Modular Resetting of Synchronous Data-Flow Programs

Grégoire Hamon
Laboratoire d'informatique de Paris 6
4 place Jussieu
75 252 Paris, France
Gregoire.Hamon@lip6.fr

Marc Pouzet
Laboratoire d'informatique de Paris 6
4 place Jussieu
75 252 Paris, France
Marc.Pouzet@lip6.fr

ABSTRACT

This paper presents an extension of a synchronous data-flow language providing full functionality with a modular *reset* operator. This operator can be considered as a basic primitive for describing dynamic reconfigurations in a purely data-flow framework. The extension proposed here is *conservative* with respect to the fundamental properties of the initial language: reactivity (i.e, execution in bounded memory and time) and referential transparency are kept. The *reset* operator is thus compatible with higher-order. This is obtained by extending the clock calculus of the initial language and providing a compilation method. We illustrate the use of this operator by describing an automatic encoding of Mode-automata. All the experiments presented in the paper has been done with LUCID SYNCHRONE, an ML extension of LUSTRE.

Keywords: Functional languages, Synchronous language, Data-flow, Reactive systems, Hybrid systems.

1. INTRODUCTION

Synchronous languages like ESTEREL [4], ARGOS [13]), LUSTRE [7] or SIGNAL [1] are dedicated to the programming of *reactive systems* – systems which have to react continuously with their environment at the speed imposed by this environment. Synchronous languages are based on the synchronous hypothesis — communications and computations are done in zero time — allowing for a logical description of the system. They are associated with special static analysis (e.g, causality analysis, clock calculus) and compilation methods insuring fundamental properties (e.g, deadlock freedom, determinism, execution in bounded time and memory).

Whereas these languages all share the synchronous hypothesis and are all computationally equivalent, they are quite different in nature. ESTEREL and ARGOS are of imperative style and are thus well adapted to the description of control-based behaviors (e.g, a mouse controller) whereas LUSTRE or SIGNAL take their origin in control-theory and data-flow programming [3] and are thus more adapted to the description of sampled continuous behaviors. Of course, real systems rarely fall into one of these paradigms but are often a combination of control-based behaviors and periodic behaviors. This explains the interest for proposing multi-paradigm languages [10, 16] or Mode-automata, as a way to deal with running modes in a synchronous data-flow language [15, 14].

In this paper, we study the description of reconfigurable systems made of sequences of periodic behaviors inside a purely data-flow framework. Whereas such a description is possible, it is not modular and often needs a complete rewriting of programs. Thus, we propose to add a *reset* operator as a way to restart, in a modular way, a periodic behavior on the reception of a signal. We think that this construction is the basis for reconfiguration. We show that such an extension is *conservative* with respect to the fundamental properties of the initial language: reactivity (i.e, execution in bounded memory and time) and referential transparency (i.e, the ability to replace a name by its definition) are kept. As an example, we present an encoding of Mode-automata into the resulting language.

This *reset* construction has been experimented inside LUCID SYNCHRONE, a functional extension of LUSTRE built on top of OBJECTIVE CAML [11].

The paper is organized as follows. Section 2 gives a short introduction to LUCID SYNCHRONE through an example. Section 3 explores the behavior of a reset, discusses the semantical problems raised by this construction and the solution we have adopted. Section 4 is the formalization section. Finally, section 5 presents an encoding of Mode-automata into the language extended with the reset construction.

2. AN EXAMPLE

This section gives a brief introduction to LUCID SYNCHRONE[1] through an example. Suppose we have to describe the movement of a ball in the earth gravity field (figure 1). For this purpose, we first define an integration step `dt` (the sampling rate of the system) and a rectangular integration `integr`:

```
let dt = 0.1
let integr x0 dx = x where
  rec x = (x0 fby x) +. dx *. dt
```

[1] A tutorial of the language can be found at `www-spi.lip6.fr/lucid-synchrone`.

PPDP '00, Montréal, Canada.

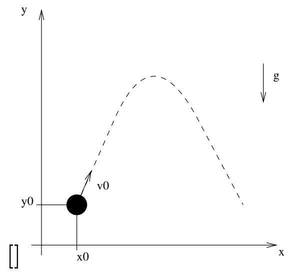


Figure 1: A ball in the earth gravity field

The language is a data-flow language, meaning that it manages infinite sequences or *streams* as primitive values. Thus, `0.1` stands for an infinite stream and `float`, for the type of streams of floats. The `integr` function receives two streams `x0` and `dx` and integrates the signal `dx`. Scalar operators, like `+.` or `*.`, are applied point-wisely on streams. `fby` is an initialized delay on streams, it returns the initial value of its first argument then the previous value of its second argument (i.e, $(a \texttt{ fby } b)_n = a_1$ for $n = 1$ and b_{n-1} otherwise). For example:

time	1	2	3	4	5	...
`x0`	0.0	0.0	0.0	0.0	0.0	...
`dx`	1.0	1.0	1.0	1.0	1.0	...
`integr`	0.1	0.2	0.3	0.4	0.5	...

Given the declaration of `integr`, the compiler automatically infers its type (`:`) and its clock (`::`):

```
node integr :  float -> float -> float
node integr :: 'a -> 'a -> 'a
```

stating that `integr` takes two (streams of) floats and returns a (stream of) floats. The clock is a Boolean information indicating for every stream the instants where it is defined. The clock `'a -> 'a -> 'a` is read like this: for any clock `'a`, if the arguments `x0` and `dx` are on clock `'a`, then the result is on clock `'a`. Said differently, the function `integr` emits an item every time it receives an item of `x0` and `dx`. Now, using this integration function, we can express the movement of the ball on one axis. Combining two movements, we obtain the expected result:

```
let movement x0 dx0 d2x0 =
  integr x0 (integr dx0 d2x0)

let ball (x0,y0,dx0,dy0) = (x,y)
  where x = movement x0 dx0 0.
  and   y = movement y0 dy0 (-. 9.81)

let throw = ball (0., 0., 50., 50.)
```

This simple program computes the stream of the successive positions of the ball. Suppose now that we would like to suspend the computation when the user is clicking on the mouse button. `throw` is now a function, taking a Boolean input `click`, the status of the mouse button:

```
let sthrow click =
  ball ((0., 0., 50., 50.) when (not click))

node sthrow :: (click_1:'a) -> 'a on not click_1
```

The `when` operator filters a stream according to a Boolean condition, making the arguments of `ball` only present at the instants where `click` is false [2]. Thus, the position of the ball only evolves at these specific instants. The clock information states that if the argument `click` is on clock `'a`, then the result is on a sub-clock `'a on not click`, i.e, the result is present when `click` is present and false. Using `when` we managed to suspend the computation, however, when the user is clicking on the mouse, `sthrow` doesn't return any value, it is absent. To be able to read the position, we would like that the function returns the last computed value while the user was clicking. We use the `merge` operator:

```
let throw click = (x,y) where
  rec (x,y) = merge cond
       (ball ((0., 0., 50., 50.) when cond))
       (((0.,0.) fby (x,y)) when not cond)
  and cond = not click
```

`merge` is an oversampling operator, combining two complementary streams (with opposite clocks). Here, we combine the stream computing the position of the ball when the user does not click and a stream returning the previous value of the position. The time diagrams are given below:

`click`	f	t	f	f	t	t	f	...
`sthrow click`	v_1	$\perp$	v_2	v_3	$\perp$	$\perp$	v_4	...
`throw click`	v_1	v_1	v_2	v_3	v_3	v_3	v_4	...

This example introduced the main synchronous primitives. Being a functional language, functions can also be abstracted over functions.

3. A RESET OPERATOR

3.1 Intuitive Behavior

We would like now to re-throw the ball every time the user clicks on the mouse button (for example with a different initial speed). A `reset` operator allows a very simple and natural expression of this behavior. Re-throwing the ball consists in restarting the `ball` function or *resetting* it. We would like to be able to write the following program:

```
let throw (dx0,dy0) click =
  reset click ball (0., 0., dx0, dy0)
```

The intuitive behavior of `reset` is simple: it takes a signal (`click`), a function (`ball`) and some inputs `(0.,0.,dx0,dy0)` and restarts the execution of `ball (0.,0.,dx0,dy0)` every time `click` is true. Considering a simpler example:

```
let up i = x where rec x = i fby (x + 1)
```

where `up` is a function counting from an initial value `i`, an execution of `reset c up 1` could be:

`c`	f	f	t	f	f	t	f	...
`up 1`	1	2	3	4	5	6	7	...
`reset c up 1`	1	2	1	2	3	1	2	...

The expected behavior of (`reset c up`) is thus the following: (`reset c up x`) acts as (`up x`) until `c` is true and after `c` has been true, `reset c up x` acts as `reset c' up x'` where `c'` (resp. `x'`) is the sub-stream of `c` (resp. `x`) starting

[2] `click` is renamed `click_1` in order to avoid clashes between names.

when c is true. This behavior is clearly recursive and can be defined as the combination of a stream ending when c is true and a stream starting at this same instant. We have:

```
let whilenot c = o where
  rec o = true -> if c then false else pre o

let rec reset_up c x =
  let cond = whilenot c in
  merge cond
     (up (x when cond))
     (reset_up (c when not cond) (x when not cond))
```

Unfortunately, recursive functions are forbidden in the language when dealing with real-time systems since the compiler may not be able to produce finite state machines. However, we may notice that this function is a particular case of recursion and can be related to *tail-recursion* in classical programming languages: the caller will never be used after the recursive call. Nonetheless, recognizing those recursions in a higher-order synchronous data-flow language and providing static analysis (e.g, clock calculus) and compilation methods for them is still an open problem.

3.2 Resetting Without Reset

Looking at our previous example, one could wonder if the `reset` couldn't be obtained by a simple rewriting of the original code. Indeed, we can write:

```
let reset_up res x = o where
  rec o = if res then x else (x fby (o + 1))
```

Thus `reset res up x` should have the same behavior as `reset_up res x`. Here, we just add the reset condition as a parameter and use it to return `x` every time the signal is true. Following this example, resetting a function on the reception of a signal can be obtained by adding an extra control wire to every initialized delay contained in the function. This wire is used for resetting the delays when the reset condition is true. Thus, it is always possible to create a "resettable" version of a function by rewriting delays:

x `fby` y $\Rightarrow$ `if` res `then` x `else` (x `fby` y)

However, several problems make this transformation hard to do explicitly (by hand) in practice:

3.2.1 Modularity

The reset signal has to be carried everywhere in the program in order to reach all the `fby`, thus causing modifications in the whole code. If we try to reset our ball example, we obtain:

```
let integr res init dx = y
  where rec y = (if res then init else
                (init fby y)) +. dx *. dt

let movement res x0 dx0 d2x0 =
  integr res x0 (integr res dx0 d2x0)

let ball res (x0,y0,dx0,dy0) = (x,y)
  where x = movement res x0 dx0 0.
  and   y = movement res y0 dy0 (-. 9.81)

let throw clic (dx,dy) =
  ball clic (0., 0., dx, dy)
```

the reset signal has to be carried up to the `fby` operator, in the integration function. When a function contains several initialized delays, all on the same clock, then the reset condition can be shared by all these delays. We can notice that, on this example, the modifications could be greatly simplified using mechanisms like implicit parameters [12].

3.2.2 Clocks

Some care must be taken when dealing with functions containing filtering or over-sampling operations (`when` or `merge`). The clock of the reset condition must be consistent with the clock of the expression. Consider the following example:

```
let f x =
  let rec nat = 0 fby (nat + 1) in
  merge x nat 0
```

`nat` is defined only when `x` is true (i.e, it has clock `'a on x`). Translating `f` into the following function:

```
let f' res x =
  let rec nat = if res then 0
               else (0 fby (nat + 1)) in
  merge x nat 0
```

would impose that `res` be on clock (`'a on x`) meaning that `nat` is only reset when `res` and `x` are true. Thus (`f' c x`) is certainly not equivalent to (`reset c f x`). In order to be able to reset `nat` even when `x` is absent, the programmer should add some extra code defining the variable `res`. The problem becomes even harder when the program contains several `fby` with different clocks as in:

```
let f x =
  let rec nat = 0 fby (nat + 1) in
  0 fby (merge x nat 0)
```

The reset conditions of these two `fby` operators can not be the same since they do not share the same clock.

Achieving the reset by means of a hand-coded program transformation is thus not straightforward. It requires the knowledge of clock information, and is certainly not modular. This is typically what a compiler should do. Thus we choose to provide in the language a dedicated reset construction. The compilation method consists in producing a new program where initialized delay restart with their initial value as it is presented in this section.

3.3 Scope of the Reset Operator

The behavior given for the reset is still intuitive, and we quickly find the need to make it more precise. If we consider the following definition:

```
let rec nat = 0 fby (nat + 1) in

let add_nat n = 0 fby (n + nat) in
reset c add_nat 1
```

our intuitive definition already shows its limits. Here `nat` is a free variable in the definition of `add_nat`. Should `nat` be reset with `add_nat` or not? Both possibilities are of interest, depending on the needs of the programmer. We shall study both solutions, and see that each of them introduces semantical problems, and we shall explain the choice made in this paper.

3.3.1 A Local Reset

The first solution is to define a *local* reset, only resetting the delays present inside the body of the function. In the previous example this means not resetting `nat` when `add_nat` is reset. This allows for the definition of global variables used in several functions, like global counters and which do not have to be reset by those functions.

This solution introduces a major semantical problem as it breaks the fundamental referential transparency property of the language:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \;=\; e_2[e_1/x] \text{ and } (\lambda x.e_2)e_1 \;=\; e_2[e_1/x]$$

Here, applying this principle on the previous example should not change the semantics of the program. We obtain:

```
let add_nat n =
  let rec nat = 0 fby (nat + 1) in
  0 fby (n + nat) in
reset c add_nat 1
```

However, doing the substitution here changes the result: the `fby` from nat is now in the scope of the reset and is thus reset.

This substitution principle is essential in synchronous data-flow languages and should be kept.

3.3.2 A Global Reset

The second solution is to define a *global* reset. This means that `nat` should be reset every time the function `add_nat` is reset. The immediate result is that this solution should keep the substitution property. However, we still encounter problems. What shall be done if `nat` is shared by several functions as in the example:

```
let rec nat = 0 fby (nat + 1)
in
let add_nat n = 0 fby n + nat in
let sub_nat n = 0 fby n - nat in
reset c add_nat (sub_nat 1)
```

when `add_nat` is reset, `nat` should be reset as it appears in its definition. However, why should the `nat` appearing in the definition of `sub_nat` be reset too? This is a problem of variable sharing and, at least, two solutions can be investigated:

- duplicate `nat`. Clearly, here, two copies of `nat` would be needed, one continuing its execution and one which is reset every time `c` is true. However this is against the sharing expressed by the `let` construction: in a synchronous data-flow language, `nat` is really a memory cell, not a function. The problem becomes harder in case of rewinding as in:

  ```
  let rec x =
    reset c (fun v -> 0 fby x + v) (0 fby x)
  ```

  Finally, what shall be done when a free variable appearing in the body of the function to be reset depends on an input of the system? An entry cannot be recomputed: the only way to restart a function depending on an input is to store the successive values of the input. Thus reactivity of programs can no longer be guaranteed (the execution in bounded memory may be lost).

- the second solution is to forbid such cases, that is, to forbid different occurrences of a stream variable to be reset on different conditions. The computations of the stream `nat` reset when `c` is true and the computation of `nat` can not be shared. If both streams are really needed, the user must define a function and use two separate instances of this function.

3.3.3 The Proposed Solution

The reset operator has to keep reactivity and has to keep the substitution principle. To ensure these two properties, the operator presented in this paper is a global reset and the use of multiple instances of a variable reset on different conditions is forbidden.

The restriction on the use of multiple instances of a variable is more a specification than a restriction: two instances of a variable reseted on different conditions should be unshared, they are definitively not the same stream.

There is no definitive argument between a local and a global reset. However, in synchronous data-flow languages, we think that the substitution principle is primary. Moreover, we want this operator as modular as possible, and fully compatible with higher order and separate compilation. Here, only the global reset offers this safety.

As a final note, a local reset can be simulated by a global one, by adding extra parameters for the free variables of the function to be reset. For example, if we do not want to reset `nat`:

```
let rec nat = 0 fby (nat + 1) in
let add_nat (var_nat, n) = 0 fby n + var_nat
in (reset c add_nat) (n,1)
```

4. FORMALIZATION

We describe a LUCID SYNCHRONE kernel and its denotational semantics on streams, following the presentation of [5, 6]. It is essentially a functional kernel complemented with special primitives managing synchronous streams.

4.1 The Basic Language and its Semantics

A LUCID SYNCHRONE program manages streams as primitive values and is built above a host language providing scalar values (e.g, integer constants, arithmetic operators). An expression (e) may be a variable (x), a stream recursion ($\texttt{rec}\ x.e$) [3], an abstraction ($\lambda x.e$), an application ($e\,e$) or a expression managing streams. sc stands for scalar values imported from the host language. Scalar values can be lifted using the constant generator `const` and the point-wise application `extend`. `const` takes an extra implicit parameter giving its clock [4].

$$\begin{array}{rcl} e & ::= & x \mid \texttt{rec}\ x.e \mid \lambda x.e \mid e\,e \\ & & \mid \texttt{merge}\ e\,e\,e \mid e\,\texttt{fby}\ e \mid e\,\texttt{when}\ e \\ & & \mid \texttt{const}[ck]\ sc \mid \texttt{extend}\ e\ e \\ sc & ::= & \mid \texttt{true} \mid \texttt{false} \mid + \mid \ldots \\ ck & ::= & x \mid ck\ \texttt{on}\ e \end{array}$$

[3] x is a stream value and not a function.

[4] As shown in previous sections, the programmer does not have access to these lifting operators. Because stream values (`node`) and scalar values are strongly separated — they are defined in different modules —, a value imported from the scalar level is automatically lifted to the level of streams. In the same way, the extra argument of `const` is automatically inferred by the clock calculus.

Basic values of the language are named *clocked streams*. A clocked stream of type T is made of:

- a main stream s of values of type T
- a Boolean stream, named a *clock*, indicating the instant where s is defined.

Their formal definition is:

$\texttt{Val}\ T = \bot\ +\ T$
$\texttt{Stream}\ T = T.(\texttt{Stream}\ T)$
$\texttt{CStream}\ T = \texttt{Stream}\ (\texttt{Val}\ T)$

$\bot$ denotes the absence of value. Access functions `hd` and `tl` are defined as usual:

$\texttt{hd}\ (x.l) = x$
$\texttt{tl}\ (x.l) = l$

The clock of a stream is a Boolean stream (f stands for $false$ and t for $true$) such that:

$\texttt{clock}\ (\bot.cc) = f.(\texttt{clock}\ cc)$
$\texttt{clock}\ (v.cc) = t.(\texttt{clock}\ cc)$

4.1.1 Lifting Constants and Primitives

Scalar values may be transformed into stream values, using the primitives `const` and `extend`.

`const` takes a value and a clock ck and builds a constant stream made with this value. The clock of ($\texttt{const}\ [ck]\ v$) is ck.

$\texttt{const}\ [t.cs]\ \ v = v.(\texttt{const}\ [cs]\ v)$
$\texttt{const}\ [f.cs]\ \ v = \bot.(\texttt{const}\ [cs]\ v)$

`extend` transforms a scalar primitive into a stream primitive.

$\texttt{extend}\ \ (\bot.fc)\ \ (\bot.vc) = \bot.(\texttt{extend}\ fc\ vc)$
$\texttt{extend}\ \ (e.ec)\ \ (v.vc) = (e\ v).(\texttt{extend}\ ec\ vc)$

We may notice that this semantics is partial: the two arguments and the result are all on the same clock (they are synchronous), otherwise the function is not defined (and corresponds to a pattern-matching failure). The purpose of the clock calculus is to statically reject such cases.

4.1.2 Delay

`fby` (for "followed-by") is the unitary delay: it conses the head of its first argument to its second argument.

$xc\ \ \texttt{fby}\ \ yc = \texttt{fby1}\ xc\ yc$
$\texttt{fby1}\ \ (\bot.xc)\ (\bot.yc) = \bot.(\texttt{fby1}\ xc\ yc)$
$\texttt{fby1}\ \ (x.xc)\ (y.yc) = x.(\texttt{fby2}\ y\ xc\ yc)$

$\texttt{fby2}\ \ z\ (\bot.xc)\ (\bot.yc) = \bot.(\texttt{fby2}\ z\ xc\ yc)$
$\texttt{fby2}\ \ z\ (x.xc)\ (y.yc) = z.(\texttt{fby2}\ y\ xc\ yc)$

Here again, the arguments and the result of `fby` must be on the same clock. `fby` corresponds to a two-state machine: while the two arguments are absent, it emits nothing and stays in its initial state `fby1`. When the two arguments become present, it emits its first argument and goes into the new state `fby2` storing the previous value of its second argument. In this state, it emits a value every time its two arguments are present.

4.1.3 Sampling operators

Sub-streams may be built from streams, using the operator `when`.

$(\bot.ec)\ \ \texttt{when}\ \ (\bot.cs) = \bot.(ec\ \texttt{when}\ cs)$
$(e.ec)\ \ \texttt{when}\ \ (f.cs) = \bot.(ec\ \texttt{when}\ cs)$
$(e.ec)\ \ \texttt{when}\ \ (t.cs) = e.(ec\ \texttt{when}\ cs)$

The two arguments of a `when` operator must be on the same clock but what is the clock of its result? The resulting stream is present only when the condition is present and true. Let ck be the clock of its two arguments and c, the Boolean condition. The clock of the result will be noted ck `on` c. Its definition is:

$(f.cls)\ \ \ \texttt{on}\ \ (\bot.cs) = f.(cls\ \ \texttt{on}\ cs)$
$(t.cls)\ \ \ \texttt{on}\ \ (f.cs) = f.(cls\ \ \texttt{on}\ cs)$
$(t.cls)\ \ \ \texttt{on}\ \ (t.cs) = t.(cls\ \ \texttt{on}\ cs)$

Here again, the semantics is partial. The second condition must be on the clock ck.

Sub-streams can be combined with the `merge` operator.

$\texttt{merge}\ \ (\bot.cc)\ (\bot.xc)\ (\bot.yc) = \bot.(\texttt{merge}\ cc\ xc\ yc)$
$\texttt{merge}\ \ (t.cc)\ (x.xc)\ (\bot.yc) = x.(\texttt{merge}\ cc\ xc\ yc)$
$\texttt{merge}\ \ (f.cc)\ (\bot.xc)\ (y.yc) = y.(\texttt{merge}\ cc\ xc\ yc)$

($\texttt{merge}\ \ cl\ \ xc\ \ yc$) merges two complementary sub-streams xc and yc. The clock of the result is the clock of the condition, provided xc is present when the condition is true and yc is absent. Conversely, xc must be absent and yc must be present when the condition is false.

These operators are the basic primitives of a synchronous language like LUSTRE or LUCID SYNCHRONE. The semantics for abstraction, application and recursion is the usual one.

Such a kernel can thus be entirely simulated inside any lazy language (e.g, HASKELL [9]). Nonetheless, it is only a simulation in the sense that streams will be allocated and deallocated at every instant and that every operator will test for presence or absence of its arguments. Data-flow synchronous languages have developed much better compilation techniques where programs are compiled into transition systems [8, 2, 6]. This aspect is not addressed here.

4.1.4 Synchronous Execution and Reactivity

The denotational semantics we have given for stream primitives is partial and may lead to pattern-matching failures. We say that a program can be *synchronously* evaluated when there is no pattern-matching failure during the execution. This definition can be formulated by complementing the domain of clocked streams with a special error value `fail` such that:

$\texttt{FCStream}\ \ T = \texttt{CStream}\ \ T + \texttt{fail}$

Now, every stream primitive *op* with a partial definition is complemented with a rule of the form:

$op\ e_1 \ldots e_n = \texttt{fail}$

Definition 1 (Synchronous execution). *A stream expression e can be synchronously evaluated if $e \not\to_\beta^* \texttt{fail}$ and if $e \to_\beta^* v.e'$ then e' can be synchronously evaluated.*

It is easy to characterize a reactive subset of this synchronous kernel: reactive programs are programs which can be synchronously evaluated, where recursive stream definitions are always guarded by delays and which do not use recursive functions. For such a kernel, programs can be compiled into finite state machines [5, 6].

4.2 Clock Calculus

The goal of the clock calculus is to provide statically checkable conditions allowing an expression to be synchronously evaluated. In Lucid Synchrone, clocks play the same role as types in ML languages, that is, to avoid some kinds of run-time errors.

Theorem 1 (Correction). *Well clocked programs can be synchronously evaluated.*

The clock calculus asserts the judgment:

$$H \vdash e \, : cl$$

meaning that "expression e has clock cl in the environment H". A clock cl is either a clock variable α, a sub-clock of a clock, cl on e monitored by some Boolean stream expression e, or a clock function $(x : cl) \to cl$. Clock expressions are decomposed into clock schemes (σ) and clock instances (cl).

$$\begin{array}{lll} cl & ::= & \alpha \mid cl \;\texttt{on}\; e \mid (x : cl) \to cl \\ \sigma & ::= & cl \mid \forall \alpha . \sigma \end{array}$$

An environment H is a list of assumptions on the clocks of free variables of e. We suppose that the names x_i are distinct from each others:

$$H ::= [x_0 \, : \sigma_0, ..., x_n \, : \sigma_n]$$

$FV(\sigma)$ is the set of free clock variables of cl whereas $fv(\sigma)$ is the set of free expression variables of cl. FV and fv are extended to be applied to environments. Clocks may be generalized or instantiated in the following way:

$$\begin{array}{l} Gen_H(cl) = \forall \alpha_1, ..., \alpha_n . cl \text{ if } \alpha_1, ..., \alpha_n \notin FV(H) \\ Inst(cl) = cl \\ Inst(\forall \alpha_1, ..., \alpha_n . cl) = cl[cl_1/\alpha_1, ..., cl_n/\alpha_n] \end{array}$$

Axioms and inference rules of the clock system are given in figure 2. The constant generator `const` has any clock cl. The arguments and result of a `fby` or a `extend` are all on the same clock. The clock of (e `when` c) is (α `on` c) if e and c have clock α. `merge` needs its two arguments to be on complementary clocks. The clocking rules for application and abstraction are the one of a dependent type system. For example, when clocking a function, we must keep track of the fact that the argument x may appear in the clock of the result and must not appear in the hypothesis.

Equality between clocks is considered modulo β-conversion.

After the clock calculus, a clock can be associated with every sub-expression. In particular, the constant generator (`const`) receives an extra argument — its clock — and may be given a denotational semantics on clocked streams [5].

[5]In practice, clocks are used for compiling memory operator (i.e, delays) and for optimization purposes. Using clocks, the compilers statically knows which part of the expression have to be computed at every instant.

$$\text{CONST} \quad H \vdash \texttt{const}\; i \, : cl$$

$$\text{FBY} \quad H \vdash \texttt{fby} : \forall \alpha . \alpha \to \alpha \to \alpha$$

$$\text{EXTEND} \quad H \vdash \texttt{extend} : \forall \alpha . \alpha \to \alpha \to \alpha$$

$$H \vdash \texttt{merge} : \forall \alpha . (x : \alpha) \to \alpha \;\; \texttt{on}\; x \to \alpha \;\; \texttt{on not}\; x \to \alpha$$

$$\text{WHEN} \quad H \vdash \texttt{when} : \forall \alpha . \alpha \to (x : \alpha) \to \alpha \;\; \texttt{on}\; x$$

$$\text{ABST} \quad \frac{H, x \, : cl \vdash e \, : cl' \quad x \notin fv(H)}{H \vdash \lambda x . e \, : (x : cl) \to cl'}$$

$$\text{APP} \quad \frac{H \vdash f \, : (x : cl) \to cl' \quad H \vdash e \, : cl}{H \vdash f \; e \, : cl'[e/x]}$$

$$\text{REC} \quad \frac{H, x \, : cl \vdash e \, : cl \quad x \notin fv(H)}{H \vdash \texttt{rec} \;\; x . e \, : cl}$$

$$\text{INST} \quad \frac{cl = Inst(\sigma)}{H, x : \sigma \vdash x : cl}$$

$$\text{LET} \quad \frac{H \vdash e_1 : cl_1 \quad H, x : Gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let} \;\; x = e_1 \;\texttt{in}\; e_2 \, : cl_2[e_1/x]}$$

Figure 2: The basic clock calculus

4.3 Adding a Reset Operator

We complement the language with a reset operator. Clocks are also enriched with a new construction cl `every` e. An expression on this clock is reset every time e is true.

$$\begin{array}{lll} e & ::= & \ldots \mid \texttt{reset}\; e \; e \; e \\ ck & ::= & \ldots \mid ck \;\texttt{every}\; e \end{array}$$

As said previously, the reset application (`reset` c f e) acts as (f e) while c is false and restarts a new application every time c is true. f acts as a totally new function every time c is true. Thus, every stream used by f must restart with its initial value.

This reset information can be taken into account in a simple way by extending the notion of clock. A clock is now a pair of Boolean streams, the first one indicating when the stream is present and the second one indicating when the stream is reset. Streams are now defined by:

$\texttt{Val}\; T = (\perp \;\; + \;\; T) \times \texttt{bool}$
$\texttt{CRstream}\; T = \texttt{Stream}\; (\texttt{Val}\; T)$
$\texttt{FCRstream}\; T = \texttt{CRStream}\; T + \texttt{fail}$

Access functions and the point-wise product of streams are defined as follows:

$\texttt{fst}\;\; (v, r).l = v.\texttt{fst}\;\; l$
$\texttt{snd}\;\; (v, r).l = r.\texttt{snd}\;\; l$
$\texttt{prod}\;\; (v.vs)\; (w.ws) = (v, w).\texttt{prod}\;\; vs\; ws$

The presence and resetting of a stream are defined by:

$\texttt{pres}\;\; (s) = \texttt{clock}(\texttt{fst}\;\; s)$

$\texttt{res}\ (s) = \texttt{snd}\ s$

The semantics of lifting primitives `const` and `extend` is modified in this way:

$\texttt{rconst}\ cl\ v = \texttt{prod}\ (\texttt{const}\ [\texttt{pres}\ cl]\ v)(\texttt{res}\ cl)$
$\texttt{rextend}\ v\ w = \texttt{prod}\ (\texttt{extend}\ (\texttt{fst}\ v)\ (\texttt{fst}\ w))(\texttt{snd}\ v)$

4.3.1 *The New Semantics for Delays*

Only the semantics of the `fby` operator has to be modified. Indeed, we have:

$x\ \texttt{fby}\ y = \texttt{fby1}\ x\ y$

$\texttt{fby1}\ ((\perp,t).xs)\ ((\perp,t).ys) = (\perp,t).(\texttt{fby1}\ xs\ ys)$
$\texttt{fby1}\ ((\perp,f).xs)\ ((\perp,f).ys) = (\perp,f).(\texttt{fby1}\ xs\ ys)$
$\texttt{fby1}\ ((x,t).xs)\ ((y,t).ys) = (x,t).(\texttt{fby2}\ y\ xs\ ys)$
$\texttt{fby1}\ ((x,f).xs)\ ((y,f).ys) = (x,f).(\texttt{fby2}\ y\ xs\ ys)$

$\texttt{fby2}\ z\ ((\perp,f).xs)\ ((\perp,f).ys) = (\perp,f).(\texttt{fby2}\ z\ xs\ ys)$
$\texttt{fby2}\ z\ ((\perp,t).xs)\ ((\perp,t).ys) = (\perp,t).(\texttt{fby1}\ xs\ ys)$
$\texttt{fby2}\ z\ ((x,f).xs)\ ((y,f).ys) = (z,f).(\texttt{fby2}\ y\ xs\ ys)$
$\texttt{fby2}\ z\ ((x,t).xs)\ ((y,t).ys) = (x,t).(\texttt{fby2}\ y\ xs\ ys)$

`fby` has two different states. While x and y are absent, it emits nothing and stays in its current state (`fby1`). When x is present, it emits x and goes in the state (`fby2`). In this state, it emits its second argument while the reset condition is false and it goes back to its initial state every time the reset condition is true.

As before, this semantics is partial and can be complemented, using the `fail` value.

4.3.2 *The New Semantics for Clocks*

We must be very careful when giving the semantics of `on` and `every`. Indeed, what happens when we reset a stream which is absent? In the same way, what does it mean to sample a stream which is reset?

The semantics of (cl `on` c) becomes:

$((f,b).cls)\ \texttt{on}\ ((\perp,b).cs) = (f,b).(cls\ \texttt{on}\ cs)$
$((t,b).cls)\ \texttt{on}\ ((f,b).cs) = (f,b).(cls\ \texttt{on}\ cs)$
$((t,b).cls)\ \texttt{on}\ ((t,b).cs) = (t,b).(cls\ \texttt{on}\ cs)$

Thus, the semantics of `on` does not take the reset information into account. An expression with clock (cl `on` c) is present when cl is true and c is present and true.

The semantics of cl `every` c becomes:

$((f,b).cls)\ \texttt{every}\ ((\perp,b).cs) = (f,b).(cls\ \texttt{every}\ cs)$
$((t,t).cls)\ \texttt{every}\ ((b,t).cs) = (t,t).(cls\ \texttt{every}\ cs)$
$((t,f).cls)\ \texttt{every}\ ((b,f).cs) = (t,b).(cls\ \texttt{every}\ cs)$

Conversely , the reset information is propagated even when the clock is false.

4.3.3 *New Clock Rule*

Clocks are enriched with a new construction, dealing with the reset information. We introduce the clock (cl `every` e). Clocks are now:

$$cl \quad ::= \quad \ldots \mid cl\ \texttt{every}\ e$$

The clock of a reset is:

$H \vdash \texttt{reset} : \forall\alpha.(c:\alpha) \rightarrow (\alpha\ \texttt{every}\ c \rightarrow \alpha\ \texttt{every}\ c) \rightarrow \alpha \rightarrow \alpha$

(`reset` $c\ f\ e$) restarts the execution of ($f\ e$) every time c is true. It means that every stream depending on the input or output of f is reset every time c is true. Thus, (`reset` $c\ f\ e$) can be understood as a special form of function application. After the clock calculus is performed, every operation is annotated with its clock following the same treatment as with regular clocks [5, 6].

4.4 Properties

We show that adding this construction does not break the main principles of the basic kernel.

Property 1 (Preservation of clocks). *For any expression e_1 and e_2 such that $H \vdash e_1 : cl$ and $e_1 \rightarrow_\beta e_2$ then $H \vdash e_2 : cl$.*

Proof. The proof is based on the following lemma. Detailed proofs are given in the appendix. □

Lemma 1 (Substitution). *If $H, x : cl_1 \vdash e_2 : cl_2$ where $x \notin fv(H)$ and $H \vdash e_1 : cl_1$ then $H \vdash e_2[e_1/x] : cl_2[e_1/x]$.*

Property 2 (Correctness). *Well clocked programs can be synchronously executed.*

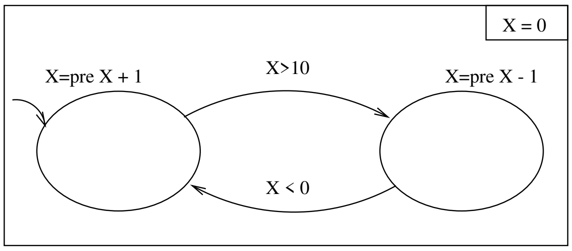
Proof. The proof is similar to the one given in [6]. □

5. APPLICATION: ENCODING MODE-AUTOMATA

In the examples we have considered, the reset operator has been used for restarting a behavior. We think that this operator can be of much more general use, and in particular can be used to define reconfigurable systems. As an example of a real use, we have developed an automatic translation of Mode-Automata [14, 15] in Lucid Synchrone.

5.1 Mode-Automata

Mode-automata has been proposed recently for dealing with running modes in the synchronous language Lustre. On one hand, data-flow languages appear as the best choice for describing sampled continuous systems and on the other hand, imperative languages allow a very simple description of control-based systems. Hybrid systems, like sequences of continuous behaviors, would require an hybrid approach. Mode-automata propose such an approach, providing an automaton formalism to express the control part of the system, the continuous part being written in Lustre. Consider, for example, a two-state mode-automaton:



This automaton defines a stream `X`, with the initial value 0. The equation defining `X` is given by the current state, which is called a *mode*. Initially, `X` is defined as `X = pre X + 1` ($mode_1$). When a transition occurs, the equation defining `X` is modified. If `X` is greater than 10, it becomes `X = pre X - 1` ($mode_2$). More complex Mode-automata are obtained by composition. The parallel composition allows for combining several automata working synchronously. The hierarchical composition allows for a mode, or a part of a node to be defined as an automaton itself.

An important remark about Mode-automata is that only the computations belonging to the running mode must be executed during an instant. Compilation following this property and producing sequential code have been proposed.

5.2 Mode-Automata in Lucid-Synchrone

The modes by themselves, being LUSTRE programs, will find a direct expression in LUCID SYNCHRONE. The difficulty in encoding mode-automata in the language comes from the control part of the system, that is, the automaton structure. It would be natural to define this structure using mutually recursive functions. However, as said before, recursive functions are forbidden in order to guarantee reactivity. Here, we'll use the reset to express this recursive behavior.

A mode is seen as a function taking as arguments the inputs of the system and the initial values of the variables. It returns the outputs of the mode and the mode to execute on the next instant (determined by the transition condition). For the simple example presented above, we obtain the following definitions [6]. `mode_init` defines the initial mode and modes are encoded by integers (e.g, `1` stands for `mode_1`).

```
let mode_1 x_init = (x, n_mode) where
  rec x = (x_init -> pre x) + 1
  and n_mode = if x > 10 then 2 else 0

let mode_2 x_init = (x, n_mode) where
  rec x = (x_init -> pre x) - 1
  and n_mode = if x < 0 then 1 else 0

let mode_init x_init = (x_init, 1)
```

Then we define a branching function choosing at every instant which mode to execute. The function takes the input of the system, the mode to execute and applies the mode to the input. For this purpose, we define the *activation condition* `cond_act` of a function. The activation condition takes a default value `default`, a Boolean condition `clk`, a function `f` and an input `input` and computes `f input` only when `clk` is true. Otherwise, it returns the previously emitted value:

```
let cond_act default clk f input = o where
  rec o = merge clk (f (input when clk))
                    ((default fby o) when not clk)
```

Using this operator, the branching function becomes:

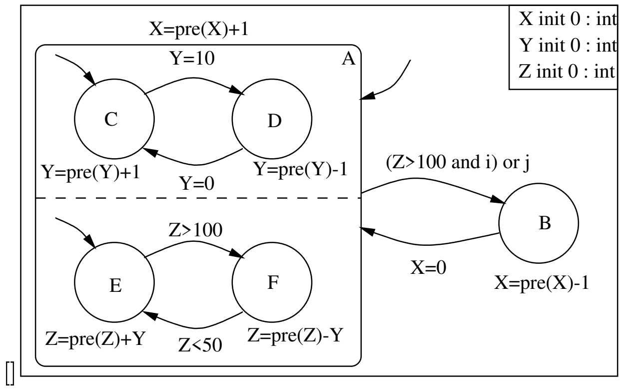
```
let run (x,mode) =
  let rec cur_mode = if mode = 0 then (0 -> pre m)
                     else mode in
  if cur_mode = 0 then
    cond_act (x,0) (cur_mode = 0) mode_init x
  else if cur_mode = 1 then
    cond_act (x,0) (cur_mode = 1) mode_1 x
  else
    cond_act (x,0) (cur_mode = 2) mode_2 x
```

[6]`->` is the initialization operator and `pre` is the non initialized delay operator. `x -> y = if true fby false then x else y` and `pre x = dummy fby x` where `dummy` is any value with a correct type.

It is important to notice here that the expected behavior of Mode-automata is achieved: thanks to the use of clocks in condition activations, only the computations belonging to the current mode are executed. Now by resetting this branching function each time a transition is taken, we obtain the global system. Modes take their new initial value (the current value of the system) when a reset awakes them. The main function, defining the whole automaton is the following:

```
let automat () =
  let rec (x,mode) =
    let imode = 1 -> pre mode in
    reset (false -> ((1 -> imode) <> 0))
          run ((0 -> pre x), (1 -> imode))
    in x
```

This translation of a simple automaton is quite straightforward. We may now consider the case of more complex Mode-automata which are obtained using parallel and hierarchical compositions. Consider for example, the following automaton:



where `Y` and `Z` in mode `A` are given by an internal automaton (hierarchical composition). This automaton being defined itself as the parallel composition of two smaller automata.

Parallel composition consists in calling each automaton in order to define the corresponding variables. Hierarchical composition appears in the definition of the external mode as a function call to the internal automaton.

Following the previous example, we can encode the automata defining `Y` and `Z` as two functions. The initial value and the initial modes are taken as arguments of the functions. `Y` is also an argument of the function defining `Z`, as it appears in the definition of `Z`. Now the parallel (synchronous) composition of these two automata is:

```
let par (y_init,z_init) mode_init = (y,z) where
  rec y = auto_y y_init mode_init
  and z = auto_z (z_init,y) mode_init
```

The function `par` defines the automaton created by the parallel composition of the two simpler automata. Thanks to higher-order, this parallel composition may even be more general, by abstracting over the modes `auto_y` and `auto_z`.

In order to encode the complete automaton, we need to express the mode `A` which contains itself the previous automaton. Again, this expression is straightforward, we follow the previous encoding. `Y` and `Z` are only defined by a call to the function `par`:

```
let mode_a ((i,j),(y_init,z_init,x_init)) =
  let rec (y,z) = par (y_init,z_init) 1
  and x = ((x_init -> pre x) + 1)
  in ((y,z,x),
      if (((z > 100) & i) or j) then 3 else 0)
```

The rest of the translation follows the first example.

5.3 Relevance of the Encoding

The encoding presented above describes the principles for a translation of Mode-automata into LUCID SYNCHRONE. An automatic translator has been written following this encoding.

The obtained code seems to be more complex than the initial automaton. Such an encoding has several properties.

- It is modular: every node is encoded as a function and we define a branching function using the reset operator and selecting the mode to execute.
- It is correct with respect to the expected behavior of Mode-automata: only the computations in the current mode are computed at a time.
- It is compatible with the whole language (e.g, higher-order, clocks). In particular, each node may be defined by a function (e.g. an equation like `x = f(y)`).

The main weakness of this encoding comes here from the obligation to explicitly pass the current value of each defined variable from mode to mode. This is because we try to express a system using shared memory in a purely functional model (here, the current value of each variable is shared among modes).

6. RELATED WORKS

This work is naturally related to several works about the combination of formalisms in synchronous languages [10, 16]. However, our approach differs in the sense that we do not want to mix several formalisms but, instead, to extend in a conservative way, the data-flow paradigm with some constructions allowing the direct expression of reconfigurable systems.

Sequential constructions on data-flows were proposed in SIGNAL [17]. The notion of intervals proposed in this work correspond to the clock `every` proposed here. Our contribution is the analysis of the scope of the construction, and the use of the clock calculus in order to ensure this scope. Our reset construction is thus fully compatible with the whole language.

7. CONCLUSION

In this paper, we have presented the addition of a reset construct to LUCID SYNCHRONE, a synchronous data-flow language. This is a strict extension of the language and all properties have been preserved. This has been obtained by an analysis of the scope of the reset operator. The addition of this operator is done by a simple extension of the clock calculus.

This operator allows the definition of reconfigurable systems, the recursive behavior of such systems could be simulated with the reset. As an example of such definitions, we have presented an encoding of Mode-automata. A tool has been developed, translating automatically Mode-automata into valid LUCID SYNCHRONE programs.

Future works include two fields of research. First, a better exploration of the expressive power of the reset, and its use in the definition of reconfigurable systems could be investigated. The work on Mode-Automata is a good starting point for such an experimentation. The second extension of this work is to understand recursive functions in a synchronous data-flow language and in particular tail-recursion.

8. ACKNOWLEDGEMENTS

We would like to thank Florence Maraninchi and Yann Rémond for their help on Mode-Automata and Thérèse Hardin for her numerous suggestions. Anonymous reviews also significantly contributed in improving the paper.

9. REFERENCES

[1] C. J. A. Benveniste, P. LeGuernic. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[2] T. Amagbegnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[3] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* Academic Press, 1985.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 96:87–152, 1992.

[5] P. Caspi and M. Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Workshop on Coalgebraic Methods in Computer Science (CMCS'98)*, ENTCS, 28-29 March 1998. Available as a VERIMAG tech. report no. 97–07.

[6] P. Caspi and M. Pouzet. Lucid synchrone, a functional extension of lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[7] 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.

[8] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[9] S. P. Jones and J. H. (editors). Report on the programming language haskell 98. Technical report, `http://haskell.org`, 1999.

[10] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *IEEE International Conference on Computer Languages (ICCL), Toulouse, France)*, 1994.

[11] X. Leroy. *The Objective Caml System - release 2.02.* INRIA, 1999.

[12] J. R. Lewis, M. B. Shields, E. Meijer, and J. Launchburry. Implicit parameters: Dynamic scoping with static types. In *International Conference*

on Principles of Programming Languages (POPL). ACM, 2000.

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

[14] F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages. In *International Symposium: Compositionality - The Significant Difference*, Malente (Holstein, Germany), Sept. 1997. Springer Verlag.

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

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

[17] E. Rutten and P. L. Guernic. Sequencing data flow tasks in Signal. In *ACM Sigplan Workshop on Language, Compiler and Tool support for real-time systems*, Orlando, 1994.

APPENDIX

LEMMA 2 (SUBSTITUTION). *If $H, x : cl_1 \vdash e_2 : cl_2$ where $x \notin fv(H)$ and $H \vdash e_1 : cl_1$ then $H \vdash e_2[e_1/x] : cl_2[e_1/x]$.*

PROOF. We prove the following stronger property.

If $H, x_1 : cl_1, H_r \vdash e : cl$ where $H_r = [x_2 : cl_2, ..., x_n : cl_n]$ such that for all i, $x_i \notin fv(H, x_1 : cl_1, ..., x_{i-1} : cl_{i-1})$ and $H \vdash e_1 : cl_1$ then $H, H_r[e_1/x_1] \vdash e[e_1/x_1] : cl[e_1/x_1]$.

The proof is done by structural induction on e.

Variables ($e = x_1$): Thus $cl = cl_1$ and $x_1[e_1/x_1] = e_1$. Then $cl_1[e_1/x_1] = cl_1$.

Variables ($e = x_i$ **with** $x_i : cl_i \in H_r$): $x_i[e_1/x_1] = x_i$. If $H, x_1 : cl_1, ..., x_i : cl_i, ... \vdash x_i : cl_i$ then $H, ..., x_i : cl_i[e_1/x], ... \vdash x_i : cl_i[e_1/x_1]$.

Variables ($e = x_i$ **with** $x_i : cl_i \in H$): $x_i[e_1/x_1] = x_i$. Then $cl_i[e_1/x_1] = cl_i$ since $x_1 \notin fv(H)$.

Synchronous primitives ($e = \texttt{extend}\ e_3\ e_4$)**:** The clock calculus derivation of e has the form:

$$\frac{H, x_1 : cl_1, H_r \vdash e_3 : cl \quad H, x_1 : cl_1, H_r \vdash e_4 : cl}{H, x_1 : cl_1, H_r \vdash \texttt{extend}\ e_3\ e_4 : cl}$$

Using the induction hypothesis on e_3 and e_4, we have the following derivation:

$$\frac{\begin{array}{c} H, H_r[e_1/x_1] \vdash e_3[e_1/x_1] : cl[e_1/x_1] \\ H, H_r[e_1/x_1] \vdash e_4[e_1/x_1] : cl[e_1/x_1] \end{array}}{H, H_r[e_1/x_1] \vdash \texttt{extend}\ e_3[e_1/x_1]\ e_4[e_1/x_1] : cl[e_1/x_1]}$$

When ($e = e_3\ \texttt{when}\ e_4$)**:** The clock calculus derivation of e is:

$$\frac{H, x_1 : cl_1, H_r \vdash e_3 : cl \qquad H, x_1 : cl_1, H_r \vdash e_4 : cl}{H, x_1 : cl_1, H_r \vdash e_3\ \texttt{when}\ e_4 : cl\ \ \texttt{on}\ e_4}$$

Using the induction hypothesis on e_3 and e_4, we now have the following derivation:

$$\frac{\begin{array}{c} H, H_r[e_1/x_1] \vdash e_3[e_1/x_1] : cl[e_1/x_1] \\ H, H_r[e_1/x_1] \vdash e_4[e_1/x_1] : cl[e_1/x_1] \end{array}}{H, H_r \vdash\ e_3[e_1/x_1]\ \texttt{when}\ e_4[e_1/x_1] : cl[e_1/x_1]\ \ \texttt{on}\ e_4[e_1/x_1]}$$

The proof for `merge` is similar.

Reset ($e = \texttt{reset}\ e_3\ e_4\ e_5$)**:** The clock derivation is:

$$\frac{\begin{array}{c} H, x_1 : cl_1, H_r \vdash e_3 : cl \quad H, x_1 : cl_1, H_r \vdash e_5 : cl \\ H, x_1 : cl_1, H_r \vdash e_4 : cl\ \texttt{every}\ e_3 \to cl\ \ \texttt{every}\ e_3 \end{array}}{H, x_1 : cl_1, H_r \vdash \texttt{reset}\ e_3\ e_4\ e_5 : cl}$$

Using the induction hypothesis, we obtain:

$$\frac{\begin{array}{c} H, H_r[e_1/x_1] \vdash e_3[e_1/x_1] : cl[e_1/x_1] \\ H, H_r[e_1/x_1] \vdash e_5[e_1/x_1] : cl[e_1/x_1] \\ H, H_r[e_1/x_1] \vdash e_4[e_1/x_1] :\ \ cl[e_1/x_1]\ \ \texttt{every}\ e_3[e_1/x_1] \\ \to cl[e_1/x_1]\ \ \texttt{every}\ e_3[e_1/x_1] \end{array}}{H \vdash \texttt{reset}\ e_3[e_1/x_1]\ e_4[e_1/x_1]\ e_5[e_1/x_1] : cl[e_1/x_1]}$$

Abstraction ($e = \lambda y.e_3$)**:** We only consider the case where $y \notin fv(e_1)$. Otherwise, y may be renamed with a fresh variable. The clock calculus derivation is:

$$\frac{H, x_1 : cl_1, H_r, y : cl \vdash e_3 : cl_3 \qquad y \notin fv(H, x_1 : cl_1, H_r)}{H, x : cl_1, H_r \vdash\ \lambda y.e_3 : (y : cl) \to cl_3}$$

Using the induction hypothesis on e_3, we obtain the derivation:

$$\frac{\begin{array}{c} H, H_r[e_1/x_1], y : cl[e_1/x_1] \vdash e_3[e_1/x_1] : cl_3[e_1/x_1] \\ y \notin fv(H, H_r[e_1/x_1]) \end{array}}{H, H_r[e_1/x_1] \vdash\ \lambda y.e_3[e_1/x_1] : (y : cl[e_1/x_1]) \to cl_3[e_1/x_1]}$$

Since $((y : cl) \to cl_3)[e_1/x_1] = (y : cl[e_1/x_1]) \to cl_3[e_1/x_1]$.

Application ($e = e_3\ e_4$)**:** We have the following derivation:

$$\frac{H, x_1 : cl_1, H_r \vdash e_3\ : (y : cl) \to cl_2 \quad H, x_1 : cl_1, H_r \vdash e_4\ : cl}{H, x_1 : cl_1, H_r \vdash e_3\ e_4 : cl_2[e_4/y]}$$

Using the induction hypothesis on e_3 and e_4, we can find the derivation:

$$\frac{\begin{array}{c} H, H_r[e_1/x_1] \vdash e_3[e_1/x_1]\ : (y : cl[e_1/x_1]) \to cl_2[e_1/x_1] \\ H, H_r[e_1/x_1] \vdash e_4[e_1/x_1]\ : cl[e_1/x_1] \end{array}}{H, H_r[e_1/x_1] \vdash e_3[e_1/x_1]\ e_4[e_1/x_1] : cl_2[e_1/x_1][e_4[e_1/x_1]/y]}$$

But $cl_2[e_1/x_1][e_4[e_1/x_1]/y] = cl_2[e_4/y][e_1/x_1]$ since $y \notin fv(e_1)$.

Recursion ($e = rec\ y.\ e_3$)**:** The derivation has the following form:

$$\frac{H, x : cl_1, H_r, y\ : cl_2 \vdash e_3\ : cl_2 \qquad y \notin fv(H, x : cl_1)}{H, x : cl_1, H_r \vdash rec\ y.e_3\ : cl_2}$$

Using the induction hypothesis on e_3, we have:

$$\frac{\begin{array}{c} H, H_r[e_1/x_1], y : cl_2[e_1/x_1] \vdash e_3[e_1/x_1]\ : cl_2[e_1/x_1] \\ y \notin fv(H, H_r[e_1/x_1]) \end{array}}{H, H_r[e_1/x_1] \vdash rec\ y.e_3[e_1/x_1]\ : cl_2[e_1/x_1]}$$

The substitution lemma is a particular case of this property where H_r is empty. □

Property 3 (Preservation of clocks). *For any expression e_1 and e_2 such that $H \vdash e_1 : cl$ and $e_1 \rightarrow_\beta e_2$ then $H \vdash e_2 : cl$.*

Proof. The proof is by induction on the structure of e_1. Two interesting case are detailed below.

When $(e_1 = e_3 \texttt{ when } e_4)$**:** The clock calculus derivation of e_1 is:

$$\frac{H \vdash e_3 : cl_4 \qquad H \vdash e_4 : cl_4}{H \vdash e_3 \texttt{ when } e_4 : cl_4 \ \texttt{ on } e_4}$$

If $e_4 \rightarrow_\beta e'_4$, $H \vdash e'_4 : cl_4$ by induction. Thus, $H \vdash e_3 \texttt{ when } e'_4 : cl \texttt{ on } e'_4$. But $cl \texttt{ on } e_4 = cl \texttt{ on } e'_4$ since equality between clocks is considered modulo β-reduction. If $e_3 \rightarrow_\beta e'_3$ then, by induction $H \vdash e'_3 : cl_4$, thus $H \vdash e'_3 \texttt{ when } e_4 : cl_4$.

Application $(e_1 = (\lambda x.e_3)\ e_4)$**:** Applying the clock rules, we have $H, x : cl_4 \vdash e_3 : cl_3$ and $H \vdash e_4 : cl_4$. If $(\lambda x.e_3)\ e_4 \rightarrow_\beta e_3[e_4/x]$, then applying the lemma, $H \vdash e_3[e_4/x] : cl_3[e_4/x]$ give the expected result. If $e_4 \rightarrow_\beta e'_4$ then $H \vdash (\lambda x.e_3)\ e'_4 : cl_3[e'_4/x]$. $cl_3[e'_4/x] = cl_3[e_4/x]$ since equality between clocks is considered modulo β-reduction. $\square$