Time Refinement in a Functional Synchronous Language
Louis Mandel, Cédric Pasteur, Marc Pouzet

To cite this version:

HAL Id: hal-00850290
https://inria.hal.science/hal-00850290
Submitted on 6 Aug 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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.

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). 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 data-flow 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 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:

```ocaml
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:

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

```ocaml
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:

```ocaml
let process sig_gather = signal s default 0 &&
  gather (+) s 2 || emit s 4 || emit s 4
```

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:

```ocaml
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/while` 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.
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 (send (msg-1)) neighbors in
    loop (" protocol ")
    await me(msgs) in run forward_msg msgs end

let rec process body (x_t , v_t , w) =
  emit env (force (x_t , w));
  await env(f) in
    (* power *)
    if last energy < e_min then emit energy (last energy -. max_power);
    pause end
  until dead done

(a) A simple node in a sensor network

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. These local instants can only be observed by processes inside the reactive domain, so hello_world_ck is equivalent to:

let rec process body (x_t , v_t , w) =
  emit env (force (x_t , w));
  await env(f) in
    (* power *)
    if last energy < e_min then emit energy (last energy -. max_power);
    pause end
  until dead done

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

Figure 2: Two simple examples

let dt = 0.01
signal energy 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
    (* power *)
    if last energy < e_min then emit energy (last energy -. max_power);
    pause end
  until dead 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.

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

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
end

3global_ck is a global variable which is the global clock of the program.
Why we use the term clock for a domain's identifier. Indeed, freely, reactive domains introduce some restrictions. First, as sibling reactive domains can thus communicate using signals emit clock instant of clock as if there were no reactive domain, but during the fourth signal's clock, as in the following process:

continuation will occur in the next instant of the emitted signal's clock, as indicated by the multiple instants of child reactive domains, 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 problem we face is the same: sig_gather_ck prints 6 during the second instant of ck, but during the first instant of the global clock:

```plaintext
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: the body of the reactive domain will be blocked waiting 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.

Indeed, in most cases, as in the previous example, it is clear that the body of the reactive domain will be blocked waiting 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 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:

```plaintext
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:

```plaintext
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 (fun x _ -> x) in
  signal r_in default (min, me) gather (fun x _ -> x) in
  signal r_ack in
  let process send_msg n =
    emit r_in (msg, n); wait 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;
              emit n msg; emit r_ack
            done;
        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
  end

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

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

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

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_prj 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 simu-
lations. 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 con-
sumption 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 micro-
seconds 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

let rec process body_heun env (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f_t) in
    (* next step *)
    let f_int = f_int x_int in
    emit env (force (x_int, w));
    await env(f_int) in
    let f_t = f_t x_t in
    let v_int = v_int ++. (dt **. f_t) in
    let x_int = x_int ++. (dt **. v_t) in
    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)
    run add_force in
  for i = 1 to 100 dopar
    body_heun env (random_planet ())
done
done

done

Figure 4: Two examples with reactive domains

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

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

let process levelorder_prj 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 simu-
lations. 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 con-
sumption 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 micro-
seconds 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
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 f and g:

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 (let process (g tmp s_out) |
  run lift f s_in tmp | run lift g tmp s_out

The problem is that, while fg1 emits the result once 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):

\[
e ::= x \mid c \mid (e, e) \mid \lambda x. e \mid e e \mid \text{rec } x = e
\]

\[
| \text{process } e \mid \text{run } e \mid \text{pause } e \mid \text{let } x = e \text{ and } x = e \in e
\]

\[
| \text{signal } x \text{ default } e \text{ gather } e \in e \mid \text{emit } e \mid e
\]

\[
| \text{await } e(x) \text{ in } e \mid \text{last } e \mid \text{do } e \text{ until } e \mid \text{do } e \text{ when } e
\]

\[
| \text{domain } x \text{ by } e \text{ do } e \mid e \text{ in } ck \mid \text{local}_ck
\]

The expression do \(e \text{ when } s\) executes its body only when \(s\) is present; \(e \text{ in } ck\) is used to represent a reactive domain executing. It represents the result of instantiating the expression domain \(x \text{ by } e \text{ do } e\) 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:

\[\text{await immediate } e \triangleq \text{ do } () \text{ when } e\]

\[e_1 \parallel e_2 \triangleq \text{ let } _1 = e_1 \text{ and } _2 = e_2 \text{ in } ()\]

\[\text{let } x = e_1 \text{ in } e_2 \triangleq \text{ let } x = e_1 \text{ and } _2 = () \text{ in } e_2\]

\[e_1; e_2 \triangleq \text{ let } _1 = e_1 \text{ in } e_2\]

\[\text{domain } x \text{ do } e \triangleq \text{ domain } x \text{ by } e \text{ do } e\]

\[\text{loop } e \triangleq \text{ run } ((\text{rec loop } = \lambda x. \text{process } (\text{run } x \cdot \text{run } (\text{loop } x))) \text{ (process }) e)\]

\[\text{signal } s \in e \triangleq \text{ signal } s \text{ default } []\]

\[\text{gather } e \triangleq \text{ do } e \mid \text{emit } e \triangleq \text{ emit } e ()\]

\[\text{pause } \triangleq \text{pause_local}_ck\]

4.2 Notations

\(C\) is a denumerable set of clock names, denoted ck. The global clock is denoted \(\top_{\text{ck}} \in C\). \(N\) 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:

\[v ::= c \mid (v, v) \mid \lambda x. e \mid \text{process } e \mid n^{ck} \mid ck\] (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 \(S\) is a partial mapping from clock names to local signal environments. If \(S(n^{ck}) = S(ck)(n) = (d, g, l, m)\), we write \(S^{d}(n^{ck}) = d\) (similarly for the others) and \(S^{l}(n^{ck}) = \text{fold } g \cdot d \cdot m \cdot m \not\in \emptyset\) \((S^{l}(n^{ck})\) is not defined otherwise). We write \(n^{ck} \in S\) when \(n\) is present, that is, \(S^{n}(n^{ck}) \not\in \emptyset\), and \(n^{ck} \not\in S\) otherwise. We denote by \(S + [v/n^{ck}]\) the environment where \(v\) is added to the multiset \(S^{l}(n^{ck})\) and by \(\text{next}(S, ck)\) the environment where the last value of any signal with clock \(ck\) is set to its current value \(S^{l}(n^{ck})\) (if defined) and \(S^{l}(n^{ck})\) is set to \(\emptyset\).

Similarly, a clock environment \(H\) 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 \((r, m \in N \cup \{\infty\}\)). The same notation is used to refer to the individual fields (for instance \(H^{r}(ck)\)). We denote by \(H[ck \leftarrow i]\) the environment where \(H^{r}(ck)\) is set to \(i\).

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

\[C^{H} = \{ck' | \exists ck \in C. ck \preceq_{H} ck'\}\]

4.3 Semantics

We define two reductions: the step reduction, denoted \(\rightsquigarrow_{\text{ck}}\), and the end-of-instant reduction \(\rightsquigarrow_{\text{ck}}\). The step reduction is parametrized by the local clock ck. The execution of a
\[
\lambda x. e/v, S \xrightarrow{ck} e[x \leftarrow v]/H, S \quad \text{rec } x = e/H, S \xrightarrow{ck} e[x \leftarrow \text{rec } x = e]/H, S \quad \text{run (process } e)/H, S \xrightarrow{ck} e/H, S
\]

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

\[
\begin{array}{l}
\quad n \notin \text{Dom}(S(ck)) \\
\quad \text{signal } x \text{ default } v_d \text{ gather } v_g \text{ in } e/H, S \xrightarrow{ck} e[x/n^{ck'}]/H, S' \\
\quad \text{emitt } n^{ck'} v/H, S \xrightarrow{ck} ()/H, S + [v/n^{ck'}]
\end{array}
\]

\[
\begin{array}{l}
\quad \text{last } n^{ck'}/H, S \xrightarrow{ck} S'(n^{ck'})/H, S \\
\quad \text{ck } \xrightarrow{H} \text{ck'} \\
\quad \text{ck } \xrightarrow{H} \text{ck'} \\
\quad \text{ck } = \text{ck'} \\
\quad \text{do } v \text{ until } n^{ck'}/H, S \xrightarrow{ck} ()/H, S \\
\quad \text{do } v \text{ when } n^{ck'}/H, S \xrightarrow{ck} v/H, S
\end{array}
\]

\[
\begin{array}{l}
\quad \text{do } x \text{ by } i \text{ do } e/H, S \xrightarrow{ck} e[x \leftarrow \text{ck'}] \text{ in } ck'/H', S'
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(TERM) } v \in \text{ck'}/H, S \xrightarrow{ck} v/H, S \\
\quad \text{local}\_ck/H, S \xrightarrow{ck} \text{ck}/H, S
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(INST) } i > 0 \quad \text{ck'} \notin \text{Dom}(H) \\
\quad \text{'H' } = H[[ck'] \rightarrow (ck, i - 1, i - 1)] \\
\quad \text{'S' } = S[[ck'] \rightarrow []]
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(CONTEXT) } e/H, S \xrightarrow{ck} e'/H', S' \\
\quad \Gamma(e)/H, S \xrightarrow{ck} \Gamma(e')/H', S'
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(WHEN) } e/H, S \xrightarrow{ck} e'/H', S' \\
\quad \text{do } e \text{ when } n^{ck'}/H, S \xrightarrow{ck} e' \text{ when } n^{ck'}/H', S'
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(STEP) } e/H, S \xrightarrow{ck} e'/H', S' \\
\quad \text{e in } ck'/H, S \xrightarrow{ck} e' \text{ in } ck'/H', S'
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(LOCALEOI) } \text{H'}(ck') > 0 \\
\quad \text{e}/H, S \xrightarrow{ck} e'/H', S' \\
\quad \text{e' } \neq \text{e} \\
\quad \text{H'' = H'}[ck' \leftarrow H''(ck') - 1] \\
\quad \text{S'' = next(S', ck')}
\end{array}
\]

\[
\begin{array}{l}
\quad \text{e in } ck'/H, S \xrightarrow{ck} e' \text{ in } ck'/H', S'
\end{array}
\]

Figure 6: The step reduction

\[
\begin{array}{l}
\quad \text{ck } \in C \\
\quad \text{pause } ck/H, S \xrightarrow{C_{\text{pa}}}(c)/H, S
\end{array}
\]

\[
\begin{array}{l}
\quad \text{let } x_1 = e_1 \text{ and } x_2 = e_2 \text{ in } e/H, S \xrightarrow{C_{\text{on}}} e'_1/H', S' \\
\quad \text{let } x_1 = e'_1 \text{ and } x_2 = e'_2 \text{ in } e/H', S''
\end{array}
\]

\[
\begin{array}{l}
\quad \text{ck } \in C/H \quad \text{n}^{ck} \in S \\
\quad \text{await } n^{ck}(x) \text{ in } e/H, S \xrightarrow{C_{\text{on}}} e[x \leftarrow S''(n^{ck})]/H, S
\end{array}
\]

\[
\begin{array}{l}
\quad \text{ck } \in C/H \quad \text{n}^{ck} \in S \\
\quad \text{do } e \text{ until } n^{ck}/H, S \xrightarrow{C_{\text{on}}} ()/H, S
\end{array}
\]

\[
\begin{array}{l}
\quad \text{ck } \in C/H \quad \text{ck } \notin C \cup \text{n}^{ck} \notin S \\
\quad \text{e}/H, S \xrightarrow{C_{\text{on}}} e'/H', S' \\
\quad \text{e } \notin e \\
\quad \text{H'' = H'}[ck' \leftarrow H''(ck') - 1] \\
\quad \text{S'' = next(S', ck')}
\end{array}
\]

\[
\begin{array}{l}
\quad \text{e in } ck'/H, S \xrightarrow{C_{\text{on}}} e' \text{ in } ck'/H', S'
\end{array}
\]

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 \( \Upsilon_{ck} \). This means that the semantics of a program \( p \) is given by the reduction of the expression \( \tilde{p} \) defined by:

\[ \tilde{p} \triangleq \text{let global}_ck = \text{local}_ck \text{ in } p \]

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

\[ e/\mathcal{H}, \mathcal{S} \xrightarrow{c} e'/\mathcal{H}', \mathcal{S}' \]

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 applied 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 (\text{INST} rule).
- Then, local reduction steps (\text{Step} rule) are applied while possible. If the body is reduced to a value, the reactive domain terminates (\text{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 (\text{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 \( ck \leq_{\mathcal{H}} ck' \) in the rules.

- The \text{CONTEXT} rule applies a head reduction in any valid evaluation context \( \Gamma \), defined by:

\[
\Gamma ::= [] | \Gamma e | e \Gamma | (\Gamma, e) | (e, \Gamma) | \text{run } \Gamma | \text{pause } \Gamma
\]

- We need to add a special rule for \( \text{do } e \text{ when } n \), 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:

\[ e/\mathcal{H}, \mathcal{S} \xrightarrow{\text{eoi}} e'/\mathcal{H}', \mathcal{S}' \]

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:

\[ e/\mathcal{H}, \mathcal{S} \xrightarrow{\text{eoi}} e/\mathcal{H}', \mathcal{S}' \]

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 \( C^\uparrow \), 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 \text{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 (\text{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:

\[
\text{let process result_escape = do domain } ck \text{ do signal } s \text{ in } s \text{ done}
\]

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.
We also have to make sure to reject programs where a signal appears in the type of an expression, as in the following example:

\[
\begin{align*}
\text{let process signal_escape} &= \\
&\text{domain slow in} \\
&\text{signal fast default } \emptyset \text{ gather (+) in} \\
&\text{emit slow fast} \\
\text{done}
\end{align*}
\]

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:

\[
\begin{align*}
\text{let process effect_escape} &= \\
&\text{domain ck do} \\
&\text{signal fast in} \\
&\text{let } f () = \text{emit fast in } f \\
\text{done}
\end{align*}
\]

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 accessed by the expression.

### 5.2 Notations

Types are defined by:

\[
\begin{align*}
ct &::= T \mid \alpha \mid \{\alpha\} \mid ct \times ct \\
&\quad \mid (ct, ct) \text{ event}\{\alpha\} \\
&\quad \mid ct \rightarrow ct \mid ct\text{ process}\{\alpha|cf\}
\end{align*}
\]

\[
\begin{align*}
\text{ce} &::= \gamma \mid ck \\
\text{cf} &::= \phi \mid \emptyset \mid \{\alpha\} \mid cf \cup cf \\
\text{cs} &::= ct \mid \forall \alpha. cs \mid \forall \gamma. cs \mid \forall \phi. cs \\
\Gamma &::= [x_1 \mapsto cs_1; \ldots; x_p \mapsto cs_p]
\end{align*}
\]

A type is either a basic type \(T\), a type variable \(\alpha\), a singleton type \(\{\alpha\}\) corresponding to the clock \(ce\), a product, a signal, a function or a process. The type \((ct, ct) \text{ event}\{\alpha\}\) 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 \(\phi\). 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:

\[
\begin{align*}
\text{cs}[\alpha \leftarrow ct] &\leq \forall \alpha. cs \\
&\quad \text{cs}[\gamma \leftarrow cf] \leq \forall \gamma. cs \\
&\quad \text{cs}[\phi \leftarrow cf] \leq \forall \phi. cs
\end{align*}
\]

\[
\begin{align*}
\text{gen}(ct, e, \Gamma) &= ct \\
\text{gen}(ct, e, \Gamma) &= \forall \alpha. \forall \gamma. \forall \phi. ct \\
&\quad \text{if } e \text{ is expansive} \\
&\quad \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
\text{if } \alpha, \gamma, \phi = fte(ct) \setminus fte(\Gamma)
\end{align*}
\]
where $f(tv(ct))$ 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 $\Gamma$ and local clock $ce$, the expression $e$ has type $ct$ and effect $cf$.

The initial typing environment $\Gamma_0$ contains the signatures of all primitives:

$$\Gamma_0 \triangleq \{\text{global}_\text{ck} : \{\text{Var}_\text{ck}\} ; \text{pause} : \forall \gamma, \gamma \vdash \gamma \Rightarrow \text{unit}; \\text{last} : \forall \alpha_1, \alpha_2, \gamma, (\alpha_1, \alpha_2) \text{event}(\gamma) \Rightarrow \alpha_2; \\text{emit} : \forall \alpha_1, \alpha_2, \gamma, (\alpha_1, \alpha_2) \text{event}(\gamma) \Rightarrow \alpha_1; \text{true} : \forall \alpha, \alpha \Rightarrow \text{unit}\}.$$  

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 PROCAIBS 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 $\text{ProcApp}$). 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 $\text{WHen}$ 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 $\text{DOMAIN}$. 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 $\Gamma$. It is similar to the typing of let in [10] and let $\text{clock}$ 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 $e \triangleq \text{signal} \ s \ in \ s$):

$$\Gamma_0, x : \gamma, \gamma \vdash e : (\alpha, \alpha \text{list}) \text{event}(\gamma) \mid [\gamma]$$

In the case of the signal_escape example, this variable appears in the type of the variable $\text{slow}$ in the typing environment:

$$\text{slow} : ([\text{int}, \text{int}] \text{event}(\gamma), [\text{int}, \text{int}] \text{event}(\gamma)) \text{event}(\Gamma, \text{ck})$$

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

$$f : \text{unit} \Rightarrow \text{unit}$$

### 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 $\Gamma_0, \text{ck} \vdash p : ct \mid cf$ with $cf \subseteq \{\text{ck}\}$, then either:

- There exists $v, H, S$ such that $\tilde{p}/H_0, S_0 \Rightarrow^* v/H,S$.
- For each $p', H', S'$ such that $\tilde{p}/H_0, S_0 \Rightarrow^* p'/H', S'$, there exists $p'', H'', S''$ such that $p'/H', S' \Rightarrow p''/H'', S''$.

The proof is given in Appendix A 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 tmp 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:

```
signal s in e ≜ signal s 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 nonomorphic: it is a type cf, 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 top-level. 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 subfecting [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 Gemünden 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

APPENDIX

A. PROOF OF SOUNDNESS OF THE TYPE SYSTEM

As in [5], we introduce two new typing environments in order to demonstrate the soundness of the type system: the signal typing environment, denoted Σ, and the clock typing environment, denoted H. Whereas the typing environment Γ maps variables x to types, Σ maps clock names to a partial map from signal names n to signal types (ct1, ct2) \text{event}\{ce\}. H maps clock names to the type of their parent clocks. We also denote Sig(S) = \{n ∈ Dom(S(ck)) | ck ∈ Dom(S)\} the set of all signal names defined in a signal environment.

Clocking judgments are now Γ, Σ, H, ce ⊢ e : ct | cf. There are two new rules for typing signals and clocks:

\begin{align*}
\Gamma, \Sigma, H, ce \vdash ct : \{ck\} &\quad | \quad \emptyset
\end{align*}

There is a new side-condition in the IN rule: H(ck') = ce.

The signal environment is left untouched by all other typing rules. The crucial point is that in the DOMAIN rule, the side-condition that γ does not escape only applies to the typing environment Γ, not the signal environment Σ.

Definition 1. We say that a signal environment S is well-typed in the signal typing environment Σ, denoted Σ ⊢ S if both contain the same clock and signal names, i.e. nck ∈ Dom(S) if and only if nck ∈ Dom(Σ(S(ck))).

Definition 2. We say that a clock environment H is well-typed in the clock typing environment H, denoted H ⊢ H, if they have the same domain and:

\[
\forall ck \in \text{Dom}(H), \exists \gamma, H^\alpha(ck) : H(ck) | \emptyset
\]

We can now define the typing judgment of a configuration e/H, S:

\[
\begin{array}{c}
\Gamma, \Sigma, H, ce \vdash e : \text{ct} | \text{cf} \\
\text{Dom}(H) = \text{Dom}(\Sigma) \\
\Sigma, H, ce \vdash e/H, S : \text{ct} | \text{cf}
\end{array}
\]

Lemma 4. If Γ, Σ, H, ce ⊢ e : ct | cf and Σ ⊢ S and H ⊢ H', then Γ', Σ', H', ce ⊢ e : ct | cf.


Lemma 6 (Clock substitution). Let θ be a substitution that maps a clock variable to a clock name (i.e. θ = \{γ ↦ ck\}). If Σ, H, ce ⊢ e/H, S : ct | cf, then Σ, H, θ(ce) ⊢ e/H, S : θ(ct) | θ(cf).

Proof. By induction on the typing rules, we can prove that Σ, H, ce ⊢ e/H, S : ct | cf.

Before proving typing preservation for the step reduction, we first need to prove that the end-of-instant relation preserves typing, as it is used by the LOCAL-Eoi rule.

Property 1 (Typing preservation for \(\text{ck}\)). If Σ, H, ce ⊢ e/H, S : ct | cf and e/H, S \text{cf} \rightarrow e'/H', S' with ck ∈ C, then there exists S', H' and cf' such that:

\(\Sigma', H', ce ⊢ e'/H', S' : ct | cf'\)

Proof. By induction on the structure of e.

Case domain \(x \bowtie v\) do e:

Then we have \(e' = [x ← ck']\) in ck' and S' = S(ck' →), where ck' /∈ Dom(H). From Domain, we deduce that Γ0, Σ, H, ce ⊢ e : ct | cf, with γ /∈ fev(cf) and cf = cf' \{γ\}. Let’s denote θ = \{γ ↦ ck\}. By applying lemma 6, we get Γ0, Σ, H, θ(γ) ⊢ e : θ(ct) | θ(cf'). Then, we have θ(cf') = ct as γ /∈ fev(cf). If we denote H' = H(ck' →) and S' = S(ck' →), then as H'(ck') = ck and H' ⊢ H, we have H'(ck') = ck'. We can replace Σ with S' and H' with H' in the previous typing judgment using Lemma 4. By applying Inx, we then get that Γ0, Σ', H', ce ⊢ e' : ct | cf', where cf' = cf[γ ↦ ck'] = cf.

Case e in ck' Then ce = v, S' = S and H' = H. From Inx, we get that Γ0, Σ, H, ck' ⊢ v : ct | cf1 with cf = cf1 \{ck\} and cf1 /∈ fex(cf). From lemma 2, we know that cf1 = cf = \emptyset. We can apply lemma 3 to get Γ0, Σ, H, ce ⊢ v : ct | \emptyset and the other conditions are immediate.

Case e in ck' (Step): From Inx, we deduce that Γ0, Σ, H, ck' ⊢ e : ct | cf1 with cf = cf1 \{ck\}. As H(ck') = \{ck\} and H ⊢ H, we have Γ0, [], γ ⊢ H^\alpha(ck') : \{ck\} | \emptyset, which means that H^\alpha(ck') = ck, i.e. ck' /\leq H ck. As we also have ck' /\leq H cf from the typing rule, it follows that ck' /\leq H cf1. By induction, we have that Γ0, Σ', H', ce' ⊢ e' : ct | cf', with ck' /\leq H cf'. We can then apply the IN rule to get the desired result.

Case e in ck' (LOCAL-Eoi): As in the previous case, we can show that Σ, H, ck' ⊢ e/H, S : ct | cf1. By applying Property 1, we get that Γ0, Σ, H, ck' ⊢ e'/H, S : ct | cf'. We can then deduce the expected result by using Lemma 1.
Case signal \( x \) default \( v_1 \) gather \( v_2 \) in \( e \) : Then 
\( e' = [e/x/n^e] \) and \( S' = Σ\{ck\}(n \mapsto (v_1, v_2, \emptyset)) \). From the typing rule, we get that \( Γ_0, x : (ct, ct_2) \) event \( Σ\{ck\}, Σ, H, ck \vdash e : ct \mid cf_1 \) where \( cf = cf_1 \cup \{ck\} \). Let’s denote \( Σ' = Σ[n \mapsto (ct_1, ct_2) \) event \( Σ\{ck\}] \). By applying Lemma 4 and 5, we get \( Γ_0, Σ', H, ck \vdash e[x \mapsto n^e] : ct \mid cf_1 \) and we can easily prove that \( Σ' \vdash S' \).

**Case exit \( n^e \) \( v \) :** We have \( e' = () \) and \( S' = S + [v/n] \)
We can easily prove that \( Γ_0, Σ, H, ck \vdash () : \) unit \mid \( \emptyset \) and that \( Σ \vdash S' \) as \( Γ_0, Σ, H, γ \vdash v : ct_1 \mid \emptyset \) where \( Σ(ck')(n) = (ct_1, ct_2) \) event \( Σ\{ck\}' \).

**Definition 3.** We say that \( e/\{\cdot\}, S \) is an end-of-instant configuration for \( ck \) if there exists \( e', H' \) and \( S' \) such that \( e/\{\cdot\}, S \xrightarrow{\text{ck}} e'/H', S' \).

For the proof of the progress property, we need to distinguish instantaneous expressions and non-instantaneous expressions. For that, a very simple syntactic classification is used, called well-formed rules and denoted \( k \vdash \) in [13] (it extends easily to the new language). Stuck instantaneous expressions are values, whereas stuck non-instantaneous expressions are end-of-instant expressions.

**Property 3** (Progress for instantaneous expressions). If 
\( Σ, H, ck \vdash e/\{\cdot\}, S : ct \mid cf \) and \( e \) is instantaneous (i.e. \( 0 \vdash e \)), then either:
- \( e \) is a value
- There exists \( e', H' \) and \( S' \) such that \( e/\{\cdot\}, S \xrightarrow{\text{ck}} e'/H', S' \)

**Property 4** (Progress for \( \text{ck} \)). If 
\( Σ, H, ck \vdash e/\{\cdot\}, S : ct \mid cf \), then either:
- \( e/\{\cdot\}, S \) is an end-of-instant configuration for \( ck \)
- There exists \( e', S' \) such that \( e/\{\cdot\}, S \xrightarrow{\text{ck}} e'/H', S' \)

**Proof.** By induction on the structure of \( e \).

**Case domain \( x \) by \( v \) do \( e \) :** Then \( e' = e[x \leftarrow ck'] \) in \( ck' \).

**Case \( e \) in \( ck' \) :** We can show that \( Σ, H, ck' \vdash e/\{\cdot\}, S : ct \mid cf_2 \) like in the proof of Property 2. By induction, either there exists \( e', H' \) and \( S' \) such that \( e/\{\cdot\}, S \xrightarrow{\text{ck}} e'/H', S' \), in which case we can apply STEP, or \( e \) is an end-of-instant expression for \( ck' \) and then:
- Either we can apply LOCALEOI to do a step reduction on \( ck \)
- Otherwise, \( e \) in \( ck' \) is an end-of-instant expression for \( ck \) (one of STUCKDOMAIN or PARENTEOI necessarily applies)

**Case exit \( e_1 \) \( e_2 \) :** From the well-formation rules, we know that \( e_1 \) and \( e_2 \) are instantaneous expressions, so we can apply Property 3. If \( e_1 \) or \( e_2 \) is not a value, then we can reduce by using the context rule. Otherwise, we have \( e_1 = n^e \) and \( e_2 = v \). From the typing rule, we get that \( ck' \in cf \), so \( ck \leq H \) \( ck' \) as \( ck \leq H \) \( cf \). We can thus reduce \( e \) to \( e' = () \). The proof is similar for all actions on signals.

**Case exit \( e_1 \) \( e_2 \) :** From the well-formation rules, we know that \( e_1 \) and \( e_2 \) are instantaneous expressions, so we can apply Property 3. If \( e_1 \) or \( e_2 \) is not a value, then we can reduce by using the context rule. Otherwise, we have \( e_1 = n^e \) and \( e_2 = v \). From the typing rule, we get that \( ck' \in cf \), so \( ck \leq H \) \( ck' \) as \( ck \leq H \) \( cf \). We can thus reduce \( e \) to \( e' = () \). The proof is similar for all actions on signals.

**Property 5** (Soundness of the type system for \( \text{ck} \)). If 
\( Σ, H, ck \vdash e/\{\cdot\}, S : ct \mid cf \) , then either:
- There exists \( e', H' \) and \( S' \) such that \( e/\{\cdot\}, S \xrightarrow{\text{ck}} e'/H', S' \) and \( e' \) is an end-of-instant expression for \( ck \) (i.e. there exists \( e'' \) such that \( e'/H', S' \xrightarrow{\text{ck}} e''/H'', S'' \))
- For each \( e'/H, S \) such that \( e/\{\cdot\}, S \xrightarrow{\text{ck}} e'/H', S' \), there exists \( e''/H'', S'' \) such that \( e'/H', S' \xrightarrow{\text{ck}} e''/H'', S'' \)

The initial clock typing environment is \( H_0 = \emptyset \). We easily check that \( H_0 \vdash H_0 \).

**Theorem A.1** (Soundness of the type system).
If \( \emptyset, H_0 \vdash p/H_0, S_0 : ct \mid cf \) , then either:
- There exists \( v, H, S \) such that \( p/H_0, S_0 \Rightarrow v/H, S \).
- For each \( p', H', S' \) such that \( p/H_0, S_0 \Rightarrow p'/H', S' \), there exists \( p'', H'', S'' \) such that \( p'/H', S' \Rightarrow p''/H'', S'' \).