Synchronous dataflow pattern matching [1]

Grégoire Hamon [2]

Computing Science Departement,
Chalmers University of Technology,
Göteborg, Sweden

Abstract

We introduce variant types and a pattern matching operation to synchronous dataflow languages. These languages are used in the design of reactive systems. As these systems grow increasingly complex, the need for abstraction mechanisms, in particular, data and control structures, is critical. Variant types provide a mechanism to precisely model structured data. The pattern matching operation, defined as a clock operator, provides an efficient control structure.

Key words: synchronous languages, dataflow, pattern matching, inductive types, clock calculus, controle structure, data structure

1 Introduction

Reactive systems [1] are systems that need to interact continuously with their environment and at the speed imposed by this environment. Such systems are found everywhere, and their number is growing: fly-by-wire control systems, cell phones, washing machines, etc. Alongside their number, their complexity is also dramatically increasing. Faced with this increasing complexity, designers need efficient abstraction mechanisms that can make both the implementation and the certification of reactive systems easier.

We are interested in extending synchronous dataflow languages with both data and control structures. To this end, we propose the addition of variant types and pattern matching, two key features of functional languages. Variant types allow precise definition of structured data. Pattern matching is defined as a control structure.

[1] This material is based on work supported by the National Science Foundation under Grant No. CCR-0086096 through the University of Illinois

[2] Email: hamon@cs.chalmers.se

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1.1 Synchronous dataflow languages

Synchronous dataflow languages are dedicated to the design of reactive systems. They combine the synchronous approach and dataflow programming. The synchronous approach [2] makes the hypothesis that the system reacts instantaneously to any stimulus. This allows a concurrent and deterministic description of systems. Dataflow languages are functional languages over infinite streams. Combining these two approaches allows efficient descriptions of reactive systems with a continuous behavior, as found in most control systems. 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 do not support a wide range of data types. Lustre supports basic types (booleans, integer, floating-point numbers) and some forms of array, Saga [6] introduced named records. Other data structures have to be imported from a host language. Access functions, written in this host language, are then required to manipulate the data. This makes writing programs difficult, as the components of the structure cannot be accessed directly. Moreover, program certification is then complicated by the need to check properties on the host language.

As the complexity of systems is increasing, the need for more complex data structures is becoming critical. Industrial users of synchronous dataflow languages like Scade are expressing the need to define and use their own data types.

1.3 A need for control structures

Synchronous dataflow languages are well adapted to represent continuous behaviors. However, expressing sequential or evolving behaviors with a set of equations is quite challenging. To design truly evolving systems, imperative synchronous languages like Esterel are better adapted. The question remains open concerning systems described as a composition of several continuous behaviors. Several works have considered combinations of formalisms (Lustre and Esterel, Lustre and Argos [7], Synchronous Eiffel and the Synchronie Workbench [8], Ptolemy [9]). Recent developments of Esterel extend the language with equations (Esterel v7). Another possibility is to extend a dataflow language with control structures. Scade provides *condition activations*, Mode-Automata [10] adds automatons as a control structure over Lustre. These additions do not mix well with the rest of the language, Mode-Automata for example have to consider a subset of Lustre.

1.4 Clocks as control structures

Dataflow synchronous languages provide a means to activate/deactivate some parts of a program through a *clock* mechanism. A clock represents the pace of an expression and can be specified using special operators. Dealing with expressions evolving on different paces can introduce inconsistencies. In order to avoid them, a static analysis known as the *clock calculus* is used. Clocks give a semantically clean way to model concurrent processes evolving at different speeds. They are also used by compilers (Signal, Reluc [11], and Lucid Synchrone) to produce efficient code.

Nonetheless, using clocks to express control has proven to be difficult, the mechanism is complex to use. Works on the Lucid Synchrone language have shown that the clock calculus can be defined as a type system close to the type system of ML [12,13]. ML has shown the possibility to combine a strong type system and ease of use – ML programmers consider the type mechanism as a help and not as a constraint. The "clocks as type" point of view of Lucid Synchrone opens the question as to whether the same can be achieved with clocks, making it possible to use them to define control structure. We study this question through the definition of the pattern matching operation associated with variant types.

1.5 Plan

In section 2 we introduce timed variants and the pattern matching operation using examples. These ideas are formalized in section 3. Compilation issues are studied 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 built 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.

2.1.1 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 variant type:

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

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

Left	Left	Right	Middle	Right	...

We implement a counter increasing on a left click, decreasing on a right click and starting over on a middle click:

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

The function `up_down` takes an argument `mouse` of type `event`. It defines a value `count`, which is a stream. Its first value is 0, and is followed by (as indicated by the `->` operator) the value defined by the `match` expression. The `match` discriminates 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, and on right click it decreases the counter. The `when` operator is used to filter a stream. Here `pre count` produces a value on every instant, we filter it in each branch to return a value only when the branch is taken. A possible execution of this program is:

e	Middle	Left	Left	Right	Left	...
up_down e	0	1	2	1	2	...

2.1.2 Constructors with arguments

Constructors can 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 change the step of the counter using the keyboard:

```
let up_down event = count where
  rec count, step = (0, 1) ->
    match event with
    | Left on c => ((pre count) + step, pre step) when c
    | Middle on c => 0, (pre step) when c
    | Right on c => ((pre count) - step, pre step) when c
    | Key i on c => (pre count) when c, i
```

A possible execution of this program is:

e	Middle	Left	Key 3	Left	Right	...
up_down e	0	1	1	4	0	...

In both those examples, using variants and pattern matching offers a clear way to write the program. Without those features, the events would have to be encoded as integers or booleans or imported from a host language. Our solution gives a simpler program, closer from the specification of the program.

2.1.3 Recursive types

Recursive types are interesting, and largely used in generalist languages. They are however 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

We now detail the clock of the pattern matching operation. This allows us to use this construction as a form of control structure.

2.2.1 One clock per branch

A variant stream can be seen as the combination of complementary streams. In this example, `e` is the composition of four streams:

e	Middle	Left	Key 3	Left	Left	Right	...
		Left			Left Left		...
	Middle						...
						Right	...
			Key 3				...

These streams are defined by the patterns: each one corresponds to one pattern in our example. They also have complementary clocks: only one of them is present at each instant. Each branch of the pattern matching thus defines a clock, the *branch-clock*, which is true only when the branch is taken. In the pattern `Key i`, the variable `i` is defined only when the event as the form `Key *`: the clock of `i` is the branch-clock. 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 is taken on each instant. This exclusivity is a property of the pattern matching operation, which as usual, uses the first pattern matching the argument if the patterns themselves are not exclusive.
- 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).

For the pattern matching to define a control structure, we ask the code associated to each branch to be on the corresponding branch-clock: it is "present", thus executed, only when the branch-clock is true. The construction 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 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.

2.2.2 Naming the branch-clock

In our examples, we have explicitly named the branch clock using the `on` construction in patterns:

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

This name is used to sample `pre count`. Otherwise, it would be on a faster clock than the branch itself, as `count` is itself faster (being the result of the `match`).

We might want to make this mechanism implicit: naming the branch-clock is necessary to sample the free variables of the branch expressions. We could systematically (and implicitly) sample all free variables with the branch clock. However, doing that, we loose some expressivity, the two following expressions:

```
pre (x when c)
```

```
(pre x) when c
```

are not equivalent. Using an implicit filtering mechanism, we can only define the first one (i.e. the expression `pre x`, where x is implicitely filtered corresponds to the expression `pre (x when c)`. More generally, an implicit mechanism, while allowing a more lightweight syntax, only allows filtering variables instead of sub-expressions.

The construction itself can be used to sample the expressions:

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

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.

3 Formalization

We formalize those ideas by considering a small functional language over streams, and enriching it with variant types and pattern matching. Then we give it a semantics and a clock calculus. The previous examples translate very simply to this core language.

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 a scalar operator *op*, the application of the `fby` (from which `->` and `pre` are defined), `when`, `whenot`, or `merge` operator. Definitions associate names to expressions, function applications 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.

$$
\begin{aligned}
e &::= c \mid x \mid op\ (e, e) \mid c\ \texttt{fby}\ e \mid e\ \texttt{when}\ e \\
D &::= D\ \texttt{and}\ D \mid D\ \texttt{in}\ D \mid x = e \mid x = e\ (e) \\
c &::= i \mid \texttt{fun}\ x.\ y\ \texttt{with}\ D \\
d &::= x\ \texttt{with}\ D \\
i &::= true \mid false \mid 0 \mid ... \\
op &::= + \mid ...
\end{aligned}
$$

This language is given an operational semantics expressing how a program interacts 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 associating values to variable. A value v can be either a constant or a special value "absent" (noted []). The rules express the fact that in an environment $\mathcal{R}$, an expression e (resp. a declaration D, a definition d) reduces 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 detail the rules here, they are given in appendix A. A full description of this semantics can be found in [14].

The semantics is partial: no semantic is given to non-synchronous programs (for example, `x+y` is only defined if `x` and `y` are either both present or both absent). Ensuring that a program is synchronous is done statically by a clock calculus. This clock calculus is defined as a type system. The language of clocks is the following:

$$\begin{aligned}
\sigma &::= \forall\alpha_1, ..., \alpha_m.\forall K_1, ..., K_k.cl \quad n, m, k \in I\!N \\
cl &::= cl \to cl \mid s \mid (k : s) \\
s &::= \texttt{base} \mid s \texttt{ on } k \mid \alpha \\
k &::= x \mid K
\end{aligned}$$

a clock is a functional clock $cl \to cl$, a stream-clock s or a dependent clock $(k : s)$. Stream-clocks are the base clock `base`, a sub-clock s `on` k, or a clock-variable α. Dependencies k are either a name x (as defined by a `let clock`), or a dependency variable K. We also have clock-schemes 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 $\mathcal{H}$, associating clocks to variables, 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. A full description of this clock calculus can be found in[14].

3.2 Adding variants and pattern matching

3.2.1 Extending the language

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

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

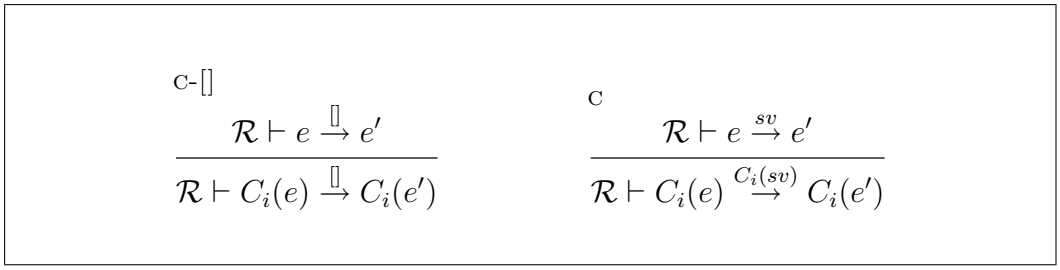

Fig. 1. Reduction rule for patterns

$\alpha_1, ..., \alpha_m$ 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 $\mathcal{F}(u)$ is the set of free-variables in a type u, $\forall i.\mathcal{F}(\tau_i) \subseteq \{\alpha_1, ..., \alpha_n\}$. We don't consider recursive types, thus $\forall i.t \notin \mathcal{F}(\tau_i)$.

Such a declaration extends the language with expressions and patterns for type t. The pattern matching operation is a declaration, in which the branch-clock is named (in the concrete syntax, as presented in the examples, the name can be omitted), the code associated to each branch is a definition:

$$
\begin{aligned}
e &::= ... \mid C_1(e) \mid ... \mid C_n(e) \\
P &::= c \mid x \mid C_1(P) \mid ... \mid C_n(P) \\
D &::= ... \mid \begin{array}{l} x = \texttt{match}\ e\ \texttt{with} \\ \quad \mid P\ \texttt{on}\ x \Rightarrow d \\ \quad ... \\ \quad \mid P\ \texttt{on}\ x \Rightarrow d \end{array}
\end{aligned}
$$

3.2.2 Semantics

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

$$
\begin{aligned}
sv &::= c \mid C_1(sv) \mid ... \mid C_n(sv) \\
v &::= sv \mid []
\end{aligned}
$$

Constructors cannot be applied to []. 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. Reduction rules for the pattern matching are given in figure 2. They follow the proposed behavior:

- If the filtered expression is absent (rule MATCH-[]): all the branch clocks are *false* (c_i takes the value F), all branches are absent, the result is absent.
- Otherwise (rule MATCH): the branch corresponding to the filtered expression's branch clock is *true*(T), it returns a result, which is returned by the

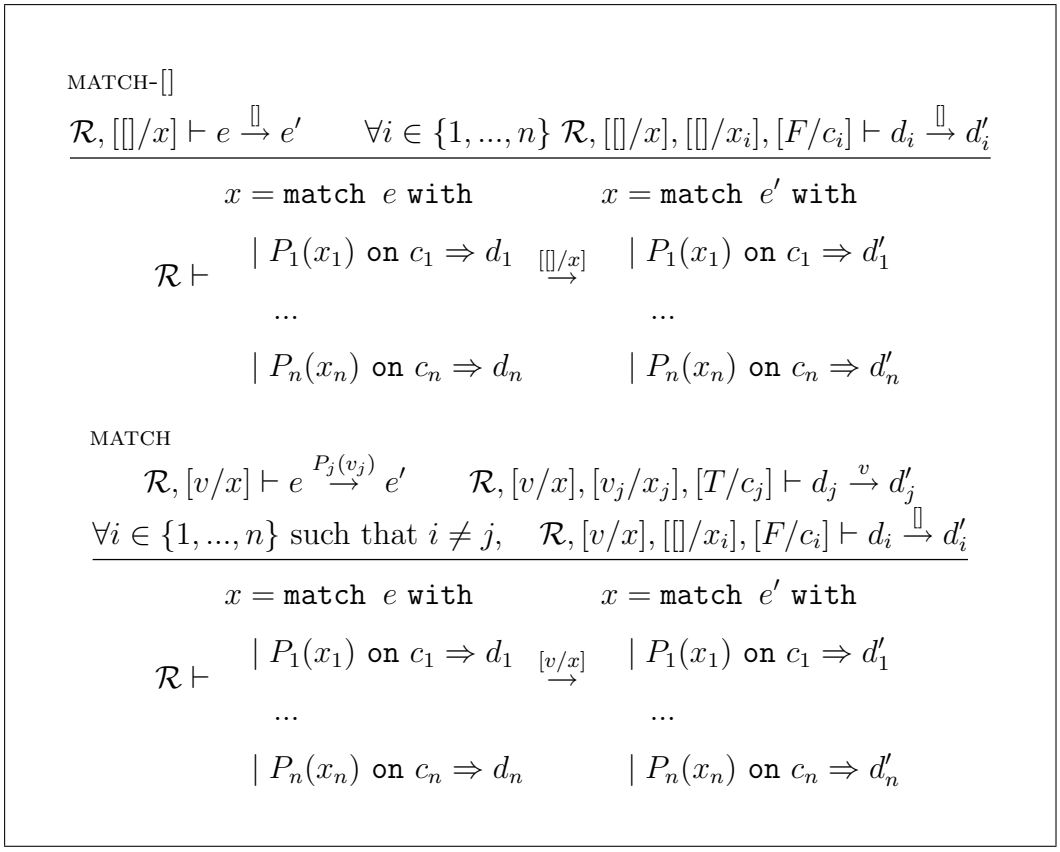

Fig. 2. Reduction rules for the pattern matching

`match`, all the other branches are absent.

3.2.3 Clock calculus

The new rules are given in figure 3. Constructed expressions have the clock of their argument. In the pattern matching 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 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 makes the pattern matching closer to a control-structure. The clock of the whole expression is the clock of the matched expression.

Correctness of the clock calculus

The extended clock calculus has been proved 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 clock calculus of the core language, only the new cases have to be considered. The full proof is given in [14](page 80).

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

$$\text{MATCH-H}\quad \frac{\begin{array}{c}\mathcal{H}, [x : cl] \vdash e : cl \\ \forall i \in \{1, ..., n\} :\ \mathcal{H}, \mathcal{H}_i \vdash P_i : cl \texttt{ on } c_i \quad \mathcal{H}, \mathcal{H}_i, [x : cl] \vdash d_i : cl \texttt{ on } c_i \\ c_i \notin fv_{cl}(\mathcal{H}),\ \ \mathrm{Dom}(\mathcal{H}_i) = fv_{P_i}\end{array}}{\begin{array}{c} x = \texttt{match } e \texttt{ with} \\ \mathcal{H} \vdash \quad \texttt{| } P_1 \texttt{ on } c_1 \Rightarrow d_1 \quad : [x : cl] \\ \quad\quad ... \\ \texttt{| } P_n \texttt{ on } c_n \Rightarrow d_n \end{array}}$$

Fig. 3. Extended clock calculus

4 Compilation issues

Our goal is to use the pattern matching operation as a control structure. It is thus important for this construction to be efficiently compiled. Compiling pattern matching in a traditional language is a known problem[15]. To compile our construction, we translate it into a similar construction in the target language (in our case, Ocaml [16]).

For simple examples, where all the sub-expressions in the branches are on the branch-clock, the compilation is straightforward: the program can be translated as is. However, if some sub-expressions are not on their branch-clock, the translation is not straightforward:

```
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`). Therefore, `nat` and its definition are on clock cl, which is faster than the branch-clock: we can't translate the construction directly. The program is equivalent to the following one:

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

`nat` is defined outside of the construction. All computations taking place

inside the branches are on the branch-clock, the `match` can be compiled as-is. The rewriting is done automatically by the compiler, using the clock of the expressions. Unfortunately, this is not always enough:

```
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 `v` has clock *cl*; `x when c` is on the branch clock (*cl* `on` *c*), `x` is on clock `cl` (faster than the branch clock), and `nat` is on the branch-clock. As before the definition of `x` would need to be moved outside the construction. However, dependencies between computations (`x when c` depends on `x` which depends on `nat`, make it difficult. This program is equivalent to the following one:

```
let nat, c =
  match c 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. However, the structure of the program has not been maintained by the translation. This goes against the idea of the construction as a control structure. The implementation restricts the use of `match` to programs not requiring such rewriting.

5 Conclusion

We have presented the addition of variant types and pattern matching in a synchronous dataflow language. The addition of variants proposes an answer the need for data-structure in those languages. They allow an efficient and clean representation of structured data. The associated operation of pattern matching provides a control structure in 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 designs. Using abstract operators requires using two separate languages and some encodings 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 by providing a control structure (the automata) on top of Lustre. With this structure, systems with several continuous behaviors can be defined and compiled efficiently. We follow here a similar approach by providing a control structure in the language. However, the match construction is here part of the language and integrates smoothly into it. Restrictions imposed on the language in Mode-Automata (no clocks, no functions, `pre` only on variables) are not necessary. This smooth integration is made possible by the use of clocks to define control.

In [17] an encoding of Mode-Automata in Lucid Synchrone was proposed. Using variant types and pattern matching makes this encoding simpler and easier to understand. The use of the `match` construction allows the compiler to produce efficient code, each mode of an automaton corresponding to one branch of a pattern matching operation.

5.2 Adaptations

Variant types and pattern matching as presented here are implemented in the current version of the Lucid Synchrone compiler. Following the same ideas, we can imagine different construction. Free variables appearing in the branches can be implicitly filtered as discussed in section 2.2.2. Compilation issues disappear: branch sub-expressions cannot go faster than the branch in this case.

The experimental version of the Scade compiler, Reluc [11], supports a similar construction, restricted to enumerated types (constructors without arguments). The restrictions we imposed due to compilation issues 4 are not imposed. The construction can get compiled as several control structures in the host language. This construction is used to model automata in the language.

5.3 From functional to synchronous dataflow language

This addition shows that synchronous dataflow language can benefit from constructions developed for functional language.

It also shows that, even if the clock calculus is a complex mechanism, constructions based on it can be easy to use. The clock calculus is essential, allowing the description of concurrent processes evolving at different speed in a safe way. While using it directly to define control is difficult, it can be used more easily through dedicated constructions. The “clock as type” point of vue allows inference and good diagnosis which are essential in defining such constructions. In a similar way as the type system in ML is an help to the programmer and not a burden, the clock calculus must be an help when

combining concurrent processes.

6 Acknowledgements

The author would like to thank Marc Pouzet and Jean-Louis Colaco for their suggestions concerning this work and Carmella Schaecher for her insightful comments. Anonymous reviewers significantly helped in improving this paper.

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] Bergerand, J.L., Pilaud, E.: Saga: a software development environment for dependability in automatic control. In: SAFECOMP'88, Pergamon Press (1988)

[7] Maraninchi, F.: The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In: IEEE Workshop on Visual Languages, Kobe, Japan (1991)

[8] Poigné, A., Morley, M., Maffeis, O., Holenderski, L., Budde, R.: The Synchronous Approach to Designing Reactive Systems. Formal Methods in System Design **12** (1998) 163–187

[9] Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems **18** (1999)

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

[11] Colaco, J.L.: (Personal communication)

[12] Caspi, P., Pouzet, M.: Synchronous kahn networks. In: ACM SIGPLAN International Conference on Functional Programming, Philadelphia, Pensylvania (1996)

[13] Colaço, J.L., Pouzet, M.: Clocks as first class abstract types. In: ACM SIGPLAN International Conference on Embedded Software. Volume 2855 of Lecture Notes in Computer Science., Springer-Verlag (2003) 134–155

[14] 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)

[15] Le Fessant, F., Maranget, L.: Optimizing pattern-matching. In: Proceedings of the 2001 International Conference on Functional Programming, ACM Press (2001)

[16] Leroy, X.: The Objective Caml system release 3.07 Documentation and user's manual. Technical report, INRIA (2003)

[17] 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

$$\text{IM-}[] \quad \mathcal{R} \vdash c \xrightarrow{[]} c \qquad \text{IM} \quad \mathcal{R} \vdash c \xrightarrow{c} c \qquad \text{TAUT} \quad \mathcal{R}, [v/x] \vdash x \xrightarrow{v} x$$

$$\text{OP-}[] \quad \frac{\mathcal{R} \vdash e_1 \xrightarrow{[]} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{[]} e_2'}{\mathcal{R} \vdash op\ (e_1, e_2) \xrightarrow{[]} op\ (e_1', e_2')} \qquad \text{OP} \quad \frac{\mathcal{R} \vdash e_1 \xrightarrow{c_1} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{c_2} e_2' \quad c = op(c_1, c_2)}{\mathcal{R} \vdash op\ (e_1, e_2) \xrightarrow{c} op\ (e_1', e_2')}$$

$$\text{FBY-}[] \quad \frac{\mathcal{R} \vdash e \xrightarrow{[]} e'}{\mathcal{R} \vdash c\ \texttt{fby}\ e \xrightarrow{[]} c\ \texttt{fby}\ e'} \qquad \text{FBY} \quad \frac{\mathcal{R} \vdash e \xrightarrow{c'} e'}{\mathcal{R} \vdash c\ \texttt{fby}\ e \xrightarrow{c} c'\ \texttt{fby}\ e'}$$

$$\text{WHEN-}[] \quad \frac{\mathcal{R} \vdash e_1 \xrightarrow{[]} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{[]} e_2'}{\mathcal{R} \vdash e_1\ \texttt{when}\ e_2 \xrightarrow{[]} e_1'\ \texttt{when}\ e_2'} \qquad \text{WHEN-T} \quad \frac{\mathcal{R} \vdash e_1 \xrightarrow{c} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{true} e_2'}{\mathcal{R} \vdash e_1\ \texttt{when}\ e_2 \xrightarrow{c} e_1'\ \texttt{when}\ e_2'}$$

$$\text{WHEN-F} \quad \frac{\mathcal{R} \vdash e_1 \xrightarrow{c} e_1' \quad \mathcal{R} \vdash e_2 \xrightarrow{false} e_2'}{\mathcal{R} \vdash e_1\ \texttt{when}\ e_2 \xrightarrow{[]} e_1'\ \texttt{when}\ e_2'}$$

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

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

DECL-E
$$\frac{\mathcal{R}, [v/x] \vdash e \overset{v}{\to} e'}{\mathcal{R} \vdash x = e \overset{[v/x]}{\to} x = e'}$$

DEF
$$\frac{\mathcal{R} \vdash D \overset{\mathcal{R}'}{\to} D' \quad \mathcal{R}, \mathcal{R}' \vdash x \overset{v}{\to} x}{\mathcal{R} \vdash x \texttt{ with } D \overset{v}{\to} x \texttt{ with } D'}$$

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

B The clock calculus

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

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

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

FBY-H
$$\frac{\mathcal{H} \vdash e_1 : s \quad \mathcal{H} \vdash e_2 : s}{\mathcal{H} \vdash e_1 \texttt{ fby } e_2 : s}$$

WHEN-H
$$\frac{\mathcal{H} \vdash e_1 : s \quad \mathcal{H} \vdash e_2 : (K : s)}{\mathcal{H} \vdash e_1 \texttt{ when } e_2 : s \texttt{ on } K}$$

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

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