Synchronous dataflow pattern-matching

Grégoire Hamon

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 - USA
hamon@csl.sri.com

Abstract. We present a pattern matching operation over variant types for synchronous dataflow languages. Those languages are used in the design of embedded systems, the complexity of those systems has been dramatically increasing, making the need for abstraction mechanisms critical. Variant types offer a powerful mechanism to model complex structured data. The pattern matching operation is defined as a clock operator and answers the need for a synchronous dataflow control structure.

1 Introduction

Variant types and pattern-matching are key features of functional languages. Variant types allow precise definition of structured types, pattern-matching makes it possible to efficiently discriminate between values of these types. We propose here to bring the power of variant types to synchronous dataflow languages. This addition allows us to consider two running questions regarding those languages: the definition of data structures and, studied through the definition of the pattern matching operation, the question of control structures.

1.1 Synchronous dataflow languages

Synchronous dataflow languages are languages dedicated to the definition of *reactive systems* [1], that is, systems that need to interact continuously with their environment and at this one's speed. They have been used successfully in developing control softwares for critical applications. They combine the synchronous approach with dataflow programming. The synchronous approach [2] is based on a hypothesis of immediate reaction of the system, which allows the description of systems in a concurrent but deterministic way. Dataflow languages are functional languages over infinite streams in which systems are defined as a set of equations. Synchronous dataflow languages include Lustre [3] and its industrial counterpart Scade, Signal [4] (strictly speaking, Signal is not functional but relational) and Lucid Synchrone [5].

1.2 A need for data structures

Synchronous dataflow languages have never supported a wide range of datatypes, Lustre supports only basic types (booleans, integer, floating-point numbers) and some forms of array, Lucid Synchrone introduced named records. This comes from the domain of applications of those languages, they have often been used to design low-level controllers implemented in hardware, that is system dealing mostly with booleans. Another reason is that understanding how data structures can be extended to dataflow, and implementing them efficiently in this context has proved to be difficult.

However, systems are more and more complex, consequently they deal with more complex datas and need higher level designs, the need for data-structures arises and is becoming critical. Encoding complex datas into basic types is always possible, but is both time consuming and error-prone, a source of error which must be avoided when dealing with critical systems. Users of Scade are expressing the need to define and use their own datatypes.

1.3 A need for control structures

Synchronous dataflow languages are also lacking control structure. A dataflow program is a set of equations over streams, which is well adapted to represent continuous behaviors. Expressing things like "do this until a condition is true, then do that" in a set of equations is quite challenging. Imperative synchronous languages like Esterel can be used to express evolving systems. The question remains open concerning system with mostly continuous behaviors, but jumping from one continuous behavior to another at certain times. Several works have considered combinations of formalisms (Lustre and Esterel, Lustre and Argos [6], Synchronous Eiffel and the Synchronie Workbench [7], Mode-Automata [8]). Recent developments of Esterel try to extend the language with equations, another possibility is to extend a dataflow language with control structures.

1.4 Clocks as control structures

Dataflow synchronous languages provide a mean to activate/deactivate some parts of a program through the *clock* mechanism. A clock represents the evolution pace of an expression and can be specified using special operators. Dealing with expressions evolving on different paces can introduce inconsistencies, the language provide a static analysis known as the *clock calculus* to avoid them. Clocks give a semantically clean and efficient way to model concurrent process evolving at different speeds.

Using clocks to express control has prooved to be very difficult. Providing control structures not based on clocks is possible, but do not mix well with the rest of the language (Scade provides *condition activation*, Mode Automata introduce automata as a control structure but do not support clocks), and can be done only at the cost of some of its properties.

Initial works on the Lucid Synchrone language have shown that the clock calculus can be defined as a type system close to ML's one. More than an interesting theoretical result, this places clocks in a widely studied framework, and opens the way to various experiments. In particular, ML has shown that strong type system and ease of use are not exclusive, most ML programmers consider the type mechanism as an help and not as a constraint. One question is whether the same can be achieved with clocks, can they be easy to use, for example could we define efficient control structures based on them ?

1.5 Synchronous variants and timed pattern-matching

In this work, we address both the need for data-structures and the need for control structures through the addition of variant types in the Lucid Synchrone language. Variant types are a key features of modern programming language, and are a worthy addition to synchronous dataflow languages. Adding the associated pattern matching operation naturally leads to the definition of a clock operator as the branches of the pattern matching are processes evolving at different paces.

1.6 Plan

In section 2 we introduce timed variants and the pattern matching operation using examples. Those ideas are formalized in section 3. Control structure are interesting if they allow efficient compilation, we consider compilation issues in section 4. In section 5 we compare our approach with related works and discuss the choice of using clocks to structure programs.

2 Timed variants and pattern matching

2.1 Lifting variants to streams

Simple types in dataflow languages are build by lifting scalar types: an integer stream is a stream made of integers. In the same way, variant types can be defined naturally: a variant stream is a stream made of values from the corresponding scalar variant type.

A first example — We consider a program that takes mouse events as input and counts depending on this input. Three distinct events can be received: left click, middle click or right click. This information can be encoded as an integer or as a pair of booleans, it is however more readable and less error-prone to create a type:

```
type event = Left | Right | Middle
```

A value of this type is a stream, evolving with time:

`Left`, `Left`, `Right`, `Middle`, `Right`, ...

is a value of type `event`. A counter increasing on a left click, decreasing on a right click and starting over on a middle click can be implemented as:

```
let up_down mouse = count where
  rec count = 0 ->
    match mouse with
    | Left => (pre count) + 1
    | Middle => 0
    | Right => (pre count) - 1
```

The function `up_down` takes an argument `mouse` of type `event`, and defines a single equation `count`. `count` is a stream, its first value is 0 and is followed by (as indicated by the `->` operator) the value of the stream defined by the `match` expression. The `match` discriminate on possible value of `mouse`. On a left click, it returns the previous value of the counter (`pre count`) incremented by 1, on middle click 0, on right click it decreases the counter. A possible execution of this program is:

e	Middle	Left_clic	Left_clic	Right_clic	Left_clic	...
up_down e	0	1	2	1	2	...

Constructors with arguments — Constructors can also carry arguments. We add a `Key` event occurring whenever the user presses a key on the numeric keypad, this event carries an integer value:

```
type event = Left | Right | Middle | Key of int
```

We can now change the step of the counter using the keyboard:

```
let up_down event = count where
  rec count, step = (0, 1) ->
    match event with
    | Left_clic => pre (count + step), pre step
    | Middle_clic => 0, pre step
    | Right_clic => pre (count - step), pre step
    | Key i => pre count, i
```

A possible execution of this program is:

e	Middle	Left_clic	Key 3	Left_clic	Right_clic	...
up_down e	0	1	1	4	0	...

In both those examples, using variants and pattern matching offers a very clear way to write the program, much cleaner than any encoding using integers or booleans to represent the events.

Inductive types — Inductive types are really interesting, and largely used in generalist languages. However, they are associated to recursive functions, which are not supported in synchronous dataflow language for synchrony reasons. We do not consider such types here.

2.2 Clocks and pattern matching

For this constructions to integrate smoothly into the language, they must be defined carefully. We want to provide a control structure through the pattern matching operation, this calls for a study of its clock.

The first thing to notice is that intuitively, the patterns are defining a clock. The pattern `Key i` for example defines the variable `i` only when the pattern is matched: `i` has a clock defined by the branch itself. More generally, each branch defines a clock (the *branch-clock*) which is true if and only if the branch is matched. In the following table, the notation `Clock C` designates the branch-clock of the `C` branch in the pattern-matching:

`e`	`Middle`	`Left_clic`	`Key 3`	`Left_clic`	`Left_clic`	...
`Clock(Middle)`	t	f	f	f	f	...
`Clock(Left_clic)`	f	t	f	t	t	...
`Clock(Key)`	f	f	t	f	f	...

Those branch-clocks have interesting properties:

- they are all sub clocks of the clock of the matched element (here e): if this element is absent, all the branches are absent.
- they are exclusive: only one branch matches on each instant.
- they are complementary: on each instant, if the filtered element is present, one branch will be present (we force the pattern matching to be complete).

The branch-clocks can be used to define the pattern matching operation as a control structure: we ask the code associated to each branch to be on the corresponding branch-clock: it will be "present", thus executed, only when the branch-clock is true. The `match` construction then combines the results of all the branches. Consider for example:

```
type number = Int of int | Float of float

let float_of_number n =
  match n with
  | Int i => float i
  | Float f => f
```

The type `number` has two constructors, `Int` and `Float`. The function `float_of_number` converts a `number` to a `float`. Supposing `n` has clock cl:

- the branch `Int i` defines a clock c_1, `i` is on clock cl `on` c_1. The expression `float i` is on clock cl `on` c_1.
- the branch `Float f` defines a clock c_2, `f` is on clock cl `on` c_2. The expression `f` is on clock cl `on` c_2.

Moreover, we have the following properties on c_1 and c_2:

$$c_1 \wedge c_2 = \texttt{false} \qquad c_1 \vee c_2 = cl$$

Thanks to these properties, the result of the pattern-matching, which is the combination of the branches is uniquely defined and is on clock *cl*.

The `match` construction is seen here as an extension of the `merge` operator of the language. In traditional functional languages, pattern-matching can be seen as a generalization of the conditional `if` and these two programs are expected to be equivalent:

```
if c then e1 else e2

match c with
| true -> e1
| false -> e2
```

Synchronous dataflow language have two conditionals: `if` and `merge`. The `if` construction is not a control structure: both its argument are expected to be computed before one of them is returned. `merge` rely on the use of clocks and combine complementary streams (when one is present, the other one is absent). Here, we expect an equivalence between those two programs:

```
merge c e1 e2

match c with
| true -> e1
| false -> e2
```

The clocks given to the subexpression `e1` and `e2` in the `merge` expression are equivalent to the one in the `match` one.

Our first examples are not correct with regards to the clocks we propose giving to the construction. All the expressions `pre count` have clock *cl* (where *cl* is the clock of `event`, whereas they should be on their branch-clock. We need here to sample them by the corresponding branch-clock. If we know then branch-clock, we can use the `when` operator for this, but how do we get the branch-clock? We need to introduce a naming mechanism:

```
match event with
...
| (Key i) on c => (pre count) when c, i
```

The `on` construction in the pattern names the branch-clock (here `c`), this name can then be used to sample `pre count`.

This mechanism can be made implicit: naming the branch-clock is only used to sample the free variables occurring in the branch expression, we can systematically (and implicitly) sample all free variables with the branch clock. The original program is in this case correct, and equivalent to the following one (where naming has been made explicit):

```
match event with
...
| (Key i) on c => pre (count when c), i
```

and we see that this programs does not compute the expected result! By looking at the parenthesis we see that what we want (in our first version) is `(pre count) when c`, and what we have here is `pre (count when c)`, which are not equivalent. By making the sampling implicit, we lost the ability to sample not only variables, but also sub-expressions.

Finally, the construction itself can be used to sample the expressions:

```
match event, (pre count) with
...
| (Key i), pc => (pc, i)
```

Here, by filtering `pre count` and having it match every branch with a trivial pattern, we get a variable `pc`, which is on the branch clocks and has the expected value. No naming mechanism is needed here. However, this solution clearly does not scale when the number of sub-expressions to be filtered increases.

In the remaining, we study the first solution, with explicit naming. Implicit naming can be an interesting restriction, but can be expressed from the explicit one. The third proposition clearly does not scale, but is an interesting way to do the sampling when all the branches are using the same sub-expression like in our example.

3 Formalization

We formalize those idea by considering a small functional language over streams, and enriching it with variant types and pattern matching, then giving it a semantics and a clock calculus.

3.1 The core language

The core language contains expressions which are either a constant c (a scalar constant lifted to streams or the definition of a function), a variable x, the application of an scalar operator op, or the application of the `fby` operator. Definitions associate names to expressions, applications of functions or clocks and can be combined concurrently (`AND`) or sequentially (`IN`). The main program is a definition d of the form x `with` D returning the value of x in the declaration D.

$$\begin{aligned} e &::= c \mid x \mid op\,(e, e) \\ &\quad\;\mid c\ \texttt{fby}\ e \end{aligned}$$

$$D ::= D\ \texttt{and}\ D \mid D\ \texttt{in}\ D \mid x = e \mid x = e\ (e) \mid \texttt{clock}\ x = e\ \texttt{in}\ D$$

$$c\ ::= i \mid \texttt{fun}\ x.\ y\ \texttt{with}\ D$$

$$d\ ::= x\ \texttt{with}\ D$$

$$\begin{aligned} i &::= true \mid false \mid 0 \mid \ldots \\ op &::= + \mid \ldots \end{aligned}$$

This language can be given an operational semantics expressing how a program reacts with its environment in an instant. This semantics is made from rules of the form:

$$\mathcal{R} \vdash e \xrightarrow{v} e' \qquad \mathcal{R} \vdash D \xrightarrow{\mathcal{R}'} D' \qquad \mathcal{R} \vdash d \xrightarrow{v} d'$$

where $\mathcal{R}$ in an environment and v a value which can be either a constant or a special value "absent" (noted []). They express the fact that in an environment $\mathcal{R}$, an expression e (resp. a declaration D, a definition d) reduce itself on an expression e' (resp. a declaration D', a definition d') and produces the value v (resp. the new environment $\mathcal{R}'$, the value v). We do not details the rules here, they are given in appendix A, full description of this semantics can be found in [9].

The semantics is partial (there are missing rules), to ensure that a program can be executed within the semantics a clock calculus is defined. This clock calculus is defined as a type system, the language of clocks is the following:

$$\sigma \quad ::= \forall \alpha_1, ..., \alpha_m . \forall K_1, ..., K_k . cl \quad n, m, k \in \mathbb{N}$$

$$\begin{array}{lll} cl & ::= & cl \rightarrow cl \mid \langle s \rangle \mid (k : \langle s \rangle) \\ \langle s \rangle & ::= & \texttt{base} \mid \langle s \rangle \texttt{ on } k \mid \langle s \rangle \texttt{ on not } k \mid \alpha \\ k & ::= & x \mid K \end{array}$$

a clock is a functional clock $cl \rightarrow cl$, a stream-clock $\langle s \rangle$ or a dependent clock $(k : \langle s \rangle)$. Stream-clocks are the base clock `base`, a subclock $\langle s \rangle$ `on` k or $\langle s \rangle$ `on not` k or a clock-variable α. Dependencies k are either a name x (as defined by a `let clock`), or dependency variables K. We also have clock-schemas σ. The clock calculus is defined by predicates of the form:

$$\mathcal{H} \vdash e : cl \quad \mathcal{H} \vdash D : \mathcal{H}' \quad \mathcal{H} \vdash d : cl$$

Stating that in a clock environment $\mathcal{H}$, an expression e (resp. a declaration D, a definition d) has clock cl (resp. defines the clock environment $\mathcal{H}'$, has clock cl). The rules are presented in appendix B, full description of this clock calculus can be found in[9].

3.2 Adding variants and pattern-matching

The language is extended with variant type declarations. Those declarations have the following form:

$$\texttt{type } (\alpha_1, ..., \alpha_m)\ t\ = C_1 \texttt{ of } \tau_1 \mid ... \mid C_n \texttt{ of } \tau_n$$

$\alpha_1, ..., \alpha_m$ are type-arguments of t, $C_1, ..., C_n$ are the constructors for type t, $\tau_1, ..., \tau_n$ are the constructors's arguments. If $\mathcal{F}(t)$ is the set of free-variables in t, we impose the following restriction: $\forall i . \mathcal{F}(\tau_i) \subseteq \{\alpha_1, ..., \alpha_n\}$.

Such a declaration extends the language with expressions and patterns for type t. The pattern-matching operation is a declaration, the branch-clock can be named:

$$\begin{array}{ll} e & ::= \ldots \mid C_1(e) \mid \ldots \mid C_n(e) \\ P & ::= c \mid x \mid C_1(P) \mid \ldots \mid C_n(P) \end{array}$$

$$D ::= \ldots \mid \begin{array}{l} x = \texttt{match}\ e\ \texttt{with} \\ \quad \mid P\ \texttt{on}\ x \Rightarrow d \\ \quad\quad \ldots \\ \quad \mid P\ \texttt{on}\ x \Rightarrow d \end{array}$$

The operational semantics is extended to handle expressions with constructors and pattern-matching. First, instantaneous values, which where either a constant or *absent* ([]) need to be extended with constructors applied to constant arguments:

$$\begin{array}{ll} sv & ::= c \mid C_1(sv) \mid \ldots \mid C_n(sv) \\ v & ::= sv \mid [] \end{array}$$

Notice that we cannot have constructors with argument []. Reduction rules for constructed expressions are given in figure 1. They are synchronous: an expression returns *absent* if and only if its argument is absent.

$$\textsc{c-[]}\quad \frac{\mathcal{R} \vdash e \xrightarrow{[]} e'}{\mathcal{R} \vdash C_i(e) \xrightarrow{[]} C_i(e')} \qquad\qquad \textsc{c}\quad \frac{\mathcal{R} \vdash e \xrightarrow{sv} e'}{\mathcal{R} \vdash C_i(e) \xrightarrow{C_i(sv)} C_i(e')}$$

Fig. 1. Reduction rule for patterns

Reduction rules for the pattern-matching are given in figure 2. They follow the proposed behavior: if the filtered expression is absent (MATCH-[]), all branches must be absent. Otherwise (MATCH), the branch corresponding to the filtered expression returns a result, all the other branches must be absent.

The clock calculus is extended to take the new construction into account. Constructed expressions have the clock of their argument. The pattern matching checks that each branch is on its branch-clock: in the branch P_i on $c_i \Rightarrow d_i$, if the filtered expression is on clock cl, d_i must be on clock cl on c_i. We also restrict the scope of the branch-clocks to the branch by forbidding them to appear in the clock environment. This restriction is not mandatory but restrict the scope of the branch-clock and make the pattern-matching closer to a control-structure. The clock of the whole expression is the clock of the matched expression.

$$\textsc{match-}[]\quad \frac{\mathcal{R},[[]/x] \vdash e \xrightarrow{[]} e' \qquad \forall i \in \{1,...,n\}\ \mathcal{R},[[]/x],[[]/x_i],[F/c_i] \vdash d_i \xrightarrow{[]} d'_i}{\mathcal{R} \vdash \begin{array}{l} x = \texttt{match}\ e\ \texttt{with} \\ \mid P_1(x_1)\ \texttt{on}\ c_1 \Rightarrow d_1 \\ \dots \\ \mid P_n(x_n)\ \texttt{on}\ c_n \Rightarrow d_n \end{array} \xrightarrow{[[]/x]} \begin{array}{l} x = \texttt{match}\ e'\ \texttt{with} \\ \mid P_1(x_1)\ \texttt{on}\ c_1 \Rightarrow d'_1 \\ \dots \\ \mid P_n(x_n)\ \texttt{on}\ c_n \Rightarrow d'_n \end{array}}$$

$$\textsc{match}\quad \frac{\begin{array}{c}\mathcal{R},[v/x] \vdash e \xrightarrow{P_j(v_j)} e' \qquad \mathcal{R},[v/x],[v_j/x_j],[T/c_j] \vdash d_j \xrightarrow{v} d'_j \\ \forall i \in \{1,...,n\} \text{ such that } i \neq j, \quad \mathcal{R},[v/x],[[]/x_i],[F/c_i] \vdash d_i \xrightarrow{[]} d'_i\end{array}}{\mathcal{R} \vdash \begin{array}{l} x = \texttt{match}\ e\ \texttt{with} \\ \mid P_1(x_1)\ \texttt{on}\ c_1 \Rightarrow d_1 \\ \dots \\ \mid P_n(x_n)\ \texttt{on}\ c_n \Rightarrow d_n \end{array} \xrightarrow{[v/x]} \begin{array}{l} x = \texttt{match}\ e'\ \texttt{with} \\ \mid P_1(x_1)\ \texttt{on}\ c_1 \Rightarrow d'_1 \\ \dots \\ \mid P_n(x_n)\ \texttt{on}\ c_n \Rightarrow d'_n \end{array}}$$

Fig. 2. Reduction rules for the pattern-matching

$$\textsc{c-h}\quad \frac{\mathcal{H} \vdash e : s}{\mathcal{H} \vdash C(e) : s}$$

$$\textsc{match-h}\quad \frac{\begin{array}{c}\mathcal{H},[x : cl] \vdash e : cl \\ \forall i \in \{1,...,n\},\ \mathcal{H},[x : cl] \vdash d_i : cl \text{ on } c_i \qquad \forall i \in \{1,...,n\},\ c_i \notin fv_{cl}(\mathcal{H})\end{array}}{\mathcal{H} \vdash \begin{array}{l} x = \texttt{match}\ e\ \texttt{with} \\ \mid P_1\ \texttt{on}\ c_1 \Rightarrow d_1 \\ \dots \\ \mid P_n\ \texttt{on}\ c_n \Rightarrow d_n \end{array} : [x : cl]}$$

Fig. 3. Extended clock calculus

Correctness of the clock calculus — The extended clock calculus has been prooved correct with regards to the extended semantics: any well-clocked program in this calculus evaluates in the extended semantics. The proof is similar to the correctness proof of the original clock calculus, only the new cases have to be considered. The full proof is given in [9](p80).

4 Compilation

We want the pattern matching construction to be a control structure. A control structure is interesting if it finds a efficient implementation, that is, the implementation should execute only required code while using a minimum number of tests.

The idea of the compilation is to translate the pattern matching into a similar construction in the target language. In our case, this target language is OCAML [10], which support variant types and pattern matching. Simple examples, as the following, are translated as-is.

```
match n with
| Int i => float i
| Float f => f
```

The produced code is very efficient. The key point here is that all the subexpressions in the branches are on the corresponding branch-clock. If this is not the case, we first need to move some code:

```
match v with
| true on c => let rec nat = 0 fby (nat + 1) in
               (nat when c)
| false => 0
```

`nat when c` is on the branch-clock cl `on c` (where cl is the clock of `v`). However, `nat` and its definition are on clock cl. This means that `nat` must be computed more often than the rest of the branch, we can't translate this code directly. The program is equivalent to the following one:

```
let rec nat = 0 fby (nat + 1) in
match c with
| true on c => nat when c
| false => 0
```

the definition of `nat` has been moved out of the construction. In this second example, every computation taking place inside the branches are on the branch-clock, the `match` can be compiled as-is. Unfortunately, a simple rewriting is not always enough. Consider the following program:

```
match v with
| true on c => let rec nat = 0 fby (nat + 1) in
               let x = (merge c nat 0) in
               (x when c)
| false => 0
```

If cl is the clock of v, we have in the true branch: x on clock cl, thus computed more often that the branch itself, nat on clock cl on c , the branch-clock, and the result x when c,as required, on clock cl on c. So, nat and the result are on the branch clock, they have to stay under the branch. x is faster, and would need to be moved outside, as in our previous example. However, computation dependencies forbid this: x depends on nat (i.e. has to be computed after it), and is used in the result (i.e. has to be computed before). To compile this program, we would have to rewrite it as:

```
let nat, c =
  match v with
  | true => let rec nat = 0 fby (nat + 1) in
            nat, true
  | false => 0, false in
let x = merge c (nat when c) 0 in
match v with
| true => x when c
| false => 0
```

both clocks and dependencies are respected, every expression in the branches are on their branch-clock, this program can be compiled directly into OCAML. However, this new program is not completely equivalent to the initial one in the sense that it contains more tests (a second pattern matching).

In short, the compilation process translates a match in a match, which gives a very good code. This can be done safely as long as all the sub-expressions in the branches are all on the branch-clock (or a sub-clock of the branch-clock). If a branch contains sub-expressions on a clock faster than the branch-clock, those sub-expressions need to be moved outside of the construction before the traduction can be made. Those code motions might require the match to be split into several ones. Code motion is done automatically by the compiler using the clock infered for each expression. In the current implementation, we restrict this code motion to the simple case, rejecting cases where the match has to be split. This choice was made to keep the construction simple and closer to usual control structures.

5 Conclusion

We have presented the addition of variant types and pattern-matching in a synchronous dataflow language. The addition of variants answers the need for data-structure in those languages. They allow an efficient and clean representation of complex structured data. The associated operation of pattern-matching provides a true control structure to synchronous dataflow languages.

5.1 Related works

Few works have been interested in data types in dataflow languages, most languages provide access to types of the host language through abstract operators.

The support for custom data types and an associated control structure directly in the language seems crucial, allowing simpler and cleaner designs. Using abstract operators makes using two separate languages and requires an encoding as soon as a control structure based on the data type is needed

The definition of a control structure in the language is related to the works on Mode-Automata, and more generally to works on combination of formalisms. Mode-Automata answered the question of combination of formalism in synchronous language by providing a real control structure (the automata) on top of Lustre. Having this structure, systems with several continuous behaviors are easy to define and can be compiled efficiently. We follow here the same approach by providing a real control structure in the language. However, whereas in Mode-Automata this structure is put on top of the language, the match construction is directly part it. Moreover, Mode-Automata do not using the clock mechanism, and are therefore not compatible with it. We choose here to use clocks as the mean to do control, this allows the construction to integrate smoothly into the language and to be entirely compatible with it.

In [11] an encoding of Mode-Automata in Lucid Synchrone was proposed. Using variant types and pattern matching makes this encoding much more simpler and easier to understand. Moreover, the use of the `match` construction allows the compiler to produces a very efficient code, each mode of an automaton corresponding to one branch of a pattern matching operation.

5.2 From functional to synchronous dataflow language

This addition shows that synchronous dataflow language can benefit from adapting powerful constructions developed for functional language. This work also shows that even if the clock calculus is a complex mechanism, carefully defined constructions can be based on it while being easy to use. Inference is a key feature here: it allows the clock calculus to be an help to the programmer, providing strong analysis and good diagnosis, while not being a burden, in a similar way as the type system in ML.

References

1. Harel, D., Pnueli, A.: On the development of reactive systems. In: Logic and Models of Concurrent Systens, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, Springer Verlag (1985)
2. Halbwachs, N.: Synchronous programming of reactive systems. Kluwer Academic Pub. (1993)
3. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. Proceedings of the IEEE **79** (1991) 1305–1320
4. Amagbégnon, P., Besnard, L., Guernic, P.L.: Implementation of the Data-flow Synchronous Language Signal. In: Conference on Programming Language Design and Implementation, La Jolla, California (1995) 163–173

5. Caspi, P., Pouzet, M.: A functional extension to Lustre. In Orgun, M.A., Ashcroft, E.A., eds.: International Symposium on Languages for Intentional Programming, Sydney, Australia, World Scientific (1995)
6. Maraninchi, F.: The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In: IEEE Workshop on Visual Languages, Kobe, Japan (1991)
7. Poigne, A., Morley, M., Maffeis, O., Holenderski, L., Budde, R.: The Synchronous Approach to Designing Reactive Systems. Formal Methods in System Design (1998)
8. Maraninchi, F., Rémond, Y.: Mode-automata: About modes and states for reactive systems. In: European Symposium On Programming, Lisbon (Portugal), Springer verlag (1998)
9. Hamon, G.: Calcul d'horloge et structures de contrôle dans Lucid Synchrone, un langage de flots synchrone à la ML. PhD thesis, Université Paris 6 (2002)
10. The Objective Caml team: The Objective Caml system release 3.07 Documentation and user's manual. Technical report, INRIA (2003)
11. Hamon, G., Pouzet, M.: Modular resetting of synchronous data-flow programs. In: ACM International conference on Principles and Practice of Declarative Programming (PPDP'00), Montreal, Canada (2000)

A Operational semantics

IM-[]

$$\mathcal{R} \vdash c \xrightarrow{[]} c$$

IM

$$\mathcal{R} \vdash c \xrightarrow{c} c$$

TAUT

$$\mathcal{R}, [v/x] \vdash x \xrightarrow{v} x$$

OP-[]

$$\frac{\mathcal{R} \vdash e_1 \xrightarrow{[]} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{[]} e_2'}{\mathcal{R} \vdash op\ (e_1, e_2) \xrightarrow{[]} op\ (e_1', e_2')}$$

OP

$$\frac{\mathcal{R} \vdash e_1 \xrightarrow{c_1} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{c_2} e_2' \quad c = op(c_1, c_2)}{\mathcal{R} \vdash op\ (e_1, e_2) \xrightarrow{c} op\ (e_1', e_2')}$$

FBY-[]

$$\frac{\mathcal{R} \vdash e \xrightarrow{[]} e'}{\mathcal{R} \vdash c\ \texttt{fby}\ e \xrightarrow{[]} c\ \texttt{fby}\ e'}$$

FBY

$$\frac{\mathcal{R} \vdash e \xrightarrow{c'} e'}{\mathcal{R} \vdash c\ \texttt{fby}\ e \xrightarrow{c} c'\ \texttt{fby}\ e'}$$

IN

$$\frac{\mathcal{R} \vdash D_1 \xrightarrow{\mathcal{R}_1} D_1' \quad \mathcal{R}, \mathcal{R}_1 \vdash D_2 \xrightarrow{\mathcal{R}_2} D_2'}{\mathcal{R} \vdash D_1\ \texttt{in}\ D_2 \xrightarrow{\mathcal{R}_2} D_1'\ \texttt{in}\ D_2'}$$

AND

$$\frac{\mathcal{R}, \mathcal{R}_2 \vdash D_1 \xrightarrow{\mathcal{R}_1} D_1' \quad \mathcal{R}, \mathcal{R}_1 \vdash D_2 \xrightarrow{\mathcal{R}_2} D_2'}{\mathcal{R} \vdash D_1\ \texttt{and}\ D_2 \xrightarrow{\mathcal{R}_1, \mathcal{R}_2} D_1'\ \texttt{and}\ D_2'}$$

DECL-E

$$\frac{\mathcal{R}, [v/x] \vdash e \xrightarrow{v} e'}{\mathcal{R} \vdash x = e \xrightarrow{[v/x]} x = e'}$$

DECL-CLOCK

$$\frac{\mathcal{R}, [v/x] \vdash e \xrightarrow{v} e' \quad \mathcal{R}, [v/x] \vdash D \xrightarrow{R'} D'}{\mathcal{R} \vdash \texttt{clock}\ x = e\ \texttt{in}\ D \xrightarrow{R'} \texttt{clock}\ x = e'\ \texttt{in}\ D'}$$

DECL-APP

$$\frac{\mathcal{R}, [v/y] \vdash f \xrightarrow{\texttt{fun}\ x.\ z\ \texttt{with}\ D} f' \qquad \mathcal{R}, [v/y] \vdash (x = e\ \texttt{and}\ D)\ \texttt{in}\ y = z \xrightarrow{R', [v/y]} D'}{\mathcal{R} \vdash y = f\ (e) \xrightarrow{[v/y]} D'}$$

DEF

$$\frac{\mathcal{R} \vdash D \stackrel{\mathcal{R}'}{\rightarrow} D' \quad \mathcal{R}, \mathcal{R}' \vdash x \stackrel{v}{\rightarrow} x}{\mathcal{R} \vdash x \texttt{ with } D \stackrel{v}{\rightarrow} x \texttt{ with } D'}$$

B The clock calculus

IM-H

$$\mathcal{H} \vdash i : \forall \alpha . \alpha$$

ABS-H

$$\frac{\mathcal{H}, [x : cl_1] \vdash D : \mathcal{H}_0 \quad gen_{\mathcal{H}}(\mathcal{H}_0) \vdash y : cl_2}{\mathcal{H} \vdash \texttt{fun } x.\, y \texttt{ with } D : cl_1 \rightarrow cl_2}$$

OP-H

$$\frac{\mathcal{H} \vdash e_1 : \langle s \rangle \quad \mathcal{H} \vdash e_2 : \langle s \rangle}{\mathcal{H} \vdash op(e_1, e_2) : \langle s \rangle}$$

FBY-H

$$\frac{\mathcal{H} \vdash e_1 : \langle s \rangle \quad \mathcal{H} \vdash e_2 : \langle s \rangle}{\mathcal{H} \vdash e_1 \texttt{ fby } e_2 : \langle s \rangle}$$

INST-H

$$\frac{cl = inst\ (\sigma)}{\mathcal{H}, [K : \sigma] \vdash K : cl}$$

DECL-E-H

$$\frac{\mathcal{H}, [x : cl] \vdash e : cl}{\mathcal{H} \vdash x = e : [x : cl]}$$

DECL-CLOCK-H

$$\frac{\mathcal{H} \vdash e : \langle s \rangle \quad \mathcal{H}, [x : (x : \langle s \rangle)] \vdash D : \mathcal{H}' \quad x \notin fv_{cl}(\mathcal{H}')}{\mathcal{H} \vdash \texttt{clock } x = e \texttt{ in } D : \mathcal{H}'}$$

DECL-APP-H

$$\frac{\mathcal{H}, [x : cl_1] \vdash f : cl_2 \rightarrow cl_1 \quad \mathcal{H}, [x : cl_1] \vdash e : cl_2}{\mathcal{H} \vdash x = f\,(e) : [x : cl_1]}$$

AND-H

$$\frac{\mathcal{H}, \mathcal{H}_2 \vdash D_1 : \mathcal{H}_1 \quad \mathcal{H}, \mathcal{H}_1 \vdash D_2 : \mathcal{H}_2}{\mathcal{H} \vdash D_1 \texttt{ and } D_2 : \mathcal{H}_1, \mathcal{H}_2}$$

IN-H

$$\frac{\mathcal{H} \vdash D_1 : \mathcal{H}_1 \quad \mathcal{H}, gen_{\mathcal{H}}(\mathcal{H}_1) \vdash D_2 : \mathcal{H}_2}{\mathcal{H} \vdash D_1 \texttt{ in } D_2 : \mathcal{H}_2}$$

DEF-H

$$\frac{\mathcal{H} \vdash D : \mathcal{H}_0 \quad \mathcal{H}, gen_{\mathcal{H}}(\mathcal{H}_0) \vdash x : cl}{\mathcal{H} \vdash x \texttt{ with } D : cl}$$