ReactiveML, ten years later
Louis Mandel, Cédric Pasteur, Marc Pouzet

To cite this version:
Louis Mandel, Cédric Pasteur, Marc Pouzet. ReactiveML, ten years later. 17th International Symposium on Principles and Practice of Declarative Programming (PPDP’15), Jul 2015, Siena, Italy. pp.6 - 17, 10.1145/2790449.2790509 . hal-01508179

HAL Id: hal-01508179
https://inria.hal.science/hal-01508179
Submitted on 25 Apr 2017

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.
Abstract
Ten years ago we introduced ReactiveML, an extension of a strict ML language with synchronous parallelism à la Esterel to program reactive applications. Our purpose was to demonstrate that synchronous language principles, originally invented and used for critical real-time control software, would integrate well with ML and prove useful in a wider context: reactive applications with complex data structures and sequential algorithms, organized as a dynamically evolving set of tightly synchronized parallel tasks.

While all ReactiveML programs presented at PPDP'05 still compile, the language has evolved continuously to incorporate novel programming constructs, compilation techniques and dedicated static analyses. ReactiveML has been used for applications that we never anticipated: the simulation of large-scale ad-hoc and sensor networks, an interactive debugger, and interactive mixed music. These applications were only possible due to the efficient compilation of ReactiveML into sequential code, which we present here for the first time. We also present a parallel implementation that uses work-stealing techniques through shared memory. Finally, we give a retrospective view on ReactiveML over the past ten years.

1. Introduction
All general purpose programming languages propose a notion of parallel composition and synchronization. It is essential to program applications where several tasks evolve concurrently and communicate. If we consider OCaml, for example, there is the preemptive threads module Thread of the standard library,1 cooperative threads libraries like Lwt,2 Async,3 and Muthreads.4 There is also a library of Functional Reactive Programming (FRP),5 and a library for event driven programming.6

A distinctive feature of ReactiveML is to provide a deterministic model of concurrency with rich control structures. ReactiveML programs can await and react simultaneously to several events, compose processes in parallel and modularly suspend or preempted parts of a system.

The concurrency model of ReactiveML is based on that of Esterel [5, 8], a language designed for programming control in safety critical real-time systems (e.g., avionic software [7]). This model of concurrency is the synchronous model [3]. It relies on a notion of time defined as a succession of logical instants. In the context of real-time systems, logical time allows software to be programmed without having to worry about physical time and then later to check that the worst case execution time of an instant respects the real-time constraints.

Frédéric Boussinot was the first to see that the synchronous model can be used outside the scope of real-time systems [10, 13]. The notion of logical time is useful even if not bound to physical time. In particular, it allows for a precise and deterministic semantics of concurrency and some expressive control structures. ReactiveML is build on these ideas.

In our 2005 article [28], we introduced the ReactiveML language and defined its formal semantics. In this article, after an illustration of the language on an example (section 2), we explain its compilation (section 3) and the runtime (section 4). The implementation of suspension and preemption is explained in section 5. We present a parallel implementation of the runtime (section 6) and compare our implementation with other related systems (section 7). Finally, we will reflect on our experiences over the last ten years and discuss the choices made in ReactiveML (section 8).

The compiler and examples are available at the address http://reactiveml.org.

2. A Complete Example
In this section we propose a solution to the 2013 ICFP Contest.7 This example illustrates the mixing of algorithmic and reactive parts in a ReactiveML program. It also serves to present the language syntax and intuitive semantics.

The purpose of the contest is to guess secret programs written in a small language called \BV. Each program to guess is called a problem. It as an ID and some meta-data about the program to guess such as its size or the operators that it uses. The contest organizers control a game server that provides the following services: (1) evaluate a secret program on an array of 256 inputs; (2) check contestants solutions. If a contestant submits an incorrect solution, the server provides a counterexample. Contestants have 5 minutes to solve each problem. They are allowed to send at most 5 requests to the game server every 20 seconds. Communication with the game server occurs through a web API.

1http://caml.inria.fr/pub/docs/manual-ocaml-4.02/libthreads.html
2http://ocisigen.org/lwt
3https://ocaml.janestreet.com/ocaml-core/111.28.00/doc/async/
4http://christophe.deleuze.free.fr/muthreads
5http://erratique.ch/software/react
6http://www.camlcity.org/archive/programming/equeue.html
7http://icfpc2013.cloudapp.net

1[Copyright notice will appear here once 'preprint' option is removed.]
In the \BV language, a program is a function that takes as input a 64-bit vector and returns a 64-bit vector. The expression defining the body of the function can be the constant 0 or 1, a variable, a test to zero, a unary or binary operation (negation, conjunction, etc.) or an iterator over a 64-bit vector. The abstract syntax tree of this \BV language can be implemented in ReactiveML with the following data structure:

```
type program = { input: ident; expr: expr }
and ident = { name: string; mutable value: Int64.t }
and expr =
  | Const of Int64.t
  | Var of ident
  | If_Zero of expr * expr * expr
...
type problem = { id: string; size: int; ... }
```

A program is a record that represents a function with an argument `input` and a body `expr`. Variables are represented by the type `ident`: a record with a field value that can be modified. An expression is represented by a sum type. A problem is a record that contains its ID and meta-data. Notice that we use the type `Int64.t` from the OCaml standard library since ReactiveML is compatible with OCaml.

We can define algorithms over these data structures. Below is a snippet of the functions `eval_expr` and `eval` that evaluate, respectively, an expression and a program:

```
let rec eval_expr e = match e with
  | Const c -> c
  | Var v -> v.value
  | If_Zero (e1, e2, e3) ->
    if eval_expr e1 = Int64.zero then eval_expr e2
    else eval_expr e3
...

let eval p n =
  p.input.value <- n;
  eval_expr p.expr
```

Data types and algorithmic functions can thus be defined as in OCaml.

ReactiveML is based on the synchronous model of computation where time is defined as a succession of logical instants. Functions that are executed over several instants are called processes. A player is a typical reactive application that is implemented as a process. A possible architecture for this process is the parallel composition of two sub-processes: (1) a process `guesser` dedicated to generation of programs that satisfy the (partial) specification of the program to guess and (2) a process `communicator` dedicated to the communication with the game server to ensure that not too many requests are sent to the game server. The `communicator` process periodically asks for more information about the program being guessed to improve the specification and as soon as the process `guesser` finds a program with a potential solution, it submits this solution to the game server. The partial specification of a secret program is a pair of inputs and corresponding expected outputs. It will be represented by the type `spec_t`:

```
type spec_t = Int64.t array * Int64.t array
```

In ReactiveML, communications between processes are realized through signals. The process `communicator` provides the specification of the secret program to the process `guesser` through a signal `spec` and the process `guesser` provides the process `communicator` with solutions to submit to the server through a signal `guess`.

The type of a signal is `(t1, t2) event` where `t1` is the type of values emitted on the signal and `t2` is the type of the value carried by the signal. The value carried by a signal is the combination of the emitted values. The combination function is provided by the programmer. For example, the signal `spec` accumulates emitted values in a list `(_: is list concatenation)`: `signal spec memory [] gather (fun x y -> x :: y)` At each emission of the signal, the combination function `(fun x y -> x :: y)` is called with `x` bound to the emitted value and `y` bound to the result of the previous combination. The first emitted value is combined with the empty list `([],)`. The keyword `memory` indicates that the value of the signal is kept from one instant to another. If the keyword `default` is used instead of `memory`, the first emission at each instant is combined with the default value.

For example, we can declare the signal `guess` as a signal that can be emitted at most once per instant or fail otherwise:

```
signal guess default None gather (fun x y ->
  match y with
  | None -> Some x
  | Some _ -> assert false)
```

At the first emission, the value of the signal becomes `Some x`. A second emission during the same instant would fail because it can detect that the signal has already been emitted (`y` would be equal to `Some x`).

The expected behavior of the process `guesser` is the following: from the description of a problem `pb` of type `problem`, this process emits on the signal `spec` programs that respect the specifications accumulated on the signal `spec`. Here, we propose a naive implementation that generates random programs and checks if they satisfy the specification:

```
let process guesser pb spec guess =
  loop
    let f = random_program pb in
    if check f (last ?spec) then emit guess f;
    pause
  end

val guesser: problem ->
  ('a, spec_t list) event -> (program, 'b) event ->
  unit process
```

This function is introduced with the keywords `process` to mark that it can be executed over several instants. The body of the process is an infinite loop `loop/end` that executes one iteration per instant: the expression `pause` marks the suspension of the process until the next instant. At each iteration, the process first generates a program `f` using the function `random_program` that returns a random value of type `program` from the description of the problem `pb`. Then the process reads the list of pairs of inputs and outputs that the program must respect from the signal `spec`. The value of the signal is accessed with the expression `last ?spec` that returns the value of `spec` from the previous instant. The delay in the access of the value of the signal ensures the absence of causality issues (as discussed in section 8). Finally, using the function `check`, the process checks that the generated program `f` satisfies the current specification and, if this is the case, emits `f` on the signal `guess` as a potential solution. The function `check` has type `program -> spec_t list -> bool`. It uses the function `eval` defined earlier to test if the generated program computes the expected outputs.

The type of the program (written in `italics`) is inferred by the compiler. We can see that `guesser` is a function parameterized by a problem and two signals. Since the process `guesser` does not emit values on the signal `spec` and does not read values of the signal `guess`, the type on emission for `spec` and the type of reception for `guess` are not constrained (`'a` and `'b` are type variables that are implicitly universally quantified).
The specification of the secret program emitted on the signal `spec` is obtained by the process `increase_spec`. This process continuously communicates with the game server:

```ocaml
let process increase_spec pb spec =
  loop
  let inputs = next_inputs (last ?spec) in
  let outputs = Webapi.eval pb.id inputs in
  emit spec (inputs, outputs);
  pause
end
val increase_spec : problem -> (spec_t, 'a) event -> unit process
```

This process is an infinite loop that first creates an array inputs of 256 inputs for which we want to know the corresponding outputs using a function `next_inputs`. This function can use the current specification to decide how to create the array. Then it communicates with the game server through HTTP using the function `Webapi.eval` to obtain the outputs corresponding to the inputs. Finally, it emits the pair of inputs and outputs on the signal `spec` to improve the specification.

We now define the process `communicator` that handles communication with the game server. It is composed of two parts that are executed in parallel. The first part, lines 2 to 8, is a loop that first waits for the emission of the signal `spec` binding `f` to the value received (line 3). This value is a potential solution computed by the process `guesser`. Then, at line 4, the loop asks the game server if this program is a solution to the problem using the function `Webapi.guess`. In that case case (line 5), the signal `finished` is emitted. Otherwise (line 6-7), a counterexample is provided by the game server and emitted on the signal `spec` to refine the specification (input and output are put in an array to have type `spec_t`). The second part of the process, lines 10 to 16, periodically asks the server for more information about the program to guess using the process `increase_spec` (which is supposed to send a request per instant) is executed inside a `do/when` construct controlled by the signal `finished`. The behavior of this construct is not a loop, it preempts the execution of its body at the end of the instant where the control signal is emitted.

```ocaml
val communicator : problem -> (spec_t, 'b) event -> unit process
```

Finally, the process `solve` tries to solve a problem given as parameter in less than 5 minutes. It declares the signals `spec` and `guess` as explain earlier and the signal `finished` always associated with the value `()`. Then, it executes in parallel the process `communicator`, the process `guesser` and a process that waits 5 minutes before emitting finished. To stop the execution of these three parallel branches as soon as a solution is found or the 5 minutes have expired, they are executed inside a `do/until` construct controlled by the signal `finished`. The behavior of this construct is not a loop, it preempts the execution of its body at the end of the instant where the control signal is emitted.

```ocaml
let process solve pb =
  signal spec memory []
  gather (fun x y -> x :: y) in
  signal guess default None
  gather (fun x y -> ...) in
  signal finished default ()
  gather (fun x y -> () in
  do
    run communicator pb spec guess finished ||
    run guesser pb spec guess ||
    run wait (5. *. 60.) ; emit finished ()
  until finished done
val solve : unit process
```

This completes our introduction to the basic constructs of ReactiveML. A more complete interactive tutorial is available at the address http://reactiveml.org/tryrml.

### 3. Compilation

The implementation of ReactiveML relies on the classical idea of implementing concurrency with continuations [32, 37]. As a first step, we consider a language without suspension and preemption. Then, in section 5, we explain how these control structures are added.

The ReactiveML compiler generates OCaml code and uses an intermediate language based on continuations called $L_s$. The translation from ReactiveML to $L_s$ is mainly a partial CPS (Continuation Passing Style) transformation. For each potentially blocking operation, like `pause` or awaiting a signal, we create a continuation containing the ensuing computations. Other language constructs remain unchanged.

### ReactiveML without suspension and preemption

We consider a non-minimal kernel of ReactiveML without preemption and suspension defined by:

```ocaml
val solve =
  signal spec memory []
  gather (fun x y -> x :: y) in
  signal guess default None
  gather (fun x y -> ...) in
  signal finished default ()
  gather (fun x y -> () in
  do
    run communicator pb spec guess finished ||
    run guesser pb spec guess ||
    run wait (5. *. 60.) ; emit finished ()
  until finished done
```

It is a call-by-value lambda calculus extended with process creation (`process`) and execution (`run`), waiting for the next instant (`pause`), parallel definitions (`let/and`), declarations of signals (`signal`), signal emissions (`emit`), awaiting signal emission (`await immediate`), awaiting a signal value (`await`) and tests for signal presence (`present`). The `await immediate` $s$ expression terminates instantaneously when the signal $s$ is emitted. The `present` $s$ then $e$ or else $e_2$ expression executes $e_1$ instantaneously if the signal $s$ is present or executes $e_2$ at the next instant if the signal is absent. The idea of introducing a delay in the `else` case originates in ReactiveV [10]: it prevents two processes from seeing different statuses for a signal at an instant.
This kernel suffices to encode most of the other constructs of the language. We write _ to denote variables that do not appear free in the body of a let and () the unique value of type unit:

\[ e_1 \mid e_2 \triangleq \text{let } _= e_1 \text{ and } _= e_2 \text{ in } () \]

let \( x = e_1 \) in \( e_2 \triangleq \text{let } _= x \text{ and } _= () \text{ in } e_2 \)

let process \( f x = e_1 \) in \( e_2 \triangleq \text{let } f = \lambda x.\text{process } e_1 \text{ in } e_2 \)

let rec process \( f x = e_1 \) in \( e_2 \triangleq \text{let } f = (\text{rec } f = \lambda x.\text{process } e_1) \text{ in } e_2 \)

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]

This is instantaneous expression. The predicate \( e \) where \( e \)

\[ k \]
We now describe the OCaml implementation of the runtime which realizes the semantics of the $L_k$ language (presented formally elsewhere [23, section 7.2]).

### Execution principles

The ReactiveML runtime is a cooperative scheduler: processes must yield control to the scheduler for other processes to execute. The scheduler maintains a set $C$ of continuations to be executed in the current instant (it is implemented as a list). Combinators like `split` add their continuations to this set. A second set `next` contains the processes to execute at the next instant. Constructs like `pause` add their continuations to this set. The execution of one instant follows this algorithm:

1. Processes in the set $C$ are executed until the set becomes empty.
2. Then, an end of instant reaction is performed. All processes that test for signal absence or that wait on the value of an emitted signal are awakened by adding their continuations to $C$ for execution at the next instant. The processes in the set `next` are transferred into $C$ and signals are reset. Finally, a new instant is started by looping back to the first step of the algorithm.

These two phases are part of the operational semantics of ReactiveML [28]. The semantics is described by two reductions that corresponds respectively to the execution during the instant and at the end of instant.

The implementation of the base algorithm in the runtime has two important features:

- It implements passive waiting so that a process waiting for the emission of a signal has no impact on the runtime until the signal is emitted. That is, processes do not test for signal presence multiple times during an instant or during an instant where it is absent. To avoid such busy waiting, we associate each signal with a waiting list: a set of continuations that depend
module type Runtime = sig

(* [on_current_instant f] executes [f] during the current instant *)
val on_current_instant : unit step -> unit

(* [on_next_instant f] executes [f] during the next instant *)
val on_next_instant : unit step -> unit

(* [on_eoi f] executes [f] during the end of instant *)
val on_eoi : unit step -> unit

(* [on_event ev f] executes [f] when [ev] is emitted *)
val on_event : ('a, 'b) event -> unit step -> unit

(* [on_event_or_next f ev f.n] executes [f.ev] if [ev] is emitted, otherwise it executes [f.n] during the next instant *)
val on_event_or_next : ('a, 'b) event -> unit step -> unit step -> unit
... 

Figure 3: Interface of the Runtime module

The expression expr_evt() triggers the evaluation of expr_evt to give a signal against which we register the continuation k.

Similarly, the primitive on_next_instant allows the direct implementation of pause:

let rml_pause k _ = Runtime.on_next_instant k

The other combinators are obtained by the composition of several primitives. For example, we can implement a function on_event_at_eoi that executes a continuation f at the end of the instant in which the signal evt is emitted:

let rml_wait_all expr_evt k _ =

    Runtime.on_next_instant (fun () -> k v)

This combinator takes as input a signal expr_evt and a continuation k that waits for the value of the signal. We use the function on_event_at_eoi to execute the function await_eoi at the end of the first instant in which the signal evt is emitted. This function reads the value of the signal using the Runtime.Event.value function that returns the current value of the signal. Finally, we register the continuation k for execution during the next instant with the value v of the signal as argument.

The expression expr_evt() triggers the evaluation of expr_evt to give a signal against which we register the continuation k.

Similarly, the primitive on_next_instant allows the direct implementation of pause:

let rml_pause k _ = Runtime.on_next_instant k

The other combinators are obtained by the composition of several primitives. For example, we can implement a function on_event_at_eoi that executes a continuation f at the end of the instant in which the signal evt is emitted:

let rml_wait_all expr_evt k _ =

    Runtime.on_next_instant (fun () -> k v)

This combinator takes as input a signal expr_evt and a continuation k that waits for the value of the signal. We use the function on_event_at_eoi to execute the function await_eoi at the end of the first instant in which the signal evt is emitted. This function reads the value of the signal using the Runtime.Event.value function that returns the current value of the signal. Finally, we register the continuation k for execution during the next instant with the value v of the signal as argument.

Synchronous parallel composition

The implementation of synchronous parallel composition (let x = e and x = e in e) corresponds to the constructs split, join and def in \( L_4 \). The principle of the implementation is as follows.

- The split combinator creates a synchronization point which is a counter initialized to the number of branches, and also one reference per branch to store the computed results.
- The join combinator decreases the counter created by split and stores the result computed by its branch in the associated reference. If the counter is not zero, control returns to the scheduler. If the counter is zero, then all branches have terminated. The continuation of this combinator is passed a tuple containing the values computed by the branches.
- The def combinator binds the values computed by its two branches to its body. In the implementation, join and def are defined in a single function.

For efficient execution, the runtime also defines an n-ary parallel operator based on the same principle. Further optimization is also possible. For example, it is possible to dynamically share synchronization points. The idea is that if the continuation of a parallel composition is a join of another parallel composition, then we can use the same counter for the two parallel compositions. To do that, each combinator takes a synchronization point as argument that is either None or Some ip where ip is the counter associated to the enclosing parallel. When split is executed, it reuses the counter given as argument or it creates a new one if the argument is None. Thanks to this approach, we only need one counter for a dynamic number of processes. Using this optimization, the following program can be executed in constant memory, because even though it creates an unbounded number of processes, there are only ever ten
processes alive at the same time (in the steady state) and there is only one synchronization point:

```ml
let rec process p =
  for i = 1 to 10 do pause done
||
  pause; run p
```

### 5. Suspension and preemption

The previous section presented the basic principles of the implementation of the ReactiveML runtime. We now extend these ideas to address suspension and preemption. These two control structures distinguish ReactiveML from the other libraries for cooperative programming but also make its implementation more complex. A formal account of the ideas presented here, that is, the semantics of $L_k$ with preemption and suspension, is given in [23].

#### Principles

As mentioned in section 4, the use of the set $C$ of continuations disregards the structure of programs and necessitates an additional data structure to implement the preemption and suspension. This structure is called the control tree. It is an n-ary tree which tracks the dynamic nesting of preemptions and suspensions. Each node corresponds to a do/until or a do/when in the running program. Consider the following example:

```ml
let process control_tree s p1 p2 =
  emit s; pause; run p1
||
    do
    pause; run p2
    when s done

The process control_tree takes as argument a signal s and two processes p1 and p2. The figure on the right shows the control tree associated with the execution of this program at the end of the first instant. Each node of the control tree contains a set next of continuations to execute during the next instant that the body of the control structure is active, that is, for the suspension in the example, the next instant where the signal s is present. Without any suspension or preemption, next is the set associated to the node Top and it represents the set of continuations activated at each instant. In this example, at the end of the first instant, the next at the root of the tree contains p1 and the next at the suspension node contains p2. To prepare for execution of the second instant, we transfer the set next at the root into the set C, but not the set at the node when s. The latter transfer is only made when the signal s is emitted.

In the implementation, all combinators take as argument the node of the control tree that corresponds to their execution context. The translation function $C[e]$ (figure 2) is modified to include these additional arguments. In a similar way, the primitives on_next_instant, on_event and on_event_or_next of figure 3 take control tree nodes as arguments. For example, on_next_instant ctrl f adds the function f to the set next associated to the control node ctrl. This registers f for execution during the next instant that the ctrl node is activated.

The language $L_k$ is also extended with new continuations to represent the start and end of do/until and do/when.

#### Implementation of do/until

In ReactiveML, preemption is weak, meaning that the body of a do/until is only interrupted at the end of the instant where the preemption signal is emitted. Therefore, one solution is to check at the end of each instant whether the signal is present. If so, we remove the corresponding sub-tree in the control tree and add the continuation of the preemption to the next set of the parent node. Otherwise, the set next of the control tree that contains the current state body of the do/until is transfer to the next set of the parent node.

Another solution is to use the on_event primitive to wait for the emission of the preemption signal, and, when the signal is emitted, to trigger a function that removes the sub-tree at the end of the instant (if it was not suspended during the instant by a do/when).

#### Implementation of do/when

The body of a do/when is active only during the instants where the control signal is present. It means that by default, the body of a do/when is put in the set next of this node in the control tree. When the control signal is emitted, the continuations that are in this next set are transferred into the set C of continuations for execution in the current instant.

During the end of an instant, the scheduler do not traverses sub-trees of the control tree that were not activated during the instant. For each do/when, a continuation that triggers the activation of these control tree nodes is registered in the next sets of their parent nodes.
Passive waiting with preemption and suspension

We described in section 4 that when a process is waiting for a signal, it is put in a waiting list attached to the signal and only woken up when the signal is emitted. But if the process is executed under the control of a do/when block, it must only be executed if the control signal is also present during the instant. Otherwise the process should remain suspended until the two signals are present simultaneously. This is illustrated in the following program:

```ocaml
let process await_when =
  signal act default () gather fun x y -> () in
  signal s default () gather fun x y -> () in
  do
    await immediate s; print_endline "Received!"
  when act done
  || loop emit act (); pause; pause end
  || pause; emit s (); pause; emit s ()
```

This process waits for the emission of a to print, in the same instant, the message "Received!" (line 5), but only during the instants where the signal act is present (lines 4 to 6), that is during the odd instants (line 7). The emission of s occurs during the second and third instants (line 8). So during the first instant, the do/when is activated, but since s is absent, the message "Received!" is not printed. During the second instant, s is emitted but the do/when is suspended. Therefore, the process awaiting s is not woken up. During the third instant, both s and act are present, and the message "Received!" is printed.

A first solution to this problem (which was used in early versions of ReactiveML) is to use busy waiting to await signals subject to suspension or preemption. Another approach is to allow passive waiting in all cases. The following algorithm is used to trigger a function \( f \) when the signal \( \text{evt} \) is emitted and the control node \( \text{ctrl} \) is active:

1. If the signal is present when we start waiting, we can call \( f \) directly. Indeed, we know that \( \text{ctrl} \) is active since we are executing the process in this context.

2. Otherwise, we put a function in the waiting list of the signal. On signal emission, the function executes the following steps:
   - If the control node is active, it calls \( f \).
   - Otherwise, we wait for both the end of the instant and an activation of the control node \( \text{ctrl} \) later in the same instant (in the previous example, \( s \) can be emitted before \( act \)). If \( \text{ctrl} \) is activated before the end of the instant, we call \( f \). Otherwise, at the end of the instant, we know that the node is not active which means that we must wait for the next emission of \( \text{evt} \). We thus restart at the beginning of step 2.

Each of the two approaches has its own advantages. For example, on one hand, busy waiting is more efficient if the signal \( s \) is present at each instant but \( act \) is often absent. On the other hand, the version with passive waiting is more efficient if the activation signal \( act \) is always present and if \( s \) is often absent.

6. A Parallel Runtime

In this section, we consider the parallel execution of the ReactiveML runtime as opposed to the concurrent but sequential execution provided by the cooperative scheduler of the preceding sections. We now wish to execute a ReactiveML program on a multi-processor to improve the performance but without modifications of programs and transparently for users.

Tools and language

ReactiveML is built on an OCaml kernel and, unfortunately, the current version of OCaml is not suitable for parallel execution of the runtime. Indeed, even though it is possible to create threads with OCaml (with the Thread module of the standard library), a global lock prevents multiple threads from executing simultaneously.\(^8\) We thus decided to use F#\(^9\) in our experiments. This language is very similar to OCaml and the source code generated by the ReactiveML compiler was already mostly compatible with F#.

Implementation

As already described, the ReactiveML runtime is based around a set \( C \) of continuations awaiting execution. The basic idea of the parallel runtime is to have several threads taking and executing continuations from \( C \) in parallel. We only parallelize executions during an instant; the end of instant reaction remains sequential.

To implement the set \( C \), we use a concurrent data structure that permits work stealing [21]. Each thread has its own set of continuations to execute, implemented by a double-ended queue or deque. This data structure minimizes conflicts between thieves and the owner of the deque. The algorithm can be implemented very efficiently without locks, using only atomic operations like compare-and-swap [16]. Our implementation is based on the ConcurrentBag class of the .NET standard library.

Since threads communicate through shared memory, we can reuse most of the code from the sequential runtime. The major changes are in taking care of concurrent access to the shared data structures. Basically, we associate a lock to each shared data structure, but some optimizations are possible:

- We decrement the counters associated to parallel compositions using atomic operations of the class Interlocked.
- We use lock-free data structures for shared lists. For example, we use the class ConcurrentStack to implement the next sets of control tree nodes.
- Locks are unnecessary if data races cannot occur. This is the case for signal values, which are only modified during the end of an instant (which is sequential). During an instant, signal values are read-only.

We use locks for the nodes of the control tree, but they are only modified occasionally. Locks are also used for signals. Indeed, during the emission of a signal, we must atomically modify the status of the signal and wake processes up.

Experimental results

Figure 4 shows the performance of the parallel runtime in F# on some typical ReactiveML programs from the standard distribution of the language. The benchmark was run on a machine with two Xeon 5150 processors each with two cores running at 2.66 Ghz. We used Windows 7 and Visual Studio 2012 Ultimate. The programs run with as many threads as cores, that is, 4, and are compiled with optimization for 64 bit architectures. For each program, we measure the total execution time for several instants.

The upper graph shows the speedup of the parallel runtime compared to the sequential version in F#, that is the ratio of computation time between the parallel and sequential versions. The first two examples are simulations of cellular automata [11] and the flocking behavior of birds (boids) [31]. The last three examples are simulations of the n-body problem with varying numbers of planets to test the ratio between computations and synchronizations. We obtain good speedups, the parallel version is approximately 2 times faster than the sequential one except for the cellular automata example and when there are very few planets in the n-body simulation. The

---

\(^8\) See, for example, section 19.10.2, “Parallel execution of long-running C code”, of the OCaml manual [22]

\(^9\)http://fsharp.org
poor result with 10 planets is not surprising since little computation is required. For the cellular automata, each process also performs very little computation and there is also a lot of memory allocation (a closure is allocated for each pause). This is a problem because the .NET garbage collector suspends all threads.

The lower graph compares the speedups of the sequential and parallel F# runtimes compared to the native OCaml version. The sequential F# version is always slower than the OCaml one. For example, it is almost 2 times slower for the cellular automata and boids. Parallelism overcomes this slowdown, but does not provide great benefits compared to the native OCaml version. These experiments suggest that the parallel runtime is not very useful in practice due to the limited liberty of processes between successive resynchronizations.

To solve this issue, we have introduced reactive domains [26]. It provides a new programming construct that allows to compute locally several steps. Therefore, two reactive domains composed in parallel synchronized less often. A type system ensures that these systems communicate only on a slow rate.

7. Related Implementations

The closest related work to ReactiveML is in the Junior [20] and SugarCubes [14] Java libraries, which are based on the same reactive model of concurrency [10]. In fact, the implementation of ReactiveML was particularly inspired by Laurent Hazard’s ‘Simple’ implementation of Junior [19]. One of the main differences between ReactiveML and other implementations of the reactive model is that ReactiveML departs from the hierarchical structure of parallelism and introduces the notion of control tree. Moreover, the other implementations are deep embeddings where the abstract syntax tree is explicitly represented and interpreted.

Compared to Fair Threads [12], which are also based on the reactive model, ReactiveML does not support the Globally Asynchronous Locally Synchronous (GALS) model. But the synchronous subset of Fair Threads is less expressive than ReactiveML. In particular, there is no hierarchical structure of parallel composition.

C. Deleuze describes the implementation of an OCaml library of cooperative concurrency based on ReactiveML’s constructs [17]. It can be seen as an alternative implementation of the ReactiveML runtime. The main difference is in the handling of preemption and suspension. In Deleuze’s implementation, a stack describing the evaluation context is associated to each thread. This stack is the root of the control tree. The drawback of this implementation is that these stacks must be traversed at the end of every instant. Experimental results [17, section 5] comparing implementations of the library with and without preemption and suspension, and show that these control structures have a strong performance impact. This is why it is difficult to properly evaluate the efficiency of ReactiveML with less expressive libraries of cooperative concurrency like Lwt [34] or Async.

ReactiveML, like other members of the reactive family, restricts the synchronous model of Esterel to give a more dynamic and supple language. In particular, ReactiveML is implemented using dynamic scheduling, whereas Esterel programs are statically scheduled. This has runtime costs (in the maintenance of scheduling data structures, and delays in reading signal values and reacting to absence), but simplifies analysis and compilation and eases dynamic process creation. In the first versions of Esterel, programs were compiled into automata [8], but the size of the generated code could be exponentially larger than the size of the source code. The translation of Esterel into equations [4], used in the later versions, avoids this combinatorial explosion and allows to generate both hardware and software from the same source code. Other approaches dedicated to the generation of software offer better performance [29].

More recently, the language HipHop [9] embeds Esterel into Hop, a language dedicated to web programming. The runtime builds an abstract syntax tree which is dynamically interpreted following the rules of the constructive semantics of Esterel [6].

8. An Historical Perspective

Finally, we discuss the design, semantics and implementation choices made for ReactiveML, those that have been successful and time resistant, and those that failed.

Choosing the Host Language

The ReactiveML project started in 2002 based on two preliminary results.

The first was the close relationship, observed by Caspi and Pouzet [15], between Wadler’s deforestation technique [35], for compiling away certain intermediate data structures in Haskell programs, and the so-called synchronous constraints of Lustre [18] that guarantee the removal of intermediate streams during compilation. Extending the synchronous constraints and compilation techniques to a wider class of functional programs promised to be use-

![Figure 4: Experimental results of the parallel runtime in F# (4 threads, 2 processors with 2 cores each, F# 3.0)](image-url)
ful in two ways: (1) to conservatively extend Lustre with ML features to increase its modularity and expressiveness, while remaining relevant for real-time control software where target code must be statically scheduled to execute in bounded time and memory; (2) to embed synchronous principles in an existing ML language for programming less constrained reactive applications for which the full expressiveness of the host language is needed. The language Lucid Synchrone [30] adopted the first approach and several of its programming constructs, compilation techniques, and dedicated type systems, influenced by ML, are incorporated in the industrial language and compiler of Scade 6. We failed to follow the second approach in the data-flow model because it proved impossible to generalize the clock calculus and causality analysis of Lustre to all OCaml programs without rejecting most of them and while still keeping, at least, the expressiveness of Lustre. The clock calculus and causality analysis guarantee the absence of deadlocks and memory leaks. Functional Reactive Programming (FRP) [36] extends a functional language with signals and signal functions but suffers from possible deadlocks and memory leaks. So, for ReactiveML, we took the second approach but adopted the programming style of Esterel that appeared to mesh more naturally with the imperative characteristics of ML than did the data-flow style of Lustre.

The second influence for ReactiveML was the pioneering work of Boussinot on ReactiveC [10] and the SugarCubes library [14] in Java. Boussinot showed that the synchronous model could be relaxed and a causality analysis avoided by delaying the reaction to signal absence by one instant. Yet, we failed to give a simple semantics to SugarCubes. In particular, we did not manage to express the communication between the host language and the reactive kernel. Moreover, due to dynamic binding for signals, the language interacted poorly with Java’s garbage collector. We thus decided to switch to static binding for signals and to experiment with Boussinot’s ideas in ML. The type system was essentially the one of ML and we designed the well formation analysis shown in Figure 1 to distinguish reactive expressions from ML ones. This simple separation never appeared too restrictive. By making signals first-class citizens, it became possible to have them carry reactive (coopera-
tive) processes can be mixed with preemptive OCaml threads. This is very useful for blocking input and output function, for example, but also when calling possibly long running OCaml functions. The library Rml_async eases the use of preemptive threads inside a ReactiveML program. For example, the function Rml_async.proc_of_fun creates a process that launches a given function on an OCaml thread. Using this function, we can define a process that repeats on its standard output what it reads on its standard input:

```
let process echo =  
  let read = Rml_async.proc_of_fun read_line in
  loop
    let s = run read () in
    print_endline s; pause
  end
```

In this example, the function read_line is executed in a separate thread. The whole process echo cooperates properly and the static analysis of [25] does not complain.

To help avoid such bugs, ReactiveML provides a type-based static analysis that warns about potentially instantaneous loops or recursions [25]. For the previous example, we have:

```
let process error s =  
  try
    emit s () || raise Exit
  with Exit -> ()
```

```
let process reactivity_issue =  
  loop () end ||
  pause; print_endline "Shy"
```

Choosing between a Language or an Embedded DSL

ReactiveML is based on the synchronous model which provides deterministic parallelism that facilitates debugging and gives a precise meaning to signal absence, suspension and preemption. Yet, the model is cooperative, meaning that uncooperative processes can block the execution of the whole system. For example, in the following program, the message "Shy" is never printed because the first instant never terminates.

To help avoid such bugs, ReactiveML provides a type-based static analysis that warns about potentially instantaneous loops or recursions [25]. For the previous example, we have:

```
let process error s =  
  try
    emit s () || raise Exit
  with Exit -> ()
```

```
let process reactivity_issue =  
  loop () end ||
  pause; print_endline "Shy"
```

```
let process echo =  
  let read = Rml_async.proc_of_fun read_line in
  loop
    let s = run read () in
    print_endline s; pause
  end
```

In this example, the function read_line is executed in a separate thread. The whole process echo cooperates properly and the static analysis of [25] does not complain.

A noticeable characteristic of ReactiveML is that (cooperat-
ive) processes can be mixed with preemptive OCaml threads. This is very useful for blocking input and output function, for example, but also when calling possibly long running OCaml functions. The library Rml_async eases the use of preemptive threads inside a ReactiveML program. For example, the function Rml_async.proc_of_fun creates a process that launches a given function on an OCaml thread. Using this function, we can define a process that repeats on its standard output what it reads on its standard input:

```
let process echo =  
  let read = Rml_async.proc_of_fun read_line in
  loop
    let s = run read () in
    print_endline s; pause
  end
```

In this example, the function read_line is executed in a separate thread. The whole process echo cooperates properly and the static analysis of [25] does not complain.

10 http://www.esterel-technologies.com/products/scade-suite
11 Parallel composition || has lower priority than sequential composition ;
Finally, there is the question of whether we should have provided basic primitives to suspend and preempt processes. The gain in expressiveness with respect to libraries of cooperative threads is clear but there is impact on the efficiency and complexity of the implementation. The success of cooperative thread libraries may suggest that the benefit would be limited. The point is that when control structures are necessary, the programmer using cooperative thread libraries has to implement them manually which has impact on performance and increase the risk of bug. Being able to precisely start, kill and suspend a process is precisely the essence of synchronous programming.

**ReactiveML is not Esterel**

ReactiveML is based on the relaxed model of Boussinot where reaction to signal absence is delayed by one instant. The advantage is that all programs are causal-by-construction and so causality analysis is not required. For example, the following program is statically rejected in Esterel because the signal `s` would have to be both present and absent during the same instant, which is incoherent: $\text{signal } s \text{ in present } s \text{ then } () \text{ else emit } s$

This program is, however, perfectly valid in the relaxed model of Boussinot and thus also in ReactiveML: the test for absence is not instantaneous. The signal `s` being absent at the first instant, it is emitted at the second instant. Awaiting a valued signal also takes a tick. In the following program, for example, the value of `s` is `1` at the first instant and `2` at the second instant: $\text{signal } s \text{ default } 0 \text{ gather } (+) \text{ in emit } s \equiv 1 || \text{ await } s(x) \text{ in emit } s \equiv (x + 1)$

Executing `emit s (x+1)` instantaneously with `await s(x)`, as in Esterel, would give an incoherent value for `s`.

Nonetheless, reacting with a delay compromises modularity. Consider the statement: $\text{await } s(x) \text{ in print_int } x$

which is not equivalent to: $\text{signal } 1 \text{ default } 0 \text{ gather } (+) \text{ in await } s(x) \text{ in emit } 1 \equiv x || \text{ await } l(y) \text{ in print_int } y$

This means that replacing a function by the parallel composition of two functions, and conversely, is not possible in general. Delays may accumulate when communicating valued signals. Reactive domains [26] resolve this problem by allowing time refinement, which hides some internal steps of processes. This way, the successive steps of a process may be seen by its environment as a single step, and that process may then be replaced by one that really only does take a single step.

### 9. Conclusion

Since its creation, ReactiveML has been used in domains that we did not even imagine when it was designed.

The first was in the domain of networks, and, in particular, the design of packet routing protocols for mobile ad-hoc networks [24]. Such networks are made up of hundreds of geo-localized computers that intercommunicate through radio signals. The standard simulation tools NS and OPNET were unable to simulate networks of this size and programming a simulator in C proved to be unexpectedly difficult. It was possible, however, to program the first complete model of the network and routing protocol in ReactiveML in only a few days and to provide all the simulation metrics of the protocol.

13Recall that `signal s default 0 gather (+)` signifies that multiple values emitted simultaneously on a are to be added together.

After this experiment, Florence Maraninchi and her colleagues at VERIMAG, made extensive use of ReactiveML for designing low power consumption sensor networks [33]. Both the model of the internal controller of a sensor and the whole simulation involving tens of thousands of them were programmed in ReactiveML.

More recently, ReactiveML was used to implement ReactiveAsco [2], a shallow embedding of the Antescofo language for score following and mixed music developed at IRCAM. ReactiveML was used to prototype new programming constructs. This showed that Antescofo shared many features with synchronous languages, like the synchronous interpretation of time and control structures that could be encoded in ReactiveML. Surprisingly for us, the efficiency of the ReactiveML interpreter compared well with the one of Antescofo: on all musical pieces we tested, response times were lower than the reaction time of the human ear. After this experiment, Arias et al. have used ReactiveML for defining and implementing interactive scores [1].

The extension of ReactiveML with reactive domains [26] provides additional expressiveness to the language and thus may allow the programming of new classes of applications. Yet, the type system for this extension is probably too complex to be used routinely. Reactive domains ensure that communications stay local to a process and reduce the number of synchronizations between processes. They can serve as a good model for a parallel implementation. It can even be used as a deterministic model for the programming of distributed systems. A challenge is that an efficient implementation relies on a concurrent or distributed garbage collector.

### Acknowledgments

We would like to thank the PPDP Steering Committee and Elvira Albert for inviting us to write this article. The development of ReactiveML started at Université Paris 6 in the team of Thérèse Hardin who provided constant advice and support. ReactiveML benefited greatly from the discussions with colleagues in the successive places where it has been developed: the Verimag laboratory in Grenoble, INRIA (Rocquencourt), University of Paris-Sud (Orsay), ENS and now IBM Research. ReactiveML would not have been possible without the pioneering work of Frédéric Boussinot. His generous and enlightening explanations have been extremely useful. Guillaume Baudart and Timothy Bourke provided numerous comments and helpful suggestions for the present article and many others. Finally, we warmly thank all the ReactiveML programmers, users and developers.

### References


14[http://reactiveml.org/reactive_ascoco](http://reactiveml.org/reactive_ascoco)

15[http://repmus.ircam.fr/antescofo](http://repmus.ircam.fr/antescofo)