized state machines greatly reduce the reliance on error-prone mechanisms such as shared memory in automaton-based programming. Signals provide a new way of programming with multi-rate data in synchronous data-flow languages. Together, they allow for a much more direct and natural programming of systems that combine data-flow and state-machines.

The proposed extension is fully implemented in the new 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; F.3.2 [**Semantics of programming languages**]: Operational semantics

General Terms

Design, Languages, Theory

Keywords

Synchronous languages. Mode automata. Compilation.

1. INTRODUCTION

The question of bringing together data-flow and state-machines — the two main paradigms that stand behind most programming languages for real-time embedded systems — has been explored by many research projects [16, 20, 8, 7, 6, 18, 10] or industrial tools.

Existing solutions rely on the collaboration between two independent languages or tools. In [10], we advocated the need for a more integrated solution inside a unique language when dealing with safety-critical applications. By providing a unified semantics, such an integration increases the modularity of designs and the quality of the generated code. Moreover both styles can be combined in any conceivable way. It is possible, for example, to define an automaton with equations attached to its states like *Mode-Automata* [18] or data-flow operators written as state machines as is traditionally done with the combination of SIMULINK and STATEFLOW [23] or SCADE and SYNCCHART [22].

In this paper, we continue the exploration of data-flow languages equipped with first class-automata. This combination provides opportunities for new programming constructs allowing for a simpler and safer description of systems. We propose two such constructs:

- *Parameterized state machines* – These provide a way to communicate values between states without using a shared memory mechanism. They are also lightweight for the user and enjoy the same efficient compilation technique as the one introduced in [10].
- *Valued signals* – These are events tagged with values as found in ESTEREL [5]. They provide a *positive* way of programming where the user only focuses on the instant where an event is produced, the event being implicitly absent otherwise. Though unusual in data-flow programming, it appears to combine well with the rest of the language. Moreover, these signals are *safe*: their content can only be accessed if the signal is present thus avoiding initialization issues.

These extensions are achieved by extending an existing synchronous data-flow language with a means to define state machines and signals. For that purpose, we define a *synchronous logical semantics* that accounts for both data-flow and imperative constructs. This semantics gives new insights on the operational behavior of automata and the way transitions are fired during a reaction.

The paper is organized as follows. Section 2 illustrates the use of parameterized state machines and signals in a synchronous data-flow system. Section 3 defines the synchronous semantics for a synchronous language extended with these constructions. This semantics is related to the *logical semantics* of ESTEREL, defining the execution of a

*This work was initiated while the author was at Chalmers University, Sweden.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

synchronous system as a sequence of atomic reactions. It is done in two steps, starting with a basic synchronous language and then adding an automaton construct and a means to manage signals. Section 5 discusses the proposed solution and compilation issues and we conclude in section 6

2. OVERVIEW

2.1 Parameterized State Machines

When programming with automata, it is often necessary to communicate values between two states upon taking a transition. A *setup* state needs to communicate initialization values to a *run* one for example.

Several mechanisms can be offered to establish such communications. This can be done, for example, through the use of shared imperative variables as it is done in Stateflow [19] where a global variable is defined, modified in the state that is exited, or on the transition itself, and read in the newly entered one. Such a mechanism is very light to use, however, it is not entirely satisfactory.

1. It breaks the locality of the communication, requiring a global variable to be declared, even though the communication only takes place between two states.

2. It breaks modularity, each state depending on this global variable to be in the current scope.

It is important to note that a shared variable mechanism allows much more general communication than the one we are considering: it can be used for example to communicate between entirely unrelated parts of a program, or through time. This is why a limited form of shared variables (precisely shared flows) have been introduced in [10] and is of natural use for describing modes. For the simpler case, which is also the one most needed, of communication between two states when transitioning from one to the other, we can offer an even simpler mechanism, with better properties, in the form of parameterized state machines. The exited state sends a value on the transition that is used to initialize or instantiate the target state.

An example of such communication is a system with several modes of normal execution, and a failure mode. The failure mode is entered from any other mode upon detection of an error. It is used to set back the system into a valid configuration before resuming execution. This failure mode typically needs to get some contextual information about the error and the measure that should be taken. Passing this information in shared memory would mean having global variables to hold it. These would need to receive meaningless values during normal execution and be set on the transition itself: the transition being taken right-away upon detection of an error. This not only breaks modularity and locality but is also very error-prone. Making sure that all such variables are always set correctly before being used is not a trivial task and it gets more and more complex as systems grow. Using parameterized state machines, the transition requires all the values to be provided, eliminating any risk of error: the program is correct by construction. Moreover, locality and modularity are preserved. Finally, as the communication is known to be instantaneous it does not use memory and is more efficiently compiled.

The example below written in Lucid Synchrone syntax [21] illustrates this typical usage. Our system is a three mode automaton, one of them being a failure state. Let `in1` and `in2` be integer input streams and `out` be an integer output stream of the system. `State1` is the entry state of the automaton. In the normal mode, the system computes a stream `out` using some auxiliary function `f`. This state is active until one of the two escaping conditions `(out > 10)` or `(in2 = 0)` is satisfied. `x` and `resume` are local variables to their state.

```
let node controller in1 in2 = out where
  automaton
  | State1 ->
     do out = f (in1, in2)
     until (out > 10) then State2
     until (in2 = 0) then Fail_safe(1, 0)
  | State2 ->
     let rec x = 0 -> (pre x) + 1 in
     do out = g (in1,x)
     until (out > 1000) then Fail_safe(2, x)
  | Fail_safe(error_code, resume_after) ->
     let rec
       resume = resume_after -> (pre resume) - 1 in
     do out = if (error_code = 1) then 0
              else 1000
     until (resume <= 0) then State2
  end
```

Upon failure, two values are passed to the `Fail_safe` state. One is an error code, the second one is a time that the system should stay in the failure mode until operation can safely resume. The communication is kept completely local, and is explicitly needed when a failure is detected.

This mechanism is reminiscent of the use of continuation passing style in place of gotos that is largely used in general purpose languages and particularly functional ones [2]. The motivations behind its introduction are basically the same, as positioning shared variables before transitioning to another state is similar to positioning shared variables before a goto instruction and is as error-prone [12]. Alternative approaches could have been considered. In [9, 15], we used the functional nature of the language and properties of tail-recursion to define reconfiguration mechanisms. Offering a dedicated mechanism for communication between states gives a much more lightweight solution and allows for more efficient implementation. In particular, these automata are a generalization of the basic automata proposed in [10] and the same analysis and compilation techniques can be applied.

Parameterized states are complementary to global shared variables and are not intended to replace then. They provide a way to keep the communication between two states local without interfering with the rest of the automaton. Moreover, it does not raise initialization issues as global variables do. Indeed, in imperative formalisms (e.g., Stateflow, Esterel V5 and V7), a global variable (or signal in Esterel) is implemented as a memory whose default behavior is to keep its previous value. As a consequence, the compiler has to check that the variables has been initialized before behing defined which call for sophisticated program analysis. Our motivation is thus to offer a simple mechanism with better properties and for which the good initialization of program in particular can be easily checked.

2.2 Signals and Signal Patterns

We now consider the addition of signals. Adding signals to a synchronous data-flow language seems like a strange idea as a stream is itself the representation of a signal: a data which might take different values in time and which can be either present or absent. So, more precisely, we use the term signal to refer to ESTEREL signals, that is, valued events. Our interest in signals is that they lead to a different way of programming, not directly available in data-flow programming. Signal-based programming is *positive*: the programmer only specifies things that happen and their effect, signal being implicitly absent otherwise. In data-flow on the other hand, the programmer needs to describe the effect of things happening, as well as not happening, which can be cumbersome when looking at control dominated system like automata.

A signal can only be emitted or tested for presence. When it is present its value can be accessed. We can for example write a function that adds the value of two integer signals if they are both present, it returns the value of the one present if only one is present and nothing otherwise:

```
let node sum x y = o where
  present
  | x(v) & y(w) -> do emit o = v + w done
  | x(v1) -> do emit o = v1 done
  | y(v2) -> do emit o = v2 done
  | _ -> do done
  end
```

The construction `x(v) & y(w)` is a synchronization pattern which is verified when both signals `x` and `y` are present. In this case, the parameters `v` and `w` are bound to the values of `x` and `y` and the correspond equation `emit o = v + w` is executed. Such a construction is new in a synchronous data-flow language and is reminiscent of pattern-matching in functional languages or join-patterns in the Join-calculus [13]. It matches patterns of events. The same function is easy to write in a purely data-flow style but is a lot less direct than this one. It requires completing the streams with the right values to get the result we want when adding them in a single operation.

Accessing signals through patterns allows accesses to be entirely safe: the value of a signal can only be read if the signal is present. This is a key difference with ESTEREL, for example, where the value of a signal can be read at any time, leading to initialization issues.

2.3 State Machines and Signals

These two features are orthogonal and address different issues. They both improve the definition of control-dominated systems in a data-flow language. They compose well and together lead to a more direct and concise programming of systems. The example below makes use of both features. It implements a mouse controller. An informal specification of this system is:

> "Return the event double when two click has been received in less than four tops. Emit single if only one click has been received".

This example can be programmed as an automaton receiving two signals `click` and `top` and emitting a signal `o` that can take two possible values from an enumerated type: `Single` and `Double`. The expression `? e` returns the value true when `e` is present and false otherwise.

```
type output = Single | Double

let node count e = cpt where
  rec cpt = if ? e then 1 -> pre cpt + 1
            else 0 -> pre cpt

let node controler click top = o where
  automaton
  | Await -> do until click(_) then One
  | One ->
     do unless click(_) then Detected(Double)
     unless (count top = 4) then Detected(Single)
  | Detected(x) ->
     do emit o = x
     until true then Await
  end
```

The automaton awaits for the first emission of the signal `click` (pattern `_` means that its carried value is ignored) then it enters in state `One`. In this state, at the instant where `click` is received, it immediately enters the parameterized state `Detected` giving the initial value `Double`. Otherwise and at the fourth occurrence of `top`, it enters in state `Detected` with initial value `Single`. In the target state, it emits the signal `o` then goes in state `Await` for the remaining execution.

Many other examples illustrating these two constructs and their combination are available in the distribution of the LUCID SYNCHRONE compiler.

3. FORMALIZATION

3.1 A Core Data-flow Language

We define a first-order synchronous data-flow kernel considered as a basic calculus. We equip it with two control structures, a means to reset the computation of a set of equations, and a pattern-matching construct over a user defined data-type. Their translation into regular synchronous code (e.g., LUSTRE) is given in [15, 10].

A program is made of a sequence of global value declarations (d) and type declarations. A global value declaration defines a node ($\texttt{let node}\ x(p) = e$). Expressions ($e$) are made of constructed values ($C^n(e_1, ..., e_n)$) built with a constructor C with arity n, an initialization operator ($e_1\ \texttt{->}_k\ e_2$), a delay initialized with a value ($\texttt{pre}_v(e)$), a local definition ($e\ \texttt{where}\ D$), variables (x), a point-wise application ($op(e)$) or a node application ($f(e)$).

A pattern p may be a value constructor (C^n) with arity n applied to n patterns ($C^n(p_1, ..., p_n)$), a variable (x) or a wildcard ($\_$). A declaration (D) can declare a value ($p = e$), a collection of parallel equations ($D\ \texttt{and}\ D$), it may introduce local names ($\texttt{let}\ D_1\ \texttt{in}\ D_2$) or it can be a control-structure. A control-structure can be a pattern-matching construction ($\texttt{match}\ e\ \texttt{with}\ p\ \texttt{->}\ D\ ...\ p\ \texttt{->}\ D$) or a definition reset on a boolean condition ($\texttt{reset}\ D\ \texttt{every}\ e$). Value constructors ($C^n$) are tagged with their arity. A sum type $\texttt{type}\ t = \{C_1^{n_1} : t_{11} \times ... \times t_{1n_1}; ...; C_m^{n_m} : t_{m1} \times ... \times t_{mn_m}\}$ is well formed if all the C_i are distinct from each other.

$$
\begin{array}{rcl}
e & ::= & C^n(e, ..., e) \mid x \mid e_1\ \texttt{->}_k\ e_2 \mid \texttt{pre}_v(e) \\
& & \mid e\ \texttt{where}\ D \mid op(e) \mid f(e)
\end{array}
$$

$$
\begin{array}{rcl}
D & ::= & D \texttt{ and } D \mid p = e \mid \texttt{let } D \texttt{ in } D \\
& & \mid \texttt{match } e \texttt{ with } p \texttt{ -> } D \ldots p \texttt{ -> } D \\
& & \mid \texttt{reset } D \texttt{ every } e
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & C^n(p, ..., p) \mid x \mid \_ \\
op & ::= & \texttt{+} \mid ... \\
k & ::= & 0 \mid 1 \\
d & ::= & \texttt{let node } f(p) = e \mid d; d \\
td & ::= & \texttt{type } t \mid td; td \\
& & \texttt{type } t = \{C^n : t \times ... \times t; ...; C^n : t \times ... \times t\}
\end{array}
$$

We note $fv(e)$ the set of free variables from an expression e. $Def(D)$ stands for the defined names from D. Their definition is given in appendix 6.

We also suppose that the last handler of `match/with` statement is of the form `_ -> D` in order to guarantee that the pattern matching is exhaustive[1].

e_1 `->`$_k$ e_2 stands for the initialization operator and $\texttt{pre}_v(e)$ for the initialized delay. The initialization operator is a simple two state machine which returns the current value of its first input and then the current value of its second input. The extra argument k where $k \in \{0, 1\}$ indicates if the automaton is in its initial state or not ($k = 0$ standing for the initial state). In conventional Lustre or Lucid Synchrone syntax we simply write e_1`->`e_2 as a short-cut for e_1 `->`$_0$ e_2. In the same way, `pre`(e) stands for the previous value of e and is a short-cut for $\texttt{pre}_{nil}(e)$ where nil denotes any value with the type of e. $op(e)$ stands for the point-wise application of a combinatorial function (typically an external function such as `+`, `not`). A pair is simply written (x, y) instead of ,(x, y). If x and y are two streams, $+(x, y)$ stands for the point-wise addition that we write $x + y$.

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` (y)	nil	y_0	y_1	y_2	...
x `fby` y	x_0	y_0	y_1	y_2	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	...
(x, y)	(x_0, y_0)	(x_1, y_1)	(x_2, y_2)	(x_3, y_3)	...

The language provides an ML-like pattern-matching construct. Such a construction allows to combine stream equations. Keeping the notation of the presented kernel, the conditional `if/then/else` can be encoded as:

```
let node ifthenelse (c, x, y) = o where
    match c with
    | true -> o = x
    | false -> o = y
```

In the rest of the paper we simply write t for `true` and f for `false`.

3.2 Synchronous Semantics

The synchronous semantics is given by reaction rules, following the logical semantics of Esterel [4]. We define the set of instantaneous values (v) that can be produced during a reaction. For simplicity of the presentation, values are structured with constructors. The language is first-order.

[1]This is stronger than necessary and ML languages have developed exhaustivity analysis [17].

$$
\begin{array}{lrcll}
\text{Values:} & v & ::= & C^n(v, ..., v) \mid C^0 & \\
\text{Environments:} & R & ::= & [v_1/x_1, ..., v_n/x_n] & \textit{(locals)} \\
& I & ::= & R & \textit{(inputs)} \\
& O & ::= & R & \textit{(outputs)}
\end{array}
$$

$$
\text{Reaction:} \quad R \overset{k}{\vdash} e_1 \xrightarrow{v} e_2 \quad k \in \{0, 1\}
$$

$$
R \overset{k}{\vdash} D \xrightarrow{R'} D' \text{ with } R' \subseteq R \quad k \in \{0, 1\} \text{ and } Def(D) = Dom(R')
$$

$Dom(R)$ denotes the domain of R. If $x \in Dom(R)$ then $R(x)$ denotes the value associated to x in R. $R_1 + R_2$ denotes the concatenation of R_1 and R_2, provided there is no name conflict, that is, $Dom(R_2) \cap Dom(R_1) = \emptyset$. In other words, a variable defined in R_2 must not be already defined in R_1. A substitution $\rho = [v_1/x_1, ..., v_n/x_n]$ is lifted to patterns such that $\rho(C^n(p_1, ..., p_n)) = C^n(\rho(p_1), ..., \rho(p_n))$. A pattern p filters a value v noted $p \rhd v$ if there exists a substitution ρ such that $\rho(p) = v$ and we write $[v/p]$ for it. A wildcard _ matches any pattern and $[v/\_]$ produces the empty substitution. k stands for a reset bit with values from $\{0, 1\}$ interpreted as booleans.

The predicate $R \overset{k}{\vdash} e_1 \xrightarrow{v} e_2$ means that the expression e_1 under the environment R emits the value v and rewrites into the expression e_2. In doing this, we introduce a special reset flag. When $k = 0$, the reaction is reset and starts in its initial state. When $k = 1$, it is not. The predicate $R \overset{k}{\vdash} D \xrightarrow{R'} D'$ means that the declaration D defines the instantaneous environment R' and rewrites into the declaration D'. To simplify the presentation, we impose that every name defined in D be produced during the reaction. A more general solution is proposed in [10] where values which are not produced keep their previous value. We shall come back to this point in section 5. The complete execution of a declaration D_1, under a sequence of input environments $H_I = I_1.I_2...$ produces a sequence of environments $H_O = O_1.O_2....$

$$
I_i + O_i + R_i \overset{1}{\vdash} D_i \xrightarrow{O_i + R_i} D_{i+1}
$$

During a synchronous reaction, computations may observe both input, local or output signals emitted during the reaction. This is why the reaction is computed in an extended environment $I_i + O_i + R_i$. An execution is made in a global static environment S made of the node definitions of a program. A program defining nodes $f_1, ..., f_n$ produces an initial environment of the following form:

$$
S = [\lambda p_1.e_1/f_1, ..., \lambda p_n.e_n/f_n]
$$

Being global, we keep it implicit in the rule and shall simply write $S(f)$ for the node definition associated to the identifier f.

Reduction rules are defined in figure 1. We consider them modulo renaming (α-conversion). Functions imported from the host language apply point-wise to their arguments (rule (app)). A node application evaluates by first expanding its definition (rule (APP)). The notation $C^n(e_1, ..., e_n)$ stands for the point-wise application of a n-ary constructor C to a n-tuple of instantaneous values. The delay shifts its input argument (rule (PRE)). The initialization operator ($\rightarrow$) acts as a two state machine. In the initial step ($k = 0$), it returns

$$\text{(C)}\ \frac{R \overset{k}{\vdash} e_1 \xrightarrow{v_1} e_1' \quad \cdots \quad R \overset{k}{\vdash} e_n \xrightarrow{v_n} e_n'}{R \overset{k}{\vdash} C^n(e_1,...,e_n) \xrightarrow{C(v_1,...,v_n)} C^n(e_1',...,e_n')} \qquad \text{(app)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{v} e' \quad v' = op(v)}{R \overset{k}{\vdash} op(e) \xrightarrow{v'} op(e')} \qquad \text{(PRE)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{v'} e'}{R \overset{k}{\vdash} \mathtt{pre}_v(e) \xrightarrow{v} \mathtt{pre}_{v'}(e')}$$

$$\rightarrow_k \frac{R \overset{0}{\vdash} e_1 \xrightarrow{v_1} e_1' \quad R \overset{0}{\vdash} e_2 \xrightarrow{v_2} e_2'}{R \overset{0}{\vdash} e_1 \texttt{ ->}_k e_2 \xrightarrow{v_1} e_1' \texttt{ ->}_1 e_2'} \qquad \rightarrow_{10} \frac{R \overset{1}{\vdash} e_1 \xrightarrow{v_1} e_1' \quad R \overset{1}{\vdash} e_2 \xrightarrow{v_2} e_2'}{R \overset{1}{\vdash} e_1 \texttt{ ->}_0 e_2 \xrightarrow{v_1} e_1' \texttt{ ->}_1 e_2'} \qquad \rightarrow_{11} \frac{R \overset{1}{\vdash} e_1 \xrightarrow{v_1} e_1' \quad R \overset{1}{\vdash} e_2 \xrightarrow{v_2} e_2'}{R \overset{1}{\vdash} e_1 \texttt{ ->}_1 e_2 \xrightarrow{v_2} e_1' \texttt{ ->}_1 e_2'}$$

$$\text{(TAUT)}\ \frac{R(x) = v}{R \overset{k}{\vdash} x \xrightarrow{v} x} \qquad \text{(WHERE)}\ \frac{R + R' \overset{k}{\vdash} D \xrightarrow{R'} D' \quad R + R' \overset{k}{\vdash} e \xrightarrow{v} e'}{R \overset{k}{\vdash} e \texttt{ where } D \xrightarrow{v} e' \texttt{ where } D'}$$

$$\text{(APP)}\ \frac{S(f) = \lambda p.e \quad R \overset{k}{\vdash} e \texttt{ where } p = e_1 \xrightarrow{v} e'}{R \overset{k}{\vdash} f(e_1) \xrightarrow{v} e'} \qquad \text{(DEF)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{v} e'}{R \overset{k}{\vdash} p = e \xrightarrow{[v/p]} p = e'}$$

$$\text{(AND)}\ \frac{R \overset{k}{\vdash} D_1 \xrightarrow{R_1} D_1' \quad R \overset{k}{\vdash} D_2 \xrightarrow{R_2} D_2'}{R \overset{k}{\vdash} D_1 \texttt{ and } D_2 \xrightarrow{R_1+R_2} D_1' \texttt{ and } D_2'} \qquad \text{(LET)}\ \frac{R + R_1 \overset{k}{\vdash} D_1 \xrightarrow{R_1} D_1' \quad R + R_1 \overset{k}{\vdash} D_2 \xrightarrow{R_2} D_2'}{R \overset{k}{\vdash} \texttt{let } D_1 \texttt{ in } D_2 \xrightarrow{R_2} \texttt{let } D_1 \texttt{ in } D_2}$$

$$\text{(RESET-t)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{t} e' \quad R \overset{0}{\vdash} D \xrightarrow{R'} D'}{R \overset{k}{\vdash} \texttt{reset } D \texttt{ every } e \xrightarrow{R'} \texttt{reset } D' \texttt{ every } e'} \qquad \text{(RESET-f)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{f} e' \quad R \overset{k}{\vdash} D \xrightarrow{R'} D'}{R \overset{k}{\vdash} \texttt{reset } D \texttt{ every } e \xrightarrow{R'} \texttt{reset } D' \texttt{ every } e'}$$

$$\text{(MATCH)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{v} e' \quad \forall j \in \{1,...,i-1\},\ p_j \not\triangleright v \quad R + [v/p_i] \overset{k}{\vdash} D_i \xrightarrow{R'} D_i'}{R \overset{k}{\vdash} \texttt{match } e \texttt{ with } p_1 \texttt{ -> } D_1 \texttt{ ... } p_n \texttt{ -> } D_n \xrightarrow{R'} \texttt{match } e \texttt{ with } p_1 \texttt{ -> } D_1 \texttt{ ... } p_i \texttt{ -> } D_i' \texttt{ ... } p_n \texttt{ -> } D_n}$$

Figure 1: Synchronous Semantics for the Basic Calculus

the value of its first input (rule $(\rightarrow_k)$). In the remaining steps ($k = 1$), it returns the value of its first argument if the automaton is in its initial state and it returns the value of its second argument otherwise (rules $(\rightarrow_{10})$ and $(\rightarrow_{11})$). Indeed, an initialization x `->` y returns the first value of x when the reset condition is true or because it is in a block which has not been already executed. In a reaction, the current value of x is the one present in the hypothesis (rule (TAUT)). The rule (WHERE) is for local definitions in expressions. An equation produces a substitution (rule (DEF)). Rules (AND) and (LET) are for parallel and local definitions. A reset definition `reset` D `every` e reacts as D does except that the body is re-initialized every time e is true or the context k imposes it. When the reset flag is true, every initialization operator from D restarts with its initial value. A pattern matching construct first evaluates the expression e then selects the first branch whose pattern matches the value of e. Note that because of the constraint on reaction, a shared variable x must have a definition in every handler of a pattern matching.

The synchronous semantics both gives meaning to the purely data-flow part as well as the `reset` construction which is more imperative. This is simply taken into account by adding an extra tag to the semantics which leads to a simpler formulation than existing ones [9, 15] where the reset was expressed as a form of tail-recursion. Moreover, it follows more closely the actual implementation in the ReLuC and Lucid Synchrone compilers where the reset bit is tested at the beginning of the reaction to decide whether the memory must be re-initialized or not.

4. ADDING MODES AND SIGNALS

In this section, we introduce a programming language which extends the basic one with new control structures and a means to manage signals.

In Esterel, signals can be either *pure* or *valued*. A pure signal is nothing but a boolean flow. A valued signal is a more complex object. It is represented as a pair (p, v) made of a value v and a presence bit (the *enable* in circuit terminology or the *clock* in synchronous data-flow programming). We provide two constructions to manage those valued signals. A signal x can be emitted with some value by writing `emit` $x = e$ whereas the presence can be tested and its carried value can be accessed through a pattern-matching construct `present`. We extend the language of equations in the following way. An equation can define the current value of a signal (`emit` $x = e$). Two extra control-structures are added. A present construct is of the form (`present` $si \rightarrow D$ ... $si \rightarrow D$) and an automaton construct ($\texttt{automaton}_k^{vs}$ $sp \rightarrow u^{k'}$ `unless` s^k ...). In a present construct, si stands for a signal pattern. A signal pattern may be a synchronization pattern (si_1 & si_2), an atomic signal testing $x\langle p\rangle$ stating that x must be present and match the pattern p, or it can be a boolean expression e. An automaton is annotated with the current entry state of the automaton vs and an initialization status $k \in \{0, 1\}$ indicating whether the entry state must reset or not on entry. Moreover, each handler of the form $sp_i \rightarrow u_i^{k_i'}$ `unless` $s_i^{k_i}$ is annotated with two bits k_i' and k_i which whether the set of equations u (with its set of weak conditions) or the strong conditions s must be reset or not. In concrete syntax, we

simply write `automaton` $sp_1 \to u_1$ `unless` s_1 ... as a shortcut for ($\texttt{automaton}_0^{sp_1}$ $sp_1 \to u_1^0$ `unless` s_1^0 ...) meaning that the first state in the list is the initial state and that every computation has to be reset on entry. In an automaton, every handler ($sp \to u^{k'}$ `unless` s^k) is made of two parts. (u) defines a set of shared variables (`do` D `until` s) which may use some auxiliary local names (`let` D `in` u) and a set of escaping conditions (s) to escape *weakly* from the handler. The second part ($u^{k'}$ `unless` s^k) defines the *strong* conditions to leave the current state. The keyword `until` means that the condition are tested at the end of the reaction. The transition is *weak* (to keep ESTEREL terminology) whereas the keyword `unless` means that the condition is tested before the definitions from u are executed. A state pattern (sp) has the form $S(p_1, ..., p_n)$ and a state expression has the form $S(e_1, ..., e_n)$. State patterns are supposed to be pairwise different and the initial state has no parameter (it is of the form S).

$$
\begin{array}{lcl}
D & ::= & D \texttt{ and } D \mid p = e \mid \texttt{let } D \texttt{ in } D \\
& & \mid \texttt{emit } x = e \\
& & \mid \texttt{match } e \texttt{ with } p \texttt{ -> } D \ldots p \texttt{ -> } D \\
& & \mid \texttt{reset } D \texttt{ every } e \\
& & \mid \texttt{present } si \texttt{ -> } D \ldots si \texttt{ -> } D \\
& & \mid \texttt{automaton}_k^{vs} \\
& & \quad sp \to u^k \texttt{ unless } s^k \\
& & \quad \ldots \\
& & \quad sp \to u^k \texttt{ unless } s^k \\
u & ::= & \texttt{let } D \texttt{ in } u \mid \texttt{do } D \texttt{ until } s \\
s & ::= & si \texttt{ then } se\ s \mid si \texttt{ continue } se\ s \mid \epsilon \\
si & ::= & si \,\&\, si \mid x\langle p \rangle \mid e \mid \_ \\
se & ::= & S \mid S(e, ..., e) \\
sp & ::= & S \mid S(p, ..., p)
\end{array}
$$

We define the set $Emit(D)$ of a set of definitions as the set of emitted names from D. Its definition is given in the appendix 6.

As we did for the pattern-matching construct, we suppose that the last handler from a `present` statement is of the form $\_ \to D$.

4.1 Synchronous Semantics

We now extend the synchronous semantics, generalizing the idea introduced in the basic one with the use of the k tag. An automaton is annotated with two informations, the current active state and an initialization tag k stating whether the current state must be reset on entry or not. The reaction of an automaton thus computes a set of equations and defines how these these two tags change.

A signal can be either present or absent and we extend the set of instantaneous values which can be emitted during a reaction. A state value is simply the name of a state parameterized by some values. We have:

$$
\begin{array}{lllll}
\text{Values:} & v' & ::= & v \mid abs & \\
\text{State values:} & vs & ::= & S \mid S(v, ..., v) & \\
\text{Environments:} & R & ::= & [v'_1/x_1, ..., v'_n/x_n] & \textit{(locals)} \\
& I & ::= & R & \textit{(inputs)} \\
& O & ::= & R & \textit{(outputs)} \\
\text{Reaction:} & R \overset{k}{\vdash} e_1 \xrightarrow{v} e_2 & & k \in \{0,1\} & \\
& R \overset{k}{\vdash} D \xrightarrow{R'} D' & & k \in \{0,1\} &
\end{array}
$$

Moreover, in order to be applied, a synchronous reaction $R \overset{k}{\vdash} D \xrightarrow{R'} D'$ must verify the following properties:

1. $x \in Def(D) \Rightarrow [v/x] \in R \wedge [v/x] \in R'$
2. $x \in Emit(D) \wedge [v/x] \in R' \Rightarrow [v/x] \in R$
3. $x \in Emit(D) \wedge [v/x] \notin R' \Rightarrow [abs/x] \in R$

The first property was already imposed on the basic language. It states that every stream variable defined in a block D must be produced by the reaction. The second property states that if the signal x is emitted with value v then it also has the value v in the hypothesis R. Finally, if a signal x is defined in a definition D but it is not emitted then it is marked absent in the hypothesis.

The emission of a signal x is done by the equation `emit` $x = e$ with the corresponding reaction rule:

$$
\text{(EMIT)}\ \frac{R \overset{k}{\vdash} e \xrightarrow{v} e'}{R \overset{k}{\vdash} \texttt{emit } x = e \xrightarrow{[v/x]} \texttt{emit } x = e'}
$$

The extended language provides a pattern matching mechanism over signals. Signals are special values which can be either present (v) or absent (abs). We define the predicate $R \overset{k}{\vdash} si \underset{v}{\overset{R'}{\to}} si'$ stating that the signal pattern si may define the local environment R' and rewrites to si'. A signal testing succeeds when $v = t$ and it fails when $v = f$. The definition of this predicate is given in figure 2. An atomic signal testing $x\langle p \rangle$ fails if x is absent (rule (TEST-abs)) or the value is not matched (rule (TEST-f)). It succeeds if x is present and its value is matched by the pattern p (rule (TEST-t)) or it is a wildcard pattern (rule (WILD)). A synchronization pattern $si_1 \,\&\, si_2$ succeeds if the two succeed and it produces the union of the two environments. A boolean test succeeds if the expression evaluates to the true value (rule (EXP)). For a `present` statement, all the signal patterns are evaluated and the selected branch is the first one in the list of handlers (rule (PRESENT)).

For example, the signal pattern $x\langle 0 \rangle \,\&\, y\langle z \rangle \,\&\, (cpt = 0)$ succeeds when the signal x is present and carries the value 0, the signal y is present and the boolean condition $cpt = 0$ is true. When this synchronization pattern occurs, it binds the variable z to the actual value carried by the signal y. In a `present` statement, synchronization patterns are tested sequentially and the first pattern to succeed determines the selected branch.

The semantics of state machines is necessarily a little more subtle because every state defines a set of equation and can be escaped with a variety of transitions. Transitions may be *strong* or *weak* and this depend on the instant where the condition is tested. Moreover, for each transition, we must decide if the target state is reset on entry or not. For this semantics, we introduce two auxiliary predicates, one for the semantics of escaping conditions and one for the execution of the body of a state. The predicate $R \underset{vs}{\overset{k}{\vdash}} s \underset{k'}{\overset{vs'}{\to}} s'$ states that in the environment R, a reset context k and the current state value vs, the escape condition s produces the new state value vs', the reset condition k' and rewrites to s'. The predicate $R \underset{vs}{\overset{k}{\vdash}} u \underset{k',vs'}{\overset{R'}{\to}} u'$ states that under the environment R, the reset condition k and the current state vs,

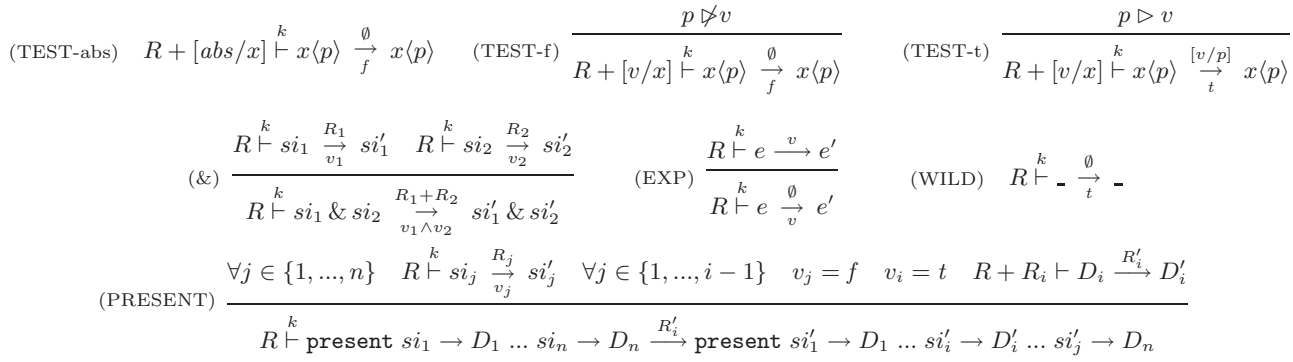

Figure 2: Signal Matching and Presence Testing

the body u produces the environment R', a reset condition k', a state value vs' and rewrites to u'. k' states that the target state is reset on entry.

The semantics for automata is given in figure 3. Two cases may occur according to the reset context k (rules (AUT_0) and (AUT_1)). When $k = 0$, this means that the current state is the initial state sp_1 of the automaton and every state is thus reset. The automaton first evaluates the strong condition s_1 which produces the state value v and reset condition k' on entry on that state. We select the state pattern that match v and, *during the same reaction*, we execute the corresponding body u_i. This execution produces several results: a reaction environment R', a reset condition k_0' for the next reaction and a target state value vs_0'. The second rule (TAUT_1) is for the remaining executions of an automaton. The current active state (which matches the stored value vs_0) is selected and the reset condition k_0 is the one that has been returned by the previous reaction. The strong conditions are first executed to determine what is the current active state sp_i. Then, it is executed during the same reaction and produces the environment and information for the next reaction.

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 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. We shall come back to this point in section 5.

Rules (LET) and (DO) define how the body of an automaton reacts. In the first rule, the local definition is first evaluated. Then, in the extended environment, u is evaluated. The second rule exhibits the main difference between strong and weak conditions. Here, the set of equations D in `do` D `until` s is executed no matter are the values of the weak conditions s. These conditions will determine the active state and reset status for the next reaction only, not the current one.

The five rules defining the semantics of escape conditions are given in figure 4. They are very similar to each other. The keyword `then` means that the target state is reset whereas the keyword `continue` means that it is not. Moreover, conditions are evaluated in sequence. If no condition succeed, then the current state vs is returned and is not reset (rule (Epsilon)).

5. DISCUSSION

In this section, we explain how parameterized automata and signals are compiled and we discuss the design choices we have made.

5.1 Translation Semantics

In [10] we have introduced a *translation semantics* for a synchronous data-flow language extended with finite state machines. The semantics was neither operational nor denotational in the usual sense: the semantics of automata was given by their translation into a basic language, thus giving at the same time a compilation method. Though satisfactory, a more direct semantics was lacking and the present paper answers to that need.

The compilation of parameterized state machines and signals does not raise technical difficulties and the translation method introduced previously has been easily generalized. We illustrate it informally on the following two states automaton. This automaton awaits for the presence of both signals a and b. When they are present, it enters in the state `Emit` parameterized by the sum of the values of a and b. This sum is constantly emitted as soon as the boolean condition r is true (it is of course possible to write a form where o is emitted only on entry). We write it in the abstract syntax given in section 3 (see [21] for its writing in concrete syntax).

```
let node sum (a, b, r) = o where
    automaton
     | Await -> do unless a⟨x⟩&b⟨y⟩ then Emit (x + y)
     | Emit (v) -> do emit o = v unless r then Await
```

A signal can be represented by a constructed value. An absent value is represented by the constructor `Abs` whereas a present value with value x is represented by `P`(x). An alternative concrete representation is to use pairs instead (this is what is done in the Lucid Synchrone compiler). Following the method introduced in [10], an automaton is translated into the core data-flow language, and generates two pattern-matching constructs. For this purpose, we introduce two extra variables. The variable *pnextstate* defines the target state produced by the previous reaction whereas the variable *state* defines the current active state. The first pattern-matching computes the current active state according to strong transitions whereas the second one defines what is computed (here the value of o and the weak transitions) and what will be the active state for the next reaction. In

$$\text{(AUT}_0\text{)}\quad \frac{R \overset{0}{\underset{sp_1}{\vdash}} s_1 \overset{vs}{\underset{k'}{\rightarrow}} s'_1 \qquad sp_i \rhd vs \qquad R + [vs/sp_i] \overset{0}{\underset{vs}{\vdash}} u_i \overset{R'}{\underset{k'_0, vs'_0}{\rightarrow}} u'_i}{\begin{array}{l} R \overset{0}{\vdash} \texttt{automaton}^{vs_0}_{k_0} \\ \qquad sp_1 \rightarrow u_1^{k'_1} \texttt{ unless } s_1^{k_1} \\ \qquad \ldots \\ \qquad sp_i \rightarrow u_i^{k'_i} \texttt{ unless } s_i^{k_i} \\ \qquad \ldots \\ \qquad sp_n \rightarrow u_n^{k'_n} \texttt{ unless } s_n^{k_n} \end{array} \xrightarrow{R'} \begin{array}{l} \texttt{automaton}^{vs'_0}_{k'_0} \\ \qquad \ldots \\ \qquad sp_1 \rightarrow u_1^{0} \texttt{ unless } {s'_1}^{1} \\ \qquad \ldots \\ \qquad sp_i \rightarrow {u'_i}^{1} \texttt{ unless } s_i^{k'} \\ \qquad \ldots \\ \qquad sp_n \rightarrow u_n^{0} \texttt{ unless } s_n^{0} \end{array}}$$

$$\text{(AUT}_1\text{)}\quad \frac{sp_j \rhd vs_0 \qquad R \overset{k_0 \wedge k_j}{\underset{vs_0}{\vdash}} s_j \overset{vs}{\underset{k'}{\rightarrow}} s'_j \qquad sp_i \rhd vs \qquad R + [vs/sp_i] \overset{k' \wedge k'_i}{\underset{vs}{\vdash}} u_i \overset{R'}{\underset{k'_0, vs'_0}{\rightarrow}} u'_i}{\begin{array}{l} R \overset{1}{\vdash} \texttt{automaton}^{vs_0}_{k_0} \\ \qquad sp_1 \rightarrow u_1^{k'_1} \texttt{ unless } s_1^{k_1} \\ \qquad \ldots \\ \qquad sp_i \rightarrow u_i^{k'_i} \texttt{ unless } s_i^{k_i} \\ \qquad \ldots \\ \qquad sp_n \rightarrow u_n^{k'_n} \texttt{ unless } s_n^{k_n} \end{array} \xrightarrow{R'} \begin{array}{l} \texttt{automaton}^{vs'_0}_{k'_0} \\ \qquad \ldots \\ \qquad sp_j \rightarrow u_j^{k_0 \wedge k_j} \texttt{ unless } {s'_j}^{1} \\ \qquad \ldots \\ \qquad sp_i \rightarrow {u'_i}^{1} \texttt{ unless } s_i^{k'} \\ \qquad \ldots \\ \qquad sp_n \rightarrow u_n^{k'_n} \texttt{ unless } s_n^{k_n} \end{array}}$$

$$\text{(LET)}\ \frac{R + R' \overset{k}{\vdash} D \xrightarrow{R'} D' \qquad R + R' \overset{k}{\underset{vs}{\vdash}} u \overset{R''}{\underset{k', vs'}{\rightarrow}} u'}{R \overset{k}{\underset{vs}{\vdash}} \texttt{let } D \texttt{ in } u \overset{R''}{\underset{k', vs'}{\rightarrow}} \texttt{let } D' \texttt{ in } u'} \qquad \text{(DO)}\ \frac{R \overset{k}{\vdash} D \xrightarrow{R'} D' \qquad R \overset{k}{\underset{vs}{\vdash}} s \overset{vs'}{\underset{k'}{\rightarrow}} s'}{R \overset{k}{\underset{vs}{\vdash}} \texttt{do } D \texttt{ until } s \overset{R'}{\underset{k', vs'}{\rightarrow}} \texttt{do } D' \texttt{ until } s'}$$

Figure 3: The Synchronous Semantics for Automata

the abstract syntax of the basic language, we get the following program.

```
let node sum (a, b, r) = o where
    match pnextstate with
     | Await -> match (a, b) with
                | (P(x), P(y) -> state = Emit(x + y)
                | _ -> state = Await
     | Emit(v) -> match r with
                  | true -> state = Await
                  | false -> state = Emit(v)
    and
    match state with
     | Await -> o = Abs and nextstate = Await
     | Emit(v) -> o = P(v) and nextstate = Emit(v)
    and
    pnextstate = Await -> pre nextstate
```

This program can in turn be compiled using the existing code generation tools of synchronous languages. Such a two step compilation strategy, where the extended language is first translated into its data-flow subset before code generation is applied, has several useful properties in the context of critical systems. It allows existing qualified code generation to be used (such as the one of SCADE), it increases tracability and produces efficient code. This is why the solution have been retained for the next SCADE compiler.

5.2 Transient States

The automata construction proposed here is rather rich in terms of transitions. A transition can be *strong* or *weak* and can *continue* or *reset* the target state. The semantics of strong transitions consists in inspecting their guards at the very beginning of the cycle in order to determine what is the state that must be evaluated in the current cycle. Conversely, weak transitions are evaluated at the end of the current active state and determine the next state. While a little subtle to use, these two types of transitions are necessary in many applications. The semantics is quite clear locally and in an homogeneous automaton (with only weak or only strong transitions). What we have considered here is an arbitrary mixture of both, which needs more design choices to decide what to do when a state is entered with a transition of a different kind that its leaving one. Thus, two possible situations may occur during a synchronous reaction:

1. entering with a *strong* and leaving with a *weak*, or
2. entering with a *weak* and leaving with a *strong*.

Note that this is already much simpler than what is offered in other automata based formalisms like SYNCCHART [1] or the STATEFLOW where an arbitrary number of states can be crossed during a reaction. Nonetheless, one can wonder if this must be more restricted. It is clear that only one state is active per cycle. We might want something similar for transitions and may impose that only one transition be fired during a cycle. To respect this condition, the rule for weak transitions must keep track on the fact that a strong one has been fired at the beginning of the cycle. This choice answers situation 1. For the second one, we can either allow to go through a state without activating it (by composing a weak transition followed by a strong one) or avoid that when testing strong transitions.

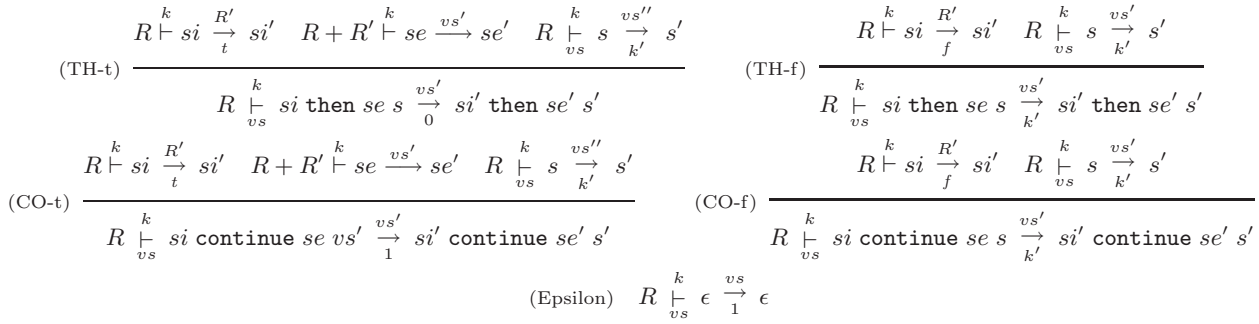

Figure 4: The Synchronous Semantics for escaping conditions

We have two choices for both thus leading to four possible semantics. The semantics proposed in this paper allows situations 1 and 2 and the Lucid Synchrone compiler follows it precisely. Deciding if a more constraining rule must be taken is still unclear. The possibility to compose these two types of transitions is useful in practice and not that much complex. Such sequences of transitions are inherently difficult to understand. More methodological consideration about the right way to design programs that uses both kind of transitions are certainly needed. Such situations are common and expected when combining several paradigms.

5.3 Completing Incomplete Definitions

The semantics proposed here requires a complete definition of streams defined in states. This means that a stream must be defined at every cycle. Since exactly one state is active during a reaction, each state must contain a definition for the streams defined by an automaton.

The language presented in [10] does not need to give a definition of a stream in every state and proposes to maintain the latest computed value. We introduced for that the construction `last` x as a way to refer to the last computed value of a shared value x and absent definitions are implicitly completed with the equation $x =$ `last` x. This choice is pretty comfortable from the user point of view as it strongly reminds the behavior of variables in any imperative languages where the value is maintained as long as no new assignment occurs. This is also the case in Stateflow where the action language is imperative with a more unsafe mechanism since multi-emission (which is feasible in Stateflow) is forbidden here. Indeed, keeping the single assignment property is a key point in the simplification of parallelism provided in synchronous languages.

The signals introduced here correspond to the case where a partial definition is allowed (the single definition per cycle still holds) without filling the holes corresponding to the cycle where the signal is not emitted. Signals have two properties: they may not be always produced and there is no need to introduce memory to maintain their values. Moreover they lead to a nice programming discipline as it is impossible to access they value when the signal is absent.

In Esterel, the choice is to have both an implicit completion of the signal value with its latest one and a way to test that this value is produced in the current cycle. But the access and the test are not necessarily grouped and it may be possible to access the value of a signal which is not emitted, thus raising initialization problems.

5.4 Signals as Clocked Values

The introduction of Esterel-like signals gives an alternative way of programming where only the emissions are considered, the signal being implicitly absent otherwise.

Internally, the proposed mechanism relies on the use of clocks as they were originally introduced in Lustre [14] and Signal [3]. These synchronous language manages *clocked sequences*, that is, sequences which are associated to a type information — a clock — defining the instants where the sequence is present. Thus, they already provide a means to manage streams which are present from time to time. Nonetheless, clocks are somethimes heavy to program with and we were interested in finding a more comfortable way of managing them through a suitable syntactic sugar. In this context, a signal is nothing but a pair made of a clock (or *enable*) and a value sampled on that clock. Thus, if ck stands for the clock type of a value v, the clock type of a signal is a dependent pair $\Sigma(c : ck).ck$ `on` c. A signal boxes (or hide) the internal clock of the stream and corresponds exactly to the actual implementation of a signal in Esterel.

The interesting consequence of this representation is that existing clock calculus [9, 11] extend to this construction.

6. CONCLUSION

In this paper, we have considered an extension of a synchronous data-flow language with a generalized form of state machines called *parameterized state machines* and a means to manage *valued signals*, that is, events carrying a value.

We have proposed a synchronous semantics for the whole language. Such a semantics for a language combining data-flow equations and state machines is new and complements the translation semantics given in previous works. Though not addressed in the paper, all the classical static analysis of synchronous languages (e.g., typing, clock calculus, causality analysis) can be extended to the resulting language. This work reveals that the operational semantics for such a language is intrinsically more complex than a translation semantics into a data-flow kernel. This confirm the interest of the later as a basis for an implementation.

Parameterized states gives an answer to the common need to pass some initial information to a state when entering it. This leads to a safer and more modular programming discipline than what is traditionnaly done through the use of global shared variables and imperative modifications in tools like the SyncChart or Stateflow. Yet, it can be compiled efficiently and combines well with the data-flow nature of the basic language. The ability to manage signals brings an

easy way to manipulate multi-rate data, and leads to a *positive* programming, usually found in imperative languages like ESTEREL.

7. 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] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] 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.

[4] Gérard Berry. The constructive semantics of pure esterel. Draft book, 1999.

[5] Gérard Berry. The esterel v5 language primer, version 5.21 release 2.0. Draft book, 1999.

[6] 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.

[7] 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.

[8] 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.

[9] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[10] 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.

[11] 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.

[12] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, Mar 1968. `http://www.acm.org/classics/oct95`.

[13] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[14] 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.

[15] 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.

[16] 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.

[17] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[18] 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.

[19] The Mathworks. *Stateflow and Stateflow Coder, User's Guide*, release 13sp1 edition, September 2003.

[20] 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.

[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/`, 2005.

[23] 2005. `www.mathworks.com`.

APPENDIX

$$
\begin{array}{lcl}
Def(D_1 \texttt{ and } D_2) & = & Def(D_1) \cup Def(D_2) \\
Def(\texttt{let } D_1 \texttt{ in } D_2) & = & Def(D_2) \\
Def(p = e) & = & fv(p) \\
Def(\texttt{emit } x = e) & = & \emptyset \\
Def(\texttt{reset } D \texttt{ every } e) & = & Def(D) \\
Def(\texttt{match } e \texttt{ with } \{p_i \rightarrow D_i\}) & = & \cup_{1 \leq i \leq n} Def(D_i) \\
Def(\texttt{ automaton } \{sp_i \rightarrow u_i^{k'_i} \texttt{ unless } s_i^{k_i}\}) & = & \cup_{1 \leq i \leq n} Def(u_i) \\
Def(\texttt{let } D \texttt{ in } u) & = & Def(u) \\
Def(\texttt{do } D \ w) & = & Def(D)
\end{array}
$$

$$
\begin{array}{lcl}
Emit(D_1 \texttt{ and } D_2) & = & Emit(D_1) \cup Emit(D_2) \\
Emit(\texttt{let } D_1 \texttt{ in } D_2) & = & Emit(D_2) \\
Emit(p = e) & = & \emptyset \\
Emit(\texttt{emit } x = e) & = & \{x\} \\
Emit(\texttt{reset } D \texttt{ every } e) & = & Emit(D) \\
Emit(\texttt{match } e \texttt{ with } \{p_i \rightarrow D_i\}) & = & \cup_{1 \leq i \leq n} Emit(D_i) \\
Emit(\texttt{ automaton } \{sp_i \rightarrow u_i^{k'_i} \texttt{ unless } s_i^{k_i}\}) & = & \cup_{1 \leq i \leq n} Emit(u_i) \\
Emit(\texttt{let } D \texttt{ in } u) & = & Emit(u) \\
Emit(\texttt{do } D \ w) & = & Emit(D)
\end{array}
$$