Time Refinement in a Functional Synchronous Language

Louis Mandel
LRI, Université Paris-Sud 11, Orsay, France
INRIA Paris-Rocquencourt, France
louis.mandel@lri.fr

Cédric Pasteur Marc Pouzet
DI, École normale supérieure, Paris, France
INRIA Paris-Rocquencourt, France
firstname.lastname@ens.fr

ABSTRACT

Concurrent and reactive systems often exhibit multiple time scales. For instance, in a discrete simulation, the scale at which agents communicate might be very different from the scale used to model the internals of each agent.

We propose an extension of the synchronous model of concurrency, called *reactive domains*, to simplify the programming of such systems. Reactive domains allow the creation of local time scales and enable *refinement*, that is, the replacement of an approximation of a system with a more detailed version without changing its behavior as observed by the rest of the program.

Our work is applied to the ReactiveML language, which extends ML with synchronous language constructs. We present an operational semantics for the extended language and a type system that ensures the soundness of programs.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; D.3.4 [**Processors**]: Compilers

General Terms

Languages, Theory

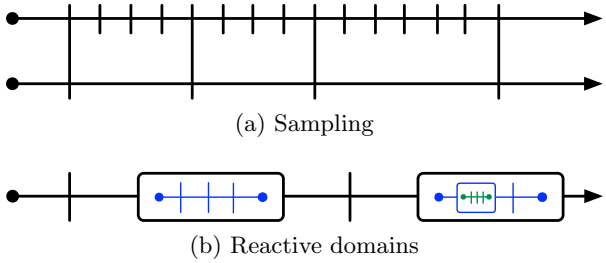
Keywords

Synchronous languages; Functional languages; Semantics; Type systems

1. INTRODUCTION

The concept of discrete logical time greatly simplifies the programming of *concurrent* and *reactive* systems. It is the basis of synchronous languages [2], which were created for programming reactive embedded systems that continuously interact with their environments, such as control command.

PPDP '13, September 16 - 18 2013, Madrid, Spain



(a) Sampling

(b) Reactive domains

Figure 1: Sampling vs. Reactive domains (each vertical line or box represent one instant of the corresponding clock, horizontal lines represent processes running in parallel)

The idea is to divide an execution into logical instants, where communications and computations are assumed to be instantaneous. It gives a deterministic model of concurrency that can be compiled to sequential imperative code. The synchronous model can also be used to program discrete simulations of systems. It is natural to divide the execution of a simulation into time steps, each corresponding to one instant of the program. A simulation then comprises multiple agents running in parallel and synchronizing on the logical time scale.

Suppose, for example, that we want to simulate the power consumption in a sensor network [18]. In order to have a precise estimate of the power consumption, we need to simulate the hardware of some nodes, in particular the radio. There are now multiple time scales: the scale of the software (i.e., MAC protocol) is milliseconds, while the time step of the hardware would be microseconds. The communication between these time scales must be restricted: a signal that varies each microsecond cannot be used to communicate with a process whose rhythm is in milliseconds. Furthermore, depending on the level of precision required for the simulation, we may like to be able to replace a precise but costly version of one agent, that takes multiple steps to be simulated, with an instantaneous approximation. This is traditionally called *temporal refinement* [14]. It should be possible to change this precision dynamically and without changing the external behavior of the system with respect to any other agent.

A traditional solution to this problem in synchronous languages is to use *sampling*: a new time scale is obtained by choosing a subset of instants. In this paper, we propose an extension of the synchronous model of concurrency called *reactive domains*, that allows doing the opposite. Instead of

creating a new time scale by going slower, it creates a faster time scale by subdividing instants. This is done by creating local instants that are unobservable from the outside, as shown in Figure 1. Reactive domains make refinement easy as they allow hiding local computation steps (Section 3).

Our work is applied to the ReactiveML language [13], which augments ML with a synchronous model of concurrency (Section 2).[1] We show how to extend the operational semantics of the language to incorporate reactive domains (Section 4). The soundness of programs in the extended setting can be checked using a standard type-and-effect system, called a *clock calculus* and reminiscent of the one in dataflow synchronous languages [2] (Section 5). The article ends with a discussion of the implementation, several extensions (Section 6) and related work (Section 7).

2. THE REACTIVEML LANGUAGE

ReactiveML[2] [13] is based on the *synchronous reactive* model of concurrency, which first appeared in the ReactiveC language [3]. It is an extension of the synchronous model that enables dynamic creation and which extends the focus of synchronous languages from programming real-time embedded systems to the problem of dealing with concurrency in a general purpose language. ReactiveML applies the same basic idea to ML, with which it shares many features, like higher-order functions and type inference, and formalisms, like its semantics and type system.

2.1 Examples

ReactiveML is a reactive extension of ML, so any ML program is also a valid ReactiveML program. For instance, we can define a tree data type and the preorder iteration of a function on a tree by:

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree

let rec preorder f t = match t with
| Empty -> ()
| Node(l, v, r) ->
   f v; preorder f l; preorder f r
```

The concrete syntax of the language is the one of OCaml, upon which ReactiveML is built. The type of trees, `'a tree`, is parametrized by the type `'a` of its labels. A tree is either empty, or made of a left child, a label and a right child. The preorder traversal of the tree is implemented with a simple recursive function that applies a given function to the label and recurses first on the left child and then on the right one. We can almost as easily define the level-order traversal of the tree in ReactiveML:

```
let rec process levelorder f t = match t with
| Empty -> ()
| Node (l, v, r) ->
    f v; pause;
    (run levelorder f l || run levelorder f r)
```

This example defines a recursive *process* named `levelorder`. Unlike regular ML functions which are instantaneous, a process can last several instants. In particular, `levelorder` awaits the next instant by using the `pause` operator and then recursively calls itself on the left and right children in parallel. The `||` operator denotes logical parallelism, that is compiled to sequential code with cooperative scheduling. The `run` operator is used to launch a process. As all processes share the same notion of instant, these two processes synchronize on the next `pause`, which means that `f` will be applied to all the labels at the same depth during a given instant. It should be noted that the order in which processes running in parallel are executed is unspecified.

Processes running in parallel can communicate using *broadcast signals*. A signal is a *stream* of values, that is, a sequence of values indexed by the instants. Processes have a consistent view of a signal's status during an instant: either present or absent and henceforth unable to change for the rest of the instant. Running the following process prints `"Hello world"` at the first instant, as the first branch of the parallel reacts immediately to the presence of the signal `go`:

```
let process hello_world =
 signal go in
 await immediate go; print_string "Hello world"
 ||
 emit go
```

A signal can also carry a value. Several processes can emit different values on a signal during the same instant, which is termed *multi-emission*. These values are combined using a function given in the definition of the signal. The value of a signal at a given instant is obtained by folding this function across emitted values, starting from a default value. In the following example, the value of the signal `s` is the sum of emitted values:

```
let process sig_gather =
  signal s default 0 gather (+) in
  emit s 2 || emit s 4
  || await s(v) in print_int v
```

This process prints `6` (i.e. `0 + 2 + 4`), but on the second instant. Indeed, we want to ensure that all processes read the same value for each signal. This means that when we try to read the value of a signal, we have to be sure that no other value will be emitted later in the instant. The easiest way to enforce this is to wait for the end of the instant and to only read the value at the next instant. One can react immediately to the presence of a signal, as in the previous example, but it takes one instant to read its value. Finally, a signal also stores its last value, that can be used for instance to maintain its value across instants:

```
let process hold s =
  loop emit s (last s); pause end
```

2.2 Programming Agents in ReactiveML

Figure 2a shows an example of a node in a simulation of a sensor network, that is, a network of small low-cost sensors that collect and communicate environmental data. It receives messages on the signal `me` (line 10), decrements them and then forwards them to all of its neighbors (line 6) (`iter` iterates a process on all the elements of a list). The second part of the node (lines 13 to 18) models energy consumption: the energy of the node is decremented by `max_power` at each time step, that corresponds to one millisecond of simulation time. The node terminates when its energy crosses the `e_min` threshold. This is achieved by using preemption through the `do/until` control structure. Indeed, `do e until dead done` executes the body `e` until the emission of the signal `dead`, then terminates in the next instant.

[1]The compiler, the extended version of the paper and the examples mentioned in the paper are available at:
`http://reactiveml.org/ppdp13`

[2]`http://www.reactiveml.org`

```
let process node me neighbors =
  signal dead in
  signal energy default e_0 gather (fun x _ -> x) in
  let process send msg n = emit n msg in
  let process forward_msg msg =
    if msg>1 then run iter (send (msg-1)) neighbors
  in
  do
    loop (* protocol *)
      await me(msgs) in run iter forward_msg msgs
    end
    ||
    loop (* power *)
      if last energy < e_min
      then emit dead
      else emit energy (last energy -. max_power);
      pause
    end
  until dead done
```

(a) A simple node in a sensor network

```
let dt = 0.01
signal env default (fun _ -> zero_vector)
           gather add_force

let rec process body (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f) in
  (* euler semi-implicit method *)
  let v_tp = v_t ++. (dt **. (f x_t)) in
  let x_tp = x_t ++. (dt **. v_tp) in
  run body (x_tp, v_tp, w)

let process main =
  for i = 1 to 100 dopar
    run body (random_planet ())
  done
```

(b) The n-body problem (++. and **. are operations on vectors)

Figure 2: Two simple examples

Another simple example of simulation is the n-body problem, solved using a fixed-step numerical integration in Figure 2b. The idea is to use a global signal `env`, whose value is a force field, that is, a function mapping a position to a force. Each body, characterized by its current position, velocity and weight, is a process that, at each instant, sends its attraction by emitting on `env`, receives the sum of all the forces emitted by other bodies and uses this force to compute its position `dt` later. The main process is made of several bodies run in parallel using a parallel for loop.

3. REACTIVE DOMAINS

A reactive domain introduces a local notion of instant, unobservable from the outside. It can also be seen as a reification of the execution engine attached to any ReactiveML program. It means that a domain behaves as if its body was executed by a separate execution engine, with its own notion of step. A reactive domain is in charge of dynamically scheduling processes and, in particular, deciding when a local instant is over. It also manages signals, for instance storing their last value or activating processes awaiting their emission.

3.1 Reactive Domains and Clocks

A reactive domain is declared by the keyword `domain`:

```
domain ck do e done
```

The name `ck` is the identifier of the domain, that we call a *clock*. It is bound only in the body `e` of the domain. The clock represents the notion of instant attached to the domain. That is why the `pause` operator now takes as argument a clock: `pause ck` waits for the next instant of the domain of clock `ck`. For instance, the following process prints `"Hello "` during the first instant of the clock `ck` and `"world"` during the second instant of `ck`.

```
let process hello_world_ck =
  domain ck do
    print_string "Hello "; pause ck;
    print_string " world"
  done
```

Both instants of `ck` are included in the first instant of the global clock, called `global_ck`.[3] These local instants can only be observed by processes inside the reactive domain, so `hello_world_ck` is equivalent to:

```
let process hello_world_seq =
  print_string "Hello ";
  print_string " world"
```

as if the synchronization on the local clock `ck` was erased.

Reactive domains form a tree, called the *clock tree*, where one reactive domain is a child of another if it is defined in the latter's scope. We will say that a clock `ck'` is faster than `ck` if `ck'` is a descendant of `ck` in the clock tree. The global clock `global_ck` is slower than any other clock, that is, it is the root of the clock tree.

While the `hello_world_ck` process terminates instantaneously, it is possible for the execution of a reactive domain to span several instants of its parent domain. It is then necessary to relate the instants of the reactive domain to those of its parent, that is, to know how many steps of the reactive domain should be taken in each step of the parent reactive domain. The simplest way is to create a periodic reactive domain, that performs n local instants per instant of its parent reactive domain, using the keyword `by`:

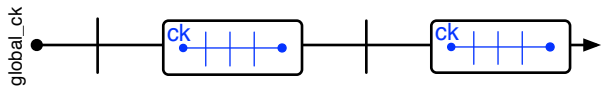
```
let process stutter msg =
  domain ck by 6 do
    loop print_string msg; pause ck end
  done
```

The expression `run stutter "a"` prints six `a`'s at each instant of the global clock. Instants of sibling reactive domains are unrelated. For instance, `run stutter "a" || run stutter "b"` prints six `a`'s and six `b`'s at each instant in an unspecified order.

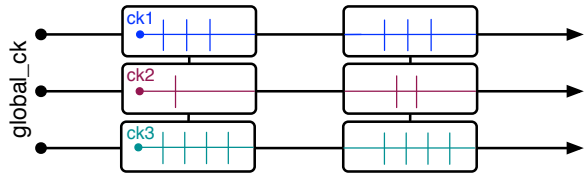
Reactive domains can be created dynamically and nested arbitrarily. For instance, the `stutter` process can be rewritten as follows:

```
let process stutter_nested msg =
  domain ck1 by 3 do
    domain ck2 by 2 do
      loop print_string msg; pause ck2 end
    done
  done
```

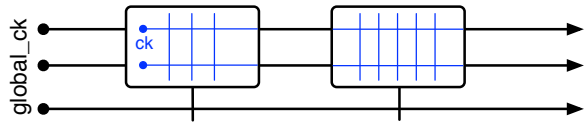
[3] `global_ck` is a global variable which is the global clock of the program.



(a) Make a process instantaneous



(b) Agents: one reactive domain per agent



(c) Hide computation steps (multi-steps integration)

Figure 3: Several patterns of programming with reactive domains

3.2 Reactive Domains and Signals

The consequence of introducing reactive domains is that each signal is now attached to a reactive domain, that is, it has one value for each instant of this domain. This explains why we use the term clock for a domain's identifier. Indeed, in synchronous languages, a clock is a boolean stream that indicates the instants when a stream is present [2]. The semantics of signals defined inside a reactive domain is exactly the same as before. For instance, if we run one of the examples of Section 2.1 inside a reactive domain, the result is the same: `sig_gather_ck` prints `6` during the second instant of `ck`, but during the first instant of the global clock:

```
let process sig_gather_ck =
  domain ck do run sig_gather done
```

Emitting a value on a signal with a slower clock (that we will call a slow signal) is not an issue thanks to multi-emission: all the values emitted during the instant of the signal's clock, including the multiple instants of child reactive domains, are gathered to compute the value for the instant. It is also possible to await the emission of a slow signal. The continuation will occur in the next instant of the emitted signal's clock, as in the following process:

```
let process slow_signal =
  signal s default 0 gather (+) in
  domain ck by 3 do
    await s(v) in print_int v
  done
  ||
  emit s 4
```

A `4` is printed during the second instant of the global clock, as if there were no reactive domain, but during the fourth instant of clock `ck`. The result would have been the same if the `emit` statement had been inside the reactive domain. Sibling reactive domains can thus communicate using signals attached to a common ancestor in the clock tree.

While in ReactiveML, any process can use any signal freely, reactive domains introduce some restrictions. First, as the instants of a reactive domain are unobservable from the outside, it does not make sense to access a signal attached to a reactive domain from outside that domain where the different values of the signal cannot be distinguished. The second restriction is that it is forbidden to react immediately to the presence of a slow signal. Let's illustrate the problem with the following process:

```
let process immediate_dep_wrong =
  signal s in
  domain ck do
    await immediate s; print_string "Ok"
    ||
    pause ck; emit s
  done
```

During the first instant of `ck`, we suppose that `s` is not present, so the first branch of the parallel is blocked. But in the second instant of `ck` – yet still in the first instant of `global_ck` – `s` is emitted, which should trigger the printing of `"Ok"`. We reject this process because it makes two different assumptions about the presence of `s` during the same instant of the clock of `s`, which goes against the principle that all processes have the same view of a signal's status and value at an instant. The type system defined in Section 5 ensures that these two types of errors never occur.

3.3 Relating Clocks

Let's consider this process:

```
let process delayed_hello_world =
  signal s default "" gather (^) in
  domain ck by 10 do
    pause global_ck; emit s "Hello world"
    ||
    await s(v) in print_string v
  done
```

At the end of the first instant of `ck`, the first branch of the parallel is waiting for the next instant of `global_ck` and the other one is waiting for a signal on `global_ck`. If the reactive domain executed another local instant, its body would not evolve. It is not necessary to do ten local instants: the reactive domain can directly wait for the next instant of `global_ck` before doing its next local instant. We can thus interpret the number given after `by` as a bound on the number of instants that a reactive domain can do.

We could just treat this property as a run-time optimization, but we believe it can be usefully incorporated in the semantics of the language so as to accept more programs. Indeed, in most cases, as in the previous example, it is clear that the body of the reactive domain will be blocked waiting for a slower clock at some point. It is thus permitted to omit the bound (as was done in the first examples). A reactive domain then not only decides when its local instants are finished, but also when to wait for the next instant of its parent clock. It does so automatically if all the processes it contains are waiting for the next instant of a slower clock, either via an explicit `pause` or by waiting for a signal with a slower clock.

Special care has to be taken not to produce a reactive domain that is not reactive, that is, that never waits for the next instant of its parent clock and behaves like an infinite loop. This will be discussed in Section 6.3.

3.4 Using Reactive Domains

Reactive domains are useful for several typical patterns. The first is to make a process instantaneous. For instance, one can hide the internal steps used in the `levelorder` example of Section 2.1 (`pause` without any argument waits for the

```
let process node_with_energy me neighbors =
 domain us by 1000 do
  signal dead in
  signal energy default e_0 gather (fun x _ -> x) in
  signal power default 0.0 gather (+.) in
  signal r_in default (0,me) gather (fun x _ -> x) in
  signal r_ack in
  let process send msg n =
    emit r_in (msg, n); await immediate r_ack
  in
  ...
  do
    ... (* protocol *) ||
    loop (* radio *)
      await r_in (msg, n) in
      for i=1 to packet_send_time do
        emit power send_power; pause us
      done;
      emit n msg; emit r_ack
    end
    ||
    loop  (* power *)
      emit power on_power;
      if last energy < e_min
      then emit dead
      else emit energy
          (last energy -. (last power /. 1000.0));
      pause us
    end
  until dead done
 done
```

(a) A node with refined power consumption

```
let rec process body_heun env (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f_t) in
  (* step 1 *)
  let f_t = f_t x_t in
  let v_int = v_t ++. (dt **. f_t) in
  let x_int = x_t ++. (dt **. v_t) in
  (* step 2 *)
  emit env (force (x_int, w));
  await env(f_int) in
  let f_int = f_int x_int in
  let v_tp = v_t ++.
      ((dt /. 2.0) **. (f_t ++. f_int)) in
  let x_tp = x_t ++.
      ((dt /. 2.0) **. (v_t ++. v_int)) in
  (* next step*)
  pause global_ck;
  run body_heun env (x_tp, v_tp, w)

let process main =
  domain computation_ck do
    signal env default (fun _ -> zero_vector)
                gather add_force in
    for i = 1 to 100 dopar
      run body_heun env (random_planet ())
    done
  done
```

(b) Multi-step integration method (Heun's method)

Figure 4: Two examples with reactive domains

next instant of the local clock, that can also be obtained using the `local_ck` operator):

```
let process levelorder_inst f t =
  domain ck do
    run levelorder f t
  done
```

Figure 3a illustrates the behavior of the reactive domain: it hides all internal steps and behaves as an instantaneous process on the global clock. This process could not have been written without the automatic waiting of reactive domains (Section 3.3) as it executes an unbounded number of local instants, equal to the tree depth.

The second pattern is for programming agent-based simulations. Reactive domains allow each agent to perform an arbitrary number of internal steps during each step of the simulation, that corresponds to one instant of the global clock. One simply has to declare one reactive domain per agent, as in Figure 3b. Agents only synchronize at the end of the instant of the global clock. Signals for communication between agents remain attached to the global reactive domain, and are thus buffered automatically.

We can use this idea to better simulate the power consumption of the node from Figure 2a, by modeling the fact that power consumption is related to the number of messages sent. An abbreviated version of the resulting program is shown in Figure 4a. The idea is to use a reactive domain to introduce a new local time scale, corresponding to microseconds of simulation time. The radio is represented by a process receiving a message to be sent and a destination on the `r_in` signal. The sending of the message is modeled by waiting `packet_send_time` microseconds, during which the power consumption is raised by `send_power`. After that, the radio actually sends the message to the destination and acknowledges the sending on the `r_ack` signal.

A similar use is to hide computation steps shared by many agents. The fast clock is then shared by several processes as in Figure 3c, whereas in Figure 3b each process has its own local clock. An example of this pattern is an extension of the n-body simulation of Figure 2b to use multi-steps integration methods, here Heun's method. The resulting code is shown in Figure 4b. Each step of the computation corresponds to one instant of a reactive domain, shared by all bodies. As these instants are unobservable from the outside, it is easy to add processes such as the GUI on the global clock (last line in Figure 3c) or to dynamically switch methods (e.g. from a two-steps to a four-steps method) without any influence on the rest of the program.

3.5 A Modularity Issue

We have seen that some communications take time because of multi-emission. This can lead to modularity problems, as we will see on a few examples, and makes it even more necessary to be able to hide local instants. Let's first define a higher-order process `lift` that turns a function on values into a function on streams (like the `arr` combinator in FRP [15]). It awaits a new value on a signal `s_in`, applies `f` to it and emits the result on another signal `s_out`:

```
let process lift f s_in s_out =
  loop
    await s_in(v) in
    emit s_out (f v)
  end
```

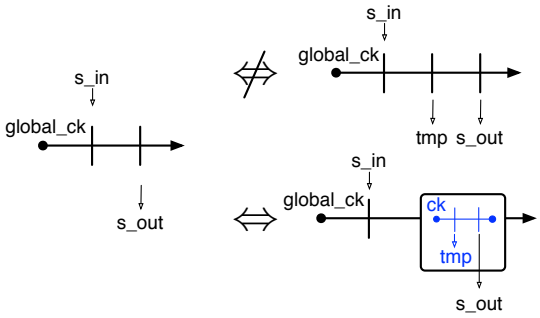


Figure 5: Fixing a modularity problem with reactive domains (left: `fg1`, top-right: `fg2`, bottom-right: `fg2_good`)

We can now define a process `fg1` that applies the composition of two functions `g` and `f`:

```
let process fg1 s_in s_out =
  run lift (fun v -> f (g v)) s_in s_out
```

Suppose that, for modularity reasons, we want to separate the computations of `f` and `g`. We use a local signal `tmp` to communicate between the two processes:

```
let process fg2 s_in s_out =
  signal tmp default 0 gather (+) in
  run lift f s_in tmp || run lift g tmp s_out
```

The problem is that, while `fg1` emits the result one instant after the emission of a value on `s_in`, it takes two instants for `fg2` to do the same. We can fix this problem by running the process inside a reactive domain:

```
let process fg2_good s_in s_out =
  domain ck do
    run fg2 s_in s_out
  done
```

The `fg2_good` process has the same behavior as the `fg1` process: it takes two instants of the local clock `ck` to compute the result, but only one on the global clock. Figure 5 illustrates the behavior of these three processes.

4. OPERATIONAL SEMANTICS

In this section, we extend the ReactiveML operational semantics [13] to support reactive domains. It is itself an extension of the small-step reduction semantics of ML.

4.1 Language Abstract Syntax

We present the semantics on a core language, based on a call-by-value functional kernel extended with synchronous primitives: defining and running a process, waiting for the next instant of a clock, a parallel `let`, declaring a signal, emitting a value, awaiting its emission or getting its last value, preemption (`until`) and suspension (`when`) control structures, declaring a reactive domain and accessing the local clock (`local_ck`):

$$
\begin{aligned}
e ::= {} & x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \\
& \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause}\ e \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \\
& \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \mid \texttt{emit}\ e\ e \\
& \mid \texttt{await}\ e(x)\ \texttt{in}\ e \mid \texttt{last}\ e \mid \texttt{do}\ e\ \texttt{until}\ e \mid \texttt{do}\ e\ \texttt{when}\ e \\
& \mid \texttt{domain}\ x\ \texttt{by}\ e\ \texttt{do}\ e \mid e\ \texttt{in}\ ck \mid \texttt{local\_ck}
\end{aligned}
$$

The expression `do` e `when` s executes its body only when s is present; e `in` ck is used to represent a reactive domain executing. It represents the result of instantiating the expression `domain` x `by` e `do` e and cannot itself be used directly in a program. We denote by _ variables that do not appear free in the body of a `let` and by () the unique value of type `unit`. Among others, it is possible to derive the following constructs from this kernel:

$$
\begin{aligned}
\texttt{await immediate}\ e &\triangleq \texttt{do}\ ()\ \texttt{when}\ e \\
e_1 \,||\, e_2 &\triangleq \texttt{let}\ \_ = e_1\ \texttt{and}\ \_ = e_2\ \texttt{in}\ () \\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 &\triangleq \texttt{let}\ x = e_1\ \texttt{and}\ \_ = ()\ \texttt{in}\ e_2 \\
e_1; e_2 &\triangleq \texttt{let}\ \_ = e_1\ \texttt{in}\ e_2 \\
\texttt{domain}\ x\ \texttt{do}\ e &\triangleq \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e \\
\texttt{loop}\ e &\triangleq \texttt{run}\ ((\texttt{rec}\ loop = \\
&\qquad \lambda x.\texttt{process}\ (\texttt{run}\ x; \texttt{run}\ (loop\ x)))\ (\texttt{process}\ e)) \\
\texttt{signal}\ s\ \texttt{in}\ e &\triangleq \texttt{signal}\ s\ \texttt{default}\ [] \\
&\qquad\qquad \texttt{gather}\ (\lambda x.\lambda y.x :: y)\ \texttt{in}\ e \\
\texttt{emit}\ e &\triangleq \texttt{emit}\ e\ () \\
\texttt{pause} &\triangleq \texttt{pause local\_ck}
\end{aligned}
$$

4.2 Notations

$\mathcal{C}$ is a denumerable set of clock names, denoted ck. The global clock is denoted $\top_{ck} \in \mathcal{C}$. $\mathcal{N}$ is a denumerable set of signal names, denoted n. Values are the regular ML values, plus processes, signal names indexed by their clock and clock names:

$$v ::= c \mid (v,v) \mid \lambda x.e \mid \texttt{process}\ e \mid n^{ck} \mid ck \qquad \text{(values)}$$

A *local signal environment* is a partial mapping from signal names to tuples (d, g, l, m) where d and g are the default value and gather function, l the last value and m the multiset of values emitted at an instant. A *signal environment* $\mathcal{S}$ is a partial mapping from clock names to local signal environments. If $\mathcal{S}(n^{ck}) = \mathcal{S}(ck)(n) = (d, g, l, m)$, we write $\mathcal{S}^d(n^{ck}) = d$ (similarly for the others) and $\mathcal{S}^v(n^{ck}) = \text{fold}\ g\ d\ m$ if $m \neq \emptyset$ ($\mathcal{S}^v(n^{ck})$ is not defined otherwise). We write $n^{ck} \in \mathcal{S}$ when n is present, that is, $\mathcal{S}^m(n^{ck}) \neq \emptyset$, and $n^{ck} \notin \mathcal{S}$ otherwise. We denote by $S + [v/n^{ck}]$ the environment where v is added to the multiset $\mathcal{S}^m(n^{ck})$ and by $\texttt{next}(\mathcal{S}, ck)$ the environment where the last value of any signal with clock ck is set to its current value $\mathcal{S}^v(n^{ck})$ (if defined) and $\mathcal{S}^m(n^{ck})$ is set to $\emptyset$.

Similarly, a *clock environment* $\mathcal{H}$ maps clock names ck to tuples (pck, r, m), where pck is the parent clock of ck and r (resp. m) tracks the number of steps remaining (resp. the maximum number of steps) in the current instant of the parent clock ($r, m \in \mathbb{N} \cup \{\infty\}$). The same notation is used to refer to the individual fields (for instance $\mathcal{H}^r(ck)$). We denote by $\mathcal{H}[ck \leftarrow i]$ the environment where $\mathcal{H}^r(ck)$ is set to i.

A clock environment induces a partial order $\preceq_{\mathcal{H}}$, which is the smallest reflexive, transitive and antisymmetric relation such that $ck \preceq_{\mathcal{H}} \mathcal{H}^{pck}(ck)$. Intuitively, $ck_F \preceq_{\mathcal{H}} ck_S$ means that ck_S is slower than ck_F. We write $ck \preceq_{\mathcal{H}} C$ iff $\forall ck' \in C.\, ck \preceq_{\mathcal{H}} ck'$. $C^{\uparrow\mathcal{H}}$ denotes the upward closure of C, that is:

$$C^{\uparrow\mathcal{H}} = \{ck' \mid \exists ck \in C.\, ck \preceq_{\mathcal{H}} ck'\}$$

4.3 Semantics

We define two reductions: the *step reduction*, denoted $\xrightarrow{ck}$, and the *end-of-instant reduction* $\xrightarrow[\text{eoi}]{C}$. The step reduction is parametrized by the local clock ck. The execution of a

$$\lambda x.e\ v/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow v]/\mathcal{H},\mathcal{S} \qquad \texttt{rec}\ x = e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow \texttt{rec}\ x = e]/\mathcal{H},\mathcal{S} \qquad \texttt{run}\ (\texttt{process}\ e)/\mathcal{H},\mathcal{S} \xrightarrow{ck} e/\mathcal{H},\mathcal{S}$$

$$\texttt{let}\ x_1 = v_1\ \texttt{and}\ x_2 = v_2\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x_1 \leftarrow v_1; x_2 \leftarrow v_2]/\mathcal{H},\mathcal{S}$$

$$\frac{n \notin Dom(\mathcal{S}(ck)) \qquad \mathcal{S}' = \mathcal{S}(ck)[n \mapsto (v_d, v_g, v_d, \emptyset)]}{\texttt{signal}\ x\ \texttt{default}\ v_d\ \texttt{gather}\ v_g\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x/n^{ck}]/\mathcal{H},\mathcal{S}'} \qquad \frac{ck \preceq_{\mathcal{H}} ck'}{\texttt{emit}\ n^{ck'}\ v/\mathcal{H},\mathcal{S} \xrightarrow{ck} ()/\mathcal{H},\mathcal{S} + [v/n^{ck'}]}$$

$$\frac{ck \preceq_{\mathcal{H}} ck'}{\texttt{last}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} \mathcal{S}^l(n^{ck'})/\mathcal{H},\mathcal{S}} \qquad \frac{ck \preceq_{\mathcal{H}} ck'}{\texttt{do}\ v\ \texttt{until}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} ()/\mathcal{H},\mathcal{S}} \qquad \frac{ck = ck' \qquad n^{ck'} \in \mathcal{S}}{\texttt{do}\ v\ \texttt{when}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} v/\mathcal{H},\mathcal{S}}$$

$$(\textsc{Inst})\ \frac{i > 0 \qquad ck' \notin Dom(\mathcal{H}) \qquad \mathcal{H}' = \mathcal{H}[ck' \mapsto (ck, i-1, i-1)] \qquad \mathcal{S}' = \mathcal{S}[ck' \mapsto []]}{\texttt{domain}\ x\ \texttt{by}\ i\ \texttt{do}\ e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow ck']\ \texttt{in}\ ck'/\mathcal{H}',\mathcal{S}'}$$

$$(\textsc{Term})\ v\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} v/\mathcal{H},\mathcal{S} \qquad \texttt{local\_ck}/\mathcal{H},\mathcal{S} \xrightarrow{ck} ck/\mathcal{H},\mathcal{S}$$

$$(\textsc{Context})\ \frac{e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'/\mathcal{H}',\mathcal{S}'}{\Gamma(e)/\mathcal{H},\mathcal{S} \xrightarrow{ck} \Gamma(e')/\mathcal{H}',\mathcal{S}'} \qquad (\textsc{When})\ \frac{ck = ck' \qquad n^{ck'} \in \mathcal{S} \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'/\mathcal{H}',\mathcal{S}'}{\texttt{do}\ e\ \texttt{when}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} \texttt{do}\ e'\ \texttt{when}\ n^{ck'}/\mathcal{H}',\mathcal{S}'}$$

$$(\textsc{Step})\ \frac{e/\mathcal{H},\mathcal{S} \xrightarrow{ck'} e'/\mathcal{H}',\mathcal{S}'}{e\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'\ \texttt{in}\ ck'/\mathcal{H}',\mathcal{S}'} \qquad (\textsc{LocalEoi})\ \frac{\begin{array}{c}\mathcal{H}^r(ck') > 0 \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{\{ck'\}}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}' \qquad e' \neq e \\ \mathcal{H}'' = \mathcal{H}'[ck' \leftarrow \mathcal{H}^r(ck') - 1] \qquad \mathcal{S}'' = \texttt{next}(\mathcal{S}', ck')\end{array}}{e\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'\ \texttt{in}\ ck'/\mathcal{H}'',\mathcal{S}''}$$

Figure 6: The step reduction

$$\frac{ck \in C}{\texttt{pause}\,ck/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} ()/\mathcal{H},\mathcal{S}} \qquad \frac{e_1/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e_1'/\mathcal{H}',\mathcal{S}' \qquad e_2/\mathcal{H}',\mathcal{S}' \xrightarrow{C}_{\text{eoi}} e_2'/\mathcal{H}'',\mathcal{S}''}{\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e/\mathcal{H}'',\mathcal{S}''}$$

$$\frac{ck \in C \qquad n^{ck} \in \mathcal{S}}{\texttt{await}\ n^{ck}(x)\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e[x \leftarrow \mathcal{S}^v(n^{ck})]/\mathcal{H},\mathcal{S}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad n^{ck} \in \mathcal{S} \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}'}{\texttt{do}\ e\ \texttt{when}\ n^{ck}/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} \texttt{do}\ e'\ \texttt{when}\ n^{ck}/\mathcal{H}',\mathcal{S}'}$$

$$\frac{ck \in C \qquad n^{ck} \in \mathcal{S} \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}'}{\texttt{do}\ e\ \texttt{until}\ n^{ck}/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} ()/\mathcal{H},\mathcal{S}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad ck \notin C \vee n^{ck} \notin \mathcal{S} \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}'}{\texttt{do}\ e\ \texttt{until}\ n^{ck}/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} \texttt{do}\ e'\ \texttt{until}\ n^{ck}/\mathcal{H}',\mathcal{S}'}$$

$$(\textsc{StuckDomain})\ \frac{e/\mathcal{H},\mathcal{S} \overset{C \cup \{ck'\}}{\hookrightarrow}_{\text{eoi}}}{e\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}} \qquad (\textsc{ParentEoi})\ \frac{\begin{array}{c}e/\mathcal{H},\mathcal{S} \xrightarrow{C \cup \{ck'\}}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}' \qquad e' \neq e \\ \mathcal{H}'' = \mathcal{H}'[ck' \leftarrow \mathcal{H}^m(ck')] \qquad \mathcal{S}'' = \texttt{next}(\mathcal{S}', ck')\end{array}}{e\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e'\ \texttt{in}\ ck'/\mathcal{H}'',\mathcal{S}''}$$

$$v/\mathcal{H},\mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad ck \notin C}{\texttt{pause}\,ck/\mathcal{H},\mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad ck \notin C \vee n^{ck} \notin \mathcal{S}}{\texttt{await}\ n^{ck}(x)\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad n^{ck} \notin \mathcal{S}}{\texttt{do}\ e\ \texttt{when}\ n^{ck}/\mathcal{H},\mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}}$$

Figure 7: The end-of instant reduction

reactive domain consists in applying the step reduction with the local clock as many times as possible, to get a so-called *end-of-instant expression*. Then, the end-of-instant reduction prepares the execution of the next instant of the domain.

A program is executed inside the global reactive domain of clock $\top_{ck}$. This means that the semantics of a program p is given by the reduction of the expression $\tilde{p}$ defined by:

$$\tilde{p} \triangleq \texttt{let } \mathit{global\_ck} = \texttt{local\_ck in } p$$

A program step, denoted $\Rightarrow$, is made of many step reductions followed by one end-of-instant reduction in the local clock $\top_{ck}$:

$$\frac{e/\mathcal{H},\mathcal{S} \xrightarrow{\top_{ck}}{}^{*} e'/\mathcal{H}',\mathcal{S}' \qquad e'/\mathcal{H}',\mathcal{S}' \xrightarrow{\{\top_{ck}\}}_{\text{eoi}} e''/\mathcal{H}'',\mathcal{S}''}{e/\mathcal{H},\mathcal{S} \Rightarrow e''/\mathcal{H}'',\mathcal{S}''}$$

The reduction starts from $e_0 = \tilde{p}$ and the initial clock and signal environments are both empty: $\mathcal{H}_0 \triangleq []$ and $\mathcal{S}_0 \triangleq []$.

Step reduction. The step reduction is expressed as:

$$e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'/\mathcal{H}',\mathcal{S}'$$

meaning that under the local clock ck, the expression e reduces to e' and transforms the clock and signal environments $\mathcal{H}$ and $\mathcal{S}$ into $\mathcal{H}'$ and $\mathcal{S}'$. The rules are given in Figure 6, where the basic rules are adapted directly from ReactiveML [13] and new rules are introduced for executing reactive domains:

- A reactive domain is initialized by first evaluating the bound on the number of steps, initializing the clock environment and instantiating the clock variable with a fresh clock (Inst rule).
- Then, local reduction steps (Step rule) are applied while possible. If the body is reduced to a value, the reactive domain terminates (Term rule), returning that value. Otherwise, a new local instant is started if the steps remaining counter has not reached zero and work remains to be done in the next local step (LocalEoi rule). Indeed, if the end-of-instant relation leaves the body unchanged (here $e = e'$), doing more local steps would not change anything, as the body is already stuck w.r.t. the step reduction. The reactive domain is then stuck waiting for the end-of-instant of its parent reactive domain, as explained in Section 3.3.
- A signal can only be accessed if its clock ck' is accessible, that is, if ck' is slower than or equal to the local clock ck, denoted $ck \preceq_{\mathcal{H}} ck'$ in the rules.
- The Context rule applies a head reduction in any valid evaluation context Γ, defined by:

$$\begin{aligned}\Gamma ::= {} & [] \mid \Gamma\ e \mid e\ \Gamma \mid (\Gamma, e) \mid (e, \Gamma) \mid \texttt{run } \Gamma \mid \texttt{pause } \Gamma \\ & \mid \texttt{let } x = \Gamma \texttt{ and } x = e \texttt{ in } e \\ & \mid \texttt{let } x = e \texttt{ and } x = \Gamma \texttt{ in } e \\ & \mid \texttt{signal } x \texttt{ default } \Gamma \texttt{ gather } e \texttt{ in } e \\ & \mid \texttt{signal } x \texttt{ default } e \texttt{ gather } \Gamma \texttt{ in } e \\ & \mid \texttt{emit } \Gamma\ e \mid \texttt{emit } e\ \Gamma \mid \texttt{await } \Gamma(x) \texttt{ in } e \mid \texttt{last } \Gamma \\ & \mid \texttt{do } \Gamma \texttt{ until } e \mid \texttt{do } e \texttt{ until } \Gamma \\ & \mid \texttt{do } e \texttt{ when } \Gamma \mid \texttt{domain } x \texttt{ by } \Gamma \texttt{ do } e\end{aligned}$$

- We need to add a special rule for `do` e `when` n, as its body is an evaluation context only if the signal n is present. The clock of the signal must also be equal to the local clock as suspension represents an immediate dependency.

End-of-instant reduction. The end-of-instant reduction is expressed as:

$$e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}'$$

meaning that during the end-of-instant of the clocks in the set C, e reduces to e' and transforms the clock and signal environments $\mathcal{H}$ and $\mathcal{S}$ into $\mathcal{H}'$ and $\mathcal{S}'$. We also write:

$$e/\mathcal{H},\mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}} \Leftrightarrow\ e/\mathcal{H},\mathcal{S} \xrightarrow{C}_{\text{eoi}} e/\mathcal{H},\mathcal{S}$$

The rules are given in Figure 7. As for the step reduction, the basic rules are the same as in regular ReactiveML. The novelties are as follows:

- In several cases, we require the clock of signals to be in the upward closure of C, denoted $C^{\uparrow\mathcal{H}}$, which is basically the set of accessible clocks. The relation is not defined if we try to access a clock that is not in this set.
- Expressions that await a signal only reduce during the end-of-instant of the signal clock.
- There are two cases for reactive domains: either the body is left untouched by the end-of-instant relation (rule StuckDomain), which means it is waiting for a slower clock and will remain stuck until the end-of-instant of that clock; or it reduces to a new expression, in which case we modify the clock and signal environments to prepare for the next instant (ParentEoi).

5. CLOCK CALCULUS

Reactive domains induce restrictions on the use of signals. As the local instants of a reactive domain are unobservable from the outside, a signal attached to a reactive domain cannot be used outside of that domain. Immediate dependencies on slow signals are also forbidden.

We want to statically reject programs that could have an incorrect behavior by using a standard type-and-effect system that we call a *clock calculus* in reference to synchronous languages [6]. This ability is one of the benefits of exposing concurrency in the language, as opposed to introducing it through a library. As usual, well-typed programs do not go wrong, which means here that they do not access a signal outside of its domain and do not depend immediately on slow signals.

5.1 Motivation

A first example of the sort of program that we want to reject is one where the result of a reactive domain contains a local signal:

```
let process result_escape =
  domain ck do
    signal s in s
  done
```

Such programs are rejected by including clocks in the type of signals and checking that the return types of reactive domains do not contain local clocks.

$$\frac{ct \leq \Gamma(x)}{\Gamma, ce \vdash x : ct \mid \emptyset} \qquad \frac{ct \leq \Gamma_0(c)}{\Gamma, ce \vdash c : ct \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_1 : ct_1 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash (e_1, e_2) : ct_1 \times ct_2 \mid \emptyset} \qquad \frac{\Gamma; x : ct, ce \vdash e : ct \mid cf}{\Gamma, ce \vdash \texttt{rec}\ x = e : ct \mid cf}$$

$$\frac{\Gamma; x : ct_1, ce \vdash e : ct_2 \mid cf}{\Gamma, ce \vdash \lambda x.e : ct_1 \xrightarrow{cf} ct_2 \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_1 : ct_2 \xrightarrow{cf} ct_1 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash e_1\ e_2 : ct_1 \mid cf}$$

$$\frac{\Gamma, ce \vdash e_1 : ct_1 \mid cf_1 \qquad \Gamma, ce \vdash e_2 : ct_2 \mid cf_2 \qquad \Gamma; x_1 : gen(ct_1, e_1, \Gamma); x_2 : gen(ct_2, e_2, \Gamma), ce \vdash e : ct \mid cf}{\Gamma, ce \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e : ct \mid cf_1 \cup cf_2 \cup cf}$$

$$\text{(ProcAbs)}\ \frac{\Gamma, ce' \vdash e : ct \mid cf}{\Gamma, ce \vdash \texttt{process}\ e : ct\,\texttt{process}\{ce'|cf\} \mid \emptyset} \qquad \text{(ProcApp)}\ \frac{\Gamma, ce \vdash e_1 : ct\,\texttt{process}\{ce|cf\} \mid \emptyset}{\Gamma, ce \vdash \texttt{run}\ e_1 : ct \mid cf}$$

$$\frac{\Gamma, ce \vdash e_1 : ct_2 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_1 \longrightarrow ct_2 \longrightarrow ct_2 \mid \emptyset \qquad \Gamma; x : (ct_1, ct_2)\,\texttt{event}\{ce\}, ce \vdash e : ct \mid cf}{\Gamma, ce \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : ct \mid cf \cup \{ce\}}$$

$$\frac{\Gamma, ce \vdash e_1 : (ct_1, ct_2)\,\texttt{event}\{ce'\} \mid \emptyset \qquad \Gamma; x : ct_2, ce \vdash e_2 : ct \mid cf_2}{\Gamma, ce \vdash \texttt{await}\ e_1(x)\ \texttt{in}\ e_2 : ct \mid cf_2 \cup \{ce'\}} \qquad \frac{\Gamma, ce \vdash e_1 : \texttt{unit} \mid cf_1 \qquad \Gamma, ce \vdash e_2 : (ct_1, ct_2)\,\texttt{event}\{ce'\} \mid \emptyset}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{until}\ e_2 : \texttt{unit} \mid cf_1 \cup \{ce'\}}$$

$$\text{(When)}\ \frac{\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_2 : (ct_1, ct_2)\,\texttt{event}\{ce\} \mid \emptyset}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{when}\ e_2 : ct \mid cf_1 \cup \{ce\}}$$

$$\text{(Domain)}\ \frac{\Gamma; x : \{\gamma\}, \gamma \vdash e : ct \mid cf \qquad \Gamma, ce \vdash e_1 : \texttt{int} \mid \emptyset \qquad \gamma \notin ftv(\Gamma, ct)}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_1\ \texttt{do}\ e : ct \mid cf \setminus \{\gamma\}}$$

$$\text{(In)}\ \frac{\Gamma, ck' \vdash e : ct \mid cf}{\Gamma, ce \vdash e\ \texttt{in}\ ck' : ct \mid cf \setminus \{ck'\}} \qquad \frac{}{\Gamma, ce \vdash \texttt{local\_ck} : \{ce\} \mid \emptyset}$$

Figure 8: Typing rules

Signals are first-class values in the language, which means that a signal can be put inside any data structure or emitted on another signal. The consequence is that a signal can escape its lexical scope and be used anywhere in the program. We also have to make sure to reject programs where a signal escapes its reactive domain through a slow signal, like this one:

```
let process signal_escape =
  signal slow in
  domain ck do
    signal fast default 0 gather (+) in
    emit slow fast
  done
```

To avoid this case, we should also check that the local clock does not appear in the type of free variables when typing a reactive domain. To ensure this, clocks are seen as abstract data types as in [10]. However, signal accesses might not appear in the type of an expression, as in the following example:

```
let process effect_escape =
  domain ck do
    signal fast in
    let f () = emit fast in f
  done
```

The traditional solution to this problem is to associate an expression with both a type and an *effect* [12, 20]. In our case, the effect records the clocks of the signals accessed by the expression.

5.2 Notations

Types are defined by:

$$\begin{array}{lll} ct ::= T \mid \alpha \mid \{ce\} \mid ct \times ct & & \text{(types)} \\ \quad \mid (ct, ct)\,\texttt{event}\{ce\} & & \\ \quad \mid ct \xrightarrow{cf} ct \mid ct\,\texttt{process}\{ce|cf\} & & \\ ce ::= \gamma \mid ck & & \text{(clocks)} \\ cf ::= \phi \mid \emptyset \mid \{ce\} \mid cf \cup cf & & \text{(effects)} \\ cs ::= ct \mid \forall\alpha.\, cs \mid \forall\gamma.\, cs \mid \forall\phi.\, cs & & \text{(type schemes)} \\ \Gamma ::= [x_1 \mapsto cs_1; \ldots; x_p \mapsto cs_p] & & \text{(environment)} \end{array}$$

A type is either a basic type T, a type variable α, a singleton type $\{ce\}$ corresponding to the clock ce, a product, a signal, a function or a process. The type $(ct_1, ct_2)\,\texttt{event}\{ce\}$ of a signal is defined by the type ct_1 of values emitted, the type ct_2 of the received value (and default value) and its clock ce. A clock is either a clock variable or a clock name. An effect cf is attached to functions and processes, and it is a set of clocks or effect variables ϕ. Processes also have an activation clock, which can however be omitted if it does not appear in the return type or the effect of the process. Types schemes generalize over the three kinds of variables. Instantiation and generalization are defined classically by:

$$cs[\alpha \leftarrow ct] \leq \forall\alpha.\, cs \qquad cs[\gamma \leftarrow ce] \leq \forall\gamma.\, cs \qquad cs[\phi \leftarrow cf] \leq \forall\phi.\, cs$$

$$\begin{array}{ll} gen(ct, e, \Gamma) = ct & \text{if } e \text{ is expansive} \\ gen(ct, e, \Gamma) = \forall\bar{\alpha}.\forall\bar{\gamma}.\forall\bar{\phi}.\ ct & \text{otherwise} \\ \quad \text{if } \bar{\alpha}, \bar{\gamma}, \bar{\phi} = ftv(ct) \setminus ftv(\Gamma) & \end{array}$$

where $ftv(ct)$ returns the free type, clock and effect variables in the type ct. As signals are mutable structures, we need to distinguish *expansive* expressions [21] – for which types cannot be generalized.

5.3 Typing Rules

A typing judgment is given by:

$$\Gamma, ce \vdash e : ct \mid cf$$

meaning that under the environment Γ and local clock ce, the expression e has type ct and effect cf.

The initial typing environment Γ_0 contains the signatures of all primitives:

$$\Gamma_0 \triangleq [\, global\_ck : \{\top_{ck}\};\ \texttt{pause} : \forall\gamma.\, \{\gamma\} \xrightarrow{\{\gamma\}} \texttt{unit};$$
$$\texttt{last} : \forall\alpha_1, \alpha_2, \gamma.\, (\alpha_1, \alpha_2)\, \texttt{event}\{\gamma\} \xrightarrow{\{\gamma\}} \alpha_2;$$
$$\texttt{emit} : \forall\alpha_1, \alpha_2, \gamma.\, (\alpha_1, \alpha_2)\, \texttt{event}\{\gamma\} \xrightarrow{\emptyset} \alpha_1 \xrightarrow{\{\gamma\}} \texttt{unit};$$
$$\texttt{true} : \texttt{bool}; \texttt{fst} : \forall\alpha_1, \alpha_2.\, \alpha_1 \times \alpha_2 \xrightarrow{\emptyset} \alpha_1; \ldots]$$

The typing rules are given in Figure 8:

- The rules of the functional kernel are the usual rules in a type-and-effect system. The ProcAbs rule types the body of the process using its activation clock as the new local clock. Only a process on the local clock can be run (rule ProcApp). Often, this is done by instantiating a process whose activation clock is a clock variable.
- In order to forbid immediate dependencies on slow signals, the type system ensures that the clock of a signal is equal to the local clock (evident in the typing judgment as the ce next to the typing environment). See, for instance, the When rule.
- A design choice made in ReactiveML is to separate ML expressions from reactive expressions. For instance, tuples can only contain ML expressions. It is enforced in [13] by a separate syntactic analysis before typing. In the case of our extended type system, we enforce an even stronger separation, by forcing ML expressions to have no effect. This does not reduce expressivity since one can always use a `let` to isolate effectful expressions.
- The most important typing rule is Domain. It checks that the local clock does not escape from its reactive domain. This is done by using a fresh variable for the clock type. The side condition prevents scope extrusion of this fresh name by checking that it does not appear free in the return type ct of the domain nor in the typing environment Γ. It is similar to the typing of `let` in [10] and `let clock` in Lucid Synchrone [6].

5.4 Examples

The `result_escape` process is rejected by the type system because the fresh clock variable associated to `ck` appears in the result type of the reactive domain (denoting $e \triangleq$ `signal` s `in` s):

$$\frac{\Gamma_0; x : \{\gamma\}, \gamma \vdash e : (\alpha, \alpha\,\texttt{list})\,\texttt{event}\{\gamma\} \mid \{\gamma\}}{\Gamma_0, ce \vdash \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e : (\alpha, \alpha\,\texttt{list})\,\texttt{event}\{\gamma\} \mid \emptyset}$$

In the case of the `signal_escape` example, this variable appears in the type of the variable *slow* in the typing environment:

$$slow : ((\texttt{int}, \texttt{int})\,\texttt{event}\{\gamma\}, (\texttt{int}, \texttt{int})\,\texttt{event}\{\gamma\})\,\texttt{event}\{\top_{ck}\}$$

Finally, this variable also appears in the return type for the `effect_escape` process, in the effect of the function:

$$f : \texttt{unit} \xrightarrow{\{\gamma\}} \texttt{unit}$$

5.5 Soundness

We can now state the soundness theorem of the type system, which is the same as that of ML: a well-typed program either reduces to a value or can be reduced infinitely often:

Theorem 5.1 (Soundness of the type system)**.** *If* $\Gamma_0, \top_{ck} \vdash p : ct \mid cf$ *with* $cf \subseteq \{\top_{ck}\}$*, then either :*

- *There exists* $v, \mathcal{H}, \mathcal{S}$ *such that* $\tilde{p}/\mathcal{H}_0, \mathcal{S}_0 \Rightarrow^* v/\mathcal{H}, \mathcal{S}$.
- *For each* $p', \mathcal{H}', \mathcal{S}'$ *such that* $\tilde{p}/\mathcal{H}_0, \mathcal{S}_0 \Rightarrow^* p'/\mathcal{H}', \mathcal{S}'$*, there exists* $p'', \mathcal{H}'', \mathcal{S}''$ *such that* $p'/\mathcal{H}', \mathcal{S}' \Rightarrow p''/\mathcal{H}'', \mathcal{S}''$.

The proof is given in Appendix **??** and uses standard syntactic soundness techniques [16]. Clocks and effects are treated similarly to [5], where the proof is performed on a functional language with regions and references. As there are two reduction relations, we need to prove that an expression that cannot do any step reduction must be able to do an end-of-instant reduction. The novelty compared to the proof of soundness of ReactiveML is that we prove that signals are never accessed unless their clocks are accessible.

6. DISCUSSION

6.1 Implementation

After typing, the ReactiveML compiler generates sequential OCaml code [13], that is linked to an OCaml library providing the ReactiveML runtime, containing mainly the execution engine that schedules the processes. Extending the language with reactive domains is easy because they are a reification of this execution engine. It means that every action associated to reactive domains, like scheduling processes or deciding the end of the instant, is already implemented in the execution engine of ReactiveML. We have only to make this execution engine modular so that multiple reactive domains can be created and nested.

The automatic waiting of the parent clock, described in Section 3.3, is easy to implement. It simply amounts to checking that the local scheduling pool of the reactive domain is empty, that is, that there is nothing left to do in the next step. The main difficulty lies in maintaining the ability to passively wait for signals, that is, a process waiting for a signal should not cost anything in terms of execution time while that signal is absent.

The extension of the type system is also reasonably straightforward. In particular, the side condition of the Domain rule that prevents scope extrusion is easily implemented. Indeed, checking that a variable does not appear free in the typing environment is already necessary when generalizing types in regular ML type inference.

6.2 Signals Clock

So far, all the signals we have used have been attached to the reactive domain at the point of definition. It is often desirable to be able to declare a signal attached to a slower clock than the local one. For example, consider this process that performs a blocking request :

```
let process send_query s =
  signal tmp in
  emit s tmp;
  await tmp(v) in v
```

The process sends the local signal `tmp` on the input signal `s`, and then awaits a reply on `tmp`. It is not possible to use this mechanism to communicate between two sibling reactive domains (e.g. two agents) as the signal `tmp` is attached to the reactive domain of the sender and cannot escape it. This can be solved by allowing a declaration of the clock of a signal:

```
signal tmp clock global_ck in ..
```

The semantics and type system are readily adapted to address this extension. The basic signal declaration can then be interpreted as a declaration on the local clock:

$$\texttt{signal}\ s\ \texttt{in}\ e \triangleq \texttt{signal}\ s\ \texttt{clock local\_ck in}\ e$$

6.3 Clocks and Reactivity

Once we allow reactive domains to perform an unbounded number of instants per instant of the parent clock, it becomes possible for a reactive domain to be non-reactive, that is, to never wait for the next instant of the parent clock, as in the following example:

```
let process nonreactive_domain =
  domain ck do
    loop pause ck end
  done
```

An easy solution would be to always require the programmer to give an explicit bound or to label unbounded reactive domains as an unsafe feature that should only be used by expert programmers. A more ambitious solution would be to design a static analysis to detect when a reactive domain is potentially non-reactive and to warn the programmer. An interesting candidate for this purpose would be a *causal* type-and-effect system in the spirit of [1].

6.4 Clocks and Parallelism

We have decided to forbid immediate dependencies on slower signals, as this violates the assumption that a signal has only one value per instant. This restriction has another benefit: during each instant of its parent reactive domain, a reactive domain can run independently of other processes and reactive domains at the same level. In particular, it can be run in parallel inside another thread of execution and will only synchronize at the end of the instant of its parent reactive domain, when all local instants have been executed. This is not by chance as parallelism was one of the motivations for developing reactive domains. Furthermore, signals declared inside a reactive domain never escape, so they can remain local to the thread and do not require any mechanism (such as locks) to deal with concurrent accesses.

6.5 Limitations of the Type System

The type system as it was presented imposes restrictions when writing combinators, as in this example:

```
let process run_domain q =
  domain ck do run q done
```

This process is rejected because the activation clock of the process `q` is equal to the local clock `ck` at the point where it is run, and by consequence `ck` escapes the scope of its domain.

The source of the problem is that, in ML, the type of a function argument is monomorphic: it is a type ct, not a type scheme cs. In particular, the activation clock of the process `q` cannot be universally quantified, as it is for most processes declared at toplevel. Many solutions to add *higher-rank polymorphism* to ML exist in the literature (see [9, 17] for instance). Most of them require some form of typing annotations from the programmer.

Another important limitation of the type system is that functions or processes stored together must have exactly the same effect. This restriction can be lifted using a simple form of subtyping restricted to effects, often called *subeffecting* [15].

These two extensions are left to future work. As our type system is standard, we believe that it should be possible to adapt existing solutions.

7. RELATED WORK

Our work is related to the *clock refinement* [8] introduced by Gemünde et al. in the synchronous language Quartz [19]. There, the main idea is also to introduce the ability to synchronize on local rhythms. Our work enables more possibilities for communication and synchronization between reactive domains. In particular, Quartz does not permit awaiting the emission of a signal within a domain as it would make it non-reactive: the `delayed_hello_world` example could not be written in Quartz, and this is also true for the sensor node and n-body examples. This greatly reduces the expressivity of the language as this construct is used in almost all programs. We solve this problem by treating each reactive domain as a separate entity that can decide to await the next instant when its body is blocked or after a certain number of local instants. Furthermore, the solution adopted in Quartz cannot be applied to ReactiveML because the latter is a dynamic language: an arbitrary number of processes and signals can be created at runtime using recursion and they can be stored in data-structures and sent via signals (similarly to mobility in the π-calculus). In this context, it is impossible to determine the potential emitters of a signal and thus to decide signal absence, which is a requirement of the clock refinement described in [8].

Other related work includes SugarCubes [4], which shares the same concurrency model as ReactiveML but which uses Java as the base language. It allows the creation of *reactive machines*, the equivalent of our reactive domains, anywhere in the program, but does not offer many ways to communicate and synchronize between machines. It is also up to the programmer to manually schedule the machines, by calling a `react` method as many times as necessary.

In other synchronous languages, time refinement is achieved using *oversampling*, as in Lustre [14] or Signal (see example 4 of [11]). However, oversampling is not as modular as reactive domains: it allows going faster only by slowing down everybody else. The parallel composition of processes that oversample is problematic, especially if each has a different number of internal steps, as in Figure 3b. In Lustre, such programs are rejected by the clock calculus. In Signal, one can specify such behaviors but the compiler is not able to generate sequential code (see the FWS example in [7]). Furthermore, oversampling cannot turn a process taking n instants into an instantaneous one. It can only reduce the delay to one instant. The `levelorder_inst` example shows that this is possible with reactive domains.

8. CONCLUSION

We have presented an extension to the synchronous model of concurrency, called reactive domains, and applied it to the ReactiveML language. It allows the creation of local notions of instant, thereby improving the modularity of the language and facilitating refinement. We have extended the semantics of the language to include this feature and formalized a type system that prevents the unsound use of signals.

The most important future work is to evaluate the usefulness of reactive domains on bigger programs, including the existing sensor network simulations [18]. We are also currently developing a parallel runtime for our extension using system processes communicating via message passing, based on the ideas presented in Section 6.4. Another interesting problem is the one of non-reactive domains discussed in Section 6.3. For instance, it remains an open question whether reactive domains that loops should always be considered as bugs (for instance writing `pause ck` instead of `pause global_ck`) or if there exist useful processes that cannot be written without such looping.

9. ACKNOWLEDGMENTS

We would like to thank Abdoulaye Gamatié for fruitful discussions on the subject and Timothy Bourke for his detailed review.

10. REFERENCES

[1] T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[3] F. Boussinot. Reactive C: an extension of C to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[4] F. Boussinot and J.F. Susini. The SugarCubes tool box: a reactive Java framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.

[5] C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002.

[6] J.L. Colaço and M. Pouzet. Clocks as First Class Abstract Types. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155. Springer Berlin / Heidelberg, 2003.

[7] A. Gamatié and T. Gautier. The signal synchronous multiclock approach to the design of distributed embedded systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(5):641–657, 2010.

[8] M. Gemünde, J. Brandt, and K. Schneider. Clock refinement in imperative synchronous languages. *SYNCHRON*, 9:3–21, 2009.

[9] S.P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

[10] K. Laufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, 1992.

[11] P. Le Guernic, J.P. Talpin, and J.C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(03):261–303, 2003.

[12] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[13] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[14] J. Mikac and P. Caspi. Temporal refinement for Lustre. In *International Workshop on Synchronous Languages, Applications and Programs*, 2005.

[15] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, New York, NY, USA, 2002. ACM.

[16] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[17] D. Rémy. Simple, partial type-inference for System F based on type-containment. In *ACM SIGPLAN Notices*, volume 40, pages 130–143. ACM, 2005.

[18] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. GLONEMO: global and accurate formal models for the analysis of ad-hoc sensor networks. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 3, New York, NY, USA, 2006. ACM.

[19] K. Schneider. *The synchronous programming language Quartz*. Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[20] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162 –173, jun 1992.

[21] M. Tofte. Type inference for polymorphic references. *Information and computation*, 89(1):1–34, 1990.