ReactiveML, Ten Years Later

Louis Mandel
IBM Research
lmandel@us.ibm.com

Cédric Pasteur
École normale supérieure
(now at ANSYS-Esterel Technologies)
cedric.pasteur@ansys.com

Marc Pouzet
École normale supérieure
marc.pouzet@ens.fr

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]

[1] `http://caml.inria.fr/pub/docs/manual-ocaml-4.02/libthreads.html`

[2] `http://ocsigen.org/lwt`

[3] `https://ocaml.janestreet.com/ocaml-core/111.28.00/doc/async/`

[4] `http://christophe.deleuze.free.fr/muthreads`

[5] `http://erratique.ch/software/react`

[6] `http://www.camlcity.org/archive/programming/equeue.html`

PPDP '15, July 14 - 16, 2015, Siena, Italy.
Copyright © 2015 ACM 978-1-4503-3516-4/15/07…$15.00.
http://dx.doi.org/10.1145/2790449.2790509

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.

[7] `http://icfpc2013.cloudapp.net`

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 has 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 `guess` 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:

```
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 `guess` 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` (the `run` keyword indicates the execution of a process). To limit the number of requests sent to the server, this process `increase_spec` (which is supposed to send a request per instant) is executed inside a `do`/`when` control structure. The `do`/`when` executes its body only during instants where the control signal `tick` is emitted. The loop at line 16 generates the signal `tick` every 4 seconds (the process `wait d` does nothing during `d` seconds and then terminates). At lines 10 and 11, the signal `tick` is declared with a combination function that always returns `()`, the only value of type `unit`.

```
1  let process communicator pb spec guess finished =
2    loop
3      await guess (f) in
4      match Webapi.guess pb.id f with
5      | Guess_win -> emit finished ()
6      | Guess_mismatch (input, output) ->
7          emit spec ([|input|], [|output|])
8    end
9    ||
10   signal tick default ()
11                gather (fun x y -> ()) in
12   do
13     run increase_spec pb spec
14   when tick done
15   ||
16   loop emit tick (); run wait 4.0 end
 val communicator: problem -> (spec_t, 'b) event ->
   ('a, program option) event -> (unit, 'c) 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.

```
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 $\mathcal{L}_k$. The translation from ReactiveML to $\mathcal{L}_k$ 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:

$$
\begin{aligned}
e ::= \;& x \mid c \mid (e,e) \mid \lambda x.e \mid e\;e \mid \texttt{rec}\;x = e \mid \texttt{process}\;e \mid \texttt{run}\;e\\
&\mid \texttt{pause} \mid \texttt{let}\;x = e\;\texttt{and}\;x = e\;\texttt{in}\;e \mid e;e\\
&\mid \texttt{signal}\;x\;\texttt{default}\;e\;\texttt{gather}\;e\;\texttt{in}\;e \mid \texttt{emit}\;e\;e\\
&\mid \texttt{await immediate}\;e \mid \texttt{await}\;e(x)\;\texttt{in}\;e\\
&\mid \texttt{present}\;e\;\texttt{then}\;e\;\texttt{else}\;e
\end{aligned}
$$

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_1 `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 ReactiveC [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`:

$$e_1 \,||\, e_2 \triangleq \texttt{let}\ \_ = e_1\ \texttt{and}\ \_ = e_2\ \texttt{in}\ ()$$

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

$$\texttt{let process}\ f\,x = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = \lambda x.\texttt{process}\ e_1\ \texttt{in}\ e_2$$

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

This kernel is not minimal, it contains, for instance, the construct $e_1; e_2$ that can be encoded as `let _ =` e_1 `in` e_2. We made this choice mainly for pedagogical reasons, but also to demonstrate that some constructs can be implemented directly more efficiently.

The $\mathcal{L}_k$ intermediate language

The $\mathcal{L}_k$ language is formally defined by:

$$\begin{aligned}
k ::= {} & \texttt{end} \mid \kappa \mid e_i.k \mid \texttt{present}\ e_i\ \texttt{then}\ k\ \texttt{else}\ k \mid \texttt{run}\ e_i.k \\
& \mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ k \\
& \mid \texttt{await immediate}\ e_i.k \mid \texttt{await}\ e_i(x)\ \texttt{in}\ k \\
& \mid \texttt{split}\ (\lambda x.(k,k)) \mid \texttt{join}\ x\ i.k \\
& \mid \texttt{def}\ x\ \texttt{and}\ x\ \texttt{in}\ k \mid \texttt{bind}\ \kappa = k\ \texttt{in}\ k \\
e_i ::= {} & x \mid c \mid (e_i, e_i) \mid \lambda x.e_i \mid \texttt{rec}\ x = e_i \mid \texttt{process}\ \Lambda\kappa.k \\
& \mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ e_i \mid \texttt{emit}\ e_i\ e_i
\end{aligned}$$

The $\mathcal{L}_k$ language distinguishes continuations k from the instantaneous expressions e_i that correspond to ML expressions. The continuation `end` represents the end of the program. κ is a variable that can be substituted by a continuation k. The expression $e_i.k$ evaluates the instantaneous expression e_i before passing the computed value to the continuation k. The expressions `split`, `join` and `def` allow the encoding of the synchronous parallel definition `let/and/in`. The construct `split` starts the parallel execution, `join` synchronizes the termination of two branches and `def` binds the values computed by two branches (provided by the `join`) before executing its body. The variable x introduced by `split` and used in `join` is shared between the two branches and used to synchronize them. The parameter i in the `join` can only be 1 or 2. It is used to distinguish the left branch ($i = 1$) of the parallel from the right branch ($i = 2$). The construct `bind` names a continuation. The instantaneous expressions e_i are similar to those of ReactiveML. The only difference is that the definition of a process takes its continuations κ as arguments introduced by the binder Λ.

Identification of instantaneous expressions

To translate ReactiveML code to $\mathcal{L}_k$, the compiler must distinguish instantaneous expressions from reactive ones; it does this using the type system introduced in [28] and that guarantees a well-formedness property. Reactive expressions can only be used within processes. In particular, reactive expressions such as `pause` are forbidden in the bodies of functions. The CPS transformation therefore concerns only processes and functions are unchanged. This benefits the performance of generated code since instantaneous expressions are translated directly into OCaml and executed without overhead.

The analysis is defined in figure 1 by a judgment of the form $\gamma \vdash e$ where $\gamma \in \{0, 1\}$. The predicate $0 \vdash e$ means that e is an instantaneous expression. The predicate $1 \vdash e$ means that e is a reactive expression. The choices made in the design of this type system are discussed in [28]. We just recall some important points:

- $\gamma \vdash e$ means that both judgments $0 \vdash e$ and $1 \vdash e$ are valid. The expression can thus be used in any context (that is in an instantaneous expression or a reactive expression). This is the case for example for variables, constants or applications.
- The body of a function must be instantaneous (ABS rule), but the body of a process can be a reactive expression (PROCABS rule).

Translation from ReactiveML to $\mathcal{L}_k$

The translation to $\mathcal{L}_k$ is defined by a function $C_k[e]$ parameterized by a continuation k. It takes as argument a ReactiveML expression e and returns a continuation in the $\mathcal{L}_k$ language. This function is defined figure 2. It uses the function $C[e]$ to translate instantaneous expressions. This $C[e]$ function does not take a continuation as argument since the CPS transformation only affects reactive expressions. The translation function $C_k[e]$ is defined figure 2a:

- If the expression e is instantaneous, that is, if $0 \vdash e$, then we must not apply the CPS transformation. We translate it as an instantaneous expression $C[e]$ that gives its value to the continuation k.
- There is no sequence composition in $\mathcal{L}_k$, it is encoded with continuations. In the case of $e_1; e_2$, we translate first e_2 with the continuation k. Then we translate e_1 with the translation of e_2 as continuation.
- For `present`, we use the `bind` construct to define the continuation shared by the two branches. Sharing is necessary to avoid the duplication of continuations and the risk of generating code of exponential size.
- For the translation of binders like `await/in`, `signal/in` and `let/and/in`, we must not capture a variable used in the continuation ($fv(k)$ the set of free variables in k).
- The translation of `let/and/in` uses `split` to trigger execution of the two branches. The continuation of each branch is a `join` expression that waits for the termination of the other branch. The continuation of the two `join`'s is a `def` expression that instantaneously receives the values computed in both branches. We use a `bind` to share κ between the branches.

The translation of instantaneous expressions is defined in figure 2b by a structural application of the translation function that finds processes definitions. The translation of a process definition introduces a parameter κ and translates the body of the definition using the function $C_\kappa[.]$, that is, using κ as the continuation of the process body.

Translation to OCaml

The translation of $\mathcal{L}_k$ into OCaml is immediate. We associate each construct of $\mathcal{L}_k$ with an OCaml function (a combinator). We represent a continuation by a value of type `'a step = 'a -> unit`, that is, a function that waits for a value of type `'a` from the preceding computation. The evaluation of instantaneous expressions is controlled by representing them using closures of type `unit -> 'a`. The continuation $e.k$ is thus represented by the `rml_compute` combinator:

```
let rml_compute e k = (fun _ -> k (e ()))
val rml_compute: (unit -> 'a) -> 'a step -> 'b step
```

The arguments of this combinator are an instantaneous expression `e` and a continuation `k`. It returns a step function that evaluates the expression `e` by applying the value `()` and passes the result to the continuation `k`. The combinators for the other expressions have similar shapes:

```
val rml_pause: unit step -> 'a step
val rml_present:
 (unit -> ('a, 'b) event) -> unit step -> unit step
  -> 'c step
```

$$\frac{}{\gamma \vdash x} \qquad \frac{}{\gamma \vdash c} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{\gamma \vdash (e_1, e_2)} \qquad (\textsc{Abs})\frac{0 \vdash e_1}{\gamma \vdash \lambda x.e_1} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{\gamma \vdash e_1\ e_2} \qquad \frac{0 \vdash e_1}{\gamma \vdash \texttt{rec}\ x = e_1}$$

$$(\textsc{ProcAbs})\frac{1 \vdash e_1}{\gamma \vdash \texttt{process}\ e_1} \qquad \frac{0 \vdash e_1}{1 \vdash \texttt{run}\ e_1} \qquad \frac{}{1 \vdash \texttt{pause}} \qquad \frac{\gamma \vdash e_1 \quad \gamma \vdash e_2 \quad \gamma \vdash e_3}{\gamma \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3} \qquad \frac{\gamma \vdash e_1 \quad \gamma \vdash e_2}{\gamma \vdash e_1; e_2}$$

$$\frac{\gamma \vdash e_1 \quad \gamma \vdash e_2 \quad \gamma \vdash e}{\gamma \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e} \qquad \frac{0 \vdash e_s \quad 0 \vdash e_1}{\gamma \vdash \texttt{emit}\ e_s\ e_1} \qquad \frac{0 \vdash e}{1 \vdash \texttt{await immediate}\ e} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{await}\ e_1(x)\ \texttt{in}\ e_2}$$

$$\frac{0 \vdash e_s \quad 1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

Figure 1: Well formation of expressions

$$C_k[e] = C[e].k \quad \text{if} \quad 0 \vdash e \qquad C_k[\texttt{run}\ e] = \texttt{run}\ C[e].k \qquad C_k[e_1; e_2] = C_{C_k[e_2]}[e_1]$$

$$C_k[\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2] = \texttt{bind}\ \kappa = k\ \texttt{in present}\ C[e]\ \texttt{then}\ C_\kappa[e_1]\ \texttt{else}\ C_\kappa[e_2] \text{ where } \kappa \text{ is fresh}$$

$$C_k[\texttt{await immediate}\ e] = \texttt{await immediate}\ C[e].k \qquad C_k[\texttt{await}\ e_1(x)\ \texttt{in}\ e_2] = \texttt{await}\ C[e_1](x)\ \texttt{in}\ C_k[e_2] \text{ where } x \notin fv(k)$$

$$C_k[\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e] = \texttt{signal}\ x\ \texttt{default}\ C[e_1]\ \texttt{gather}\ C[e_2]\ \texttt{in}\ C_k[e] \text{ where } x \notin fv(k)$$

$$C_k[\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e] = \texttt{bind}\ \kappa = (\texttt{def}\ x_1\ \texttt{and}\ x_2\ \texttt{in}\ C_k[e])\ \texttt{in}$$
$$\texttt{split}\ (\lambda y.(C_{\texttt{join}\ y\ 1.\kappa}[e_1], C_{\texttt{join}\ y\ 2.\kappa}[e_2])) \text{ where } y \text{ is fresh and } x_1, x_2 \notin fv(k)$$

(a) Translation of reactive expressions

$$C[x] = x \qquad C[c] = c \qquad C[(e_1, e_2)] = (C[e_1], C[e_2]) \qquad C[e_1\ e_2] = C[e_1]\ C[e_2] \qquad C[\lambda x.e] = \lambda x.C[e]$$

$$C[\texttt{rec}\ x = e] = \texttt{rec}\ x = C[e] \qquad C[\texttt{process}\ e] = \texttt{process}\ \Lambda\kappa.C_\kappa[e]$$

$$C[\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e] = \texttt{signal}\ x\ \texttt{default}\ C[e_1]\ \texttt{gather}\ C[e_2]\ \texttt{in}\ C[e] \qquad C[\texttt{emit}\ e_1\ e_2] = \texttt{emit}\ C[e_1]\ C[e_2]$$

(b) Translation of instantaneous expressions

Figure 2: Translation from ReactiveML to $\mathcal{L}_k$

```
val rml_await_immediate:
 (unit -> ('a, 'b) event) -> unit step -> 'c step
val rml_await_all:
 (unit -> ('a, 'b) event) -> ('b -> unit step)
  -> 'c step
...
```

The implementation of these combinators is the subject of the following sections.

4. Sequential Runtime

We now describe the OCaml implementation of the runtime which realizes the semantics of the $\mathcal{L}_k$ 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 $\mathcal{C}$ 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 $\mathcal{C}$ 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 $\mathcal{C}$ for execution at the next instant. The processes in the set *next* are transferred into $\mathcal{C}$ 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 ev 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

on the signal presence. The continuations in these lists are only woken up when the signal is emitted.

- The other important feature, which is also the most complex, is the handling of preemption (`do`/`until`) and suspension (`do`/`when`). The use of the set of continuations $\mathcal{C}$ disregards the hierarchical structure of the program. This allows for good performance, but an additional data structure is needed to track processes to kill or suspend. We describe it in section 5.

Combinators

The various combinators are implemented via a limited number of scheduling primitives defined in a module called `Runtime`. The separation between the definition of the combinators and the core primitives allows some code to be shared between the sequential and parallel implementations presented in section 6.

Figure 3 presents a snippet of the interface of the `Runtime` module. Recall that the type of a continuation that waits for a value of type `'a` is `'a step = 'a -> unit`.

- `on_current_instant` schedules a continuation for execution in the current instant. It corresponds to adding the continuation to the set $\mathcal{C}$.
- `on_next_instant` schedules a continuation for execution during the next instant. It corresponds to adding the continuation to the set $next$. This is the behavior of the `pause` combinator.
- `on_eoi` executes a continuation during the end of instant. It is used for example to get the value of a signal at the end of instant.
- `on_event` executes a continuation when a signal is emitted. This is the behavior of `await immediate`.
- `on_event_or_next` takes as arguments two continuations. It executes the first one if the signal is present during the current instant and the second one during the next instant if the signal is not emitted in the current instant. It is used, for instance, to implement the `present` combinator.

The implementation of some combinators becomes trivial using these primitives. For example, the construct `await immediate` $e.k$ is implemented by the combinator `rml_await_immediate` that uses the primitive `Runtime.on_event` directly:

```
let rml_await_immediate expr_evt k _ =
  Runtime.on_event (expr_evt()) k ()
```

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 on_event_at_eoi evt f =
  Runtime.on_event evt (fun () -> Runtime.on_eoi f)
```

When `evt` is emitted, we add `f` to the set of continuations to execute at the end of the instant. We can now implement the combinator `rml_await_all` for the construct `await` $e(x)$ `in` k that binds the value of signal e to variable x in continuation k:

```
let rml_await_all expr_evt k _ =
  let evt = expr_evt () in
  let await_eoi _ =
    let v = Runtime.Event.value evt in
    Runtime.on_next_instant (fun () -> k v)
  in
  on_event_at_eoi evt await_eoi
```

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 $\mathcal{L}_k$. 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 jp` where `jp` 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:

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

Signal handling

As mentioned earlier, busy waiting on signals must be avoided for the sake of efficiency: waiting for a signal should not cost anything while the signal is absent. We achieve this by associating two waiting lists to each signal. A signal is thus a tuple `(n, wa, wp)` where `n` is a data structure containing information on the signal (its status, the emitted values, the gathering function, etc.), `wa` contains the processes waiting for the emission of the signal, and `wp` contains the processes testing the instantaneous presence of the signal with the `present` construct. Processes in these two waiting lists are woken up when the signal is emitted. Processes in `wp` are also woken up at the end of instant if the signal was absent.

The behavior is realised by the primitives `on_event` and `on_event_or_next`. The `on_event` primitive is implemented as:

```
let on_event (n, wa, wp) f =
  if Event.status n then f () else wa := f :: !wa
```

It first tests if the signal is present, that is, if it has already been emitted during the instant. If this is the case, then it executes the function `f` immediately. Otherwise, it puts `f` in the list `wa` of functions to execute when the signal is emitted.

The `on_event_or_next` primitive is implemented as:

```
let on_event_or_next (n, wa, wp) f_ev f_n =
  let act () =
    if is_eoi () then next := f_n :: !next
    else f_ev ()
  in
  if Event.status n then f_ev ()
  else (wp := act :: !wp; weoi := wp :: !weoi)
```

If the signal is present, the primitive calls the function `f_ev` immediately. Otherwise, it adds the `act` function to the `wp` list and it adds this list `wp` to the global list `weoi` (which is a list of references to lists of functions). All the functions accessible from `weoi` at the end of the instant are woken up. The `act` function tests if the function is executed during the instant or at the end of instant by calling `is_eoi`. If it is during the instant, it means that the process has been awakened by emission of the signal. Thus it immediately calls `f_ev`. If the function is executed during the end of the instant, it means that the signal has not been emitted during the instant and is therefore absent. In this case, the function `f_n` is added to the list *next* of processes to execute during the next instant. Notice that it is not necessary to remove `wp` from the list `weoi` when the signal is emitted since in this case `wp` contains the empty list. Similarly, there is no problem if `wp` is added more than once to `weoi`.

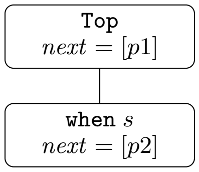
5. Suspension and preemption

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 $\mathcal{L}_k$ with preemption and suspension, is given in [23].

Principles

As mentioned in section 4, the use of the set $\mathcal{C}$ 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:

```
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 $\mathcal{C}$, 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 $C_k[e]$ (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 $\mathcal{L}_k$ 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 $\mathcal{C}$ 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:

```
1 let process await_when =
2   signal act default () gather fun x y -> () in
3   signal s default () gather fun x y -> () in
4   do
5     await immediate s; print_endline "Received!"
6   when act done
7   || loop emit act (); pause; pause end
8   || pause; emit s (); pause; emit s ()
```

This process waits for the emission of `s` 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 `evt` is emitted and the control node `ctrl` is active:

1. If the signal is present when we start waiting, we can call `f` directly. Indeed, we know that `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 `ctrl` later in the same instant (in the previous example, `s` can be emitted before `act`). If `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 `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 multiprocessor 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 $\mathcal{C}$ of continuations awaiting execution. The basic idea of the parallel runtime is to have several threads taking and executing continuations from $\mathcal{C}$ in parallel. We only parallelize executions during an instant; the end of instant reaction remains sequential.

To implement the set $\mathcal{C}$, 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`

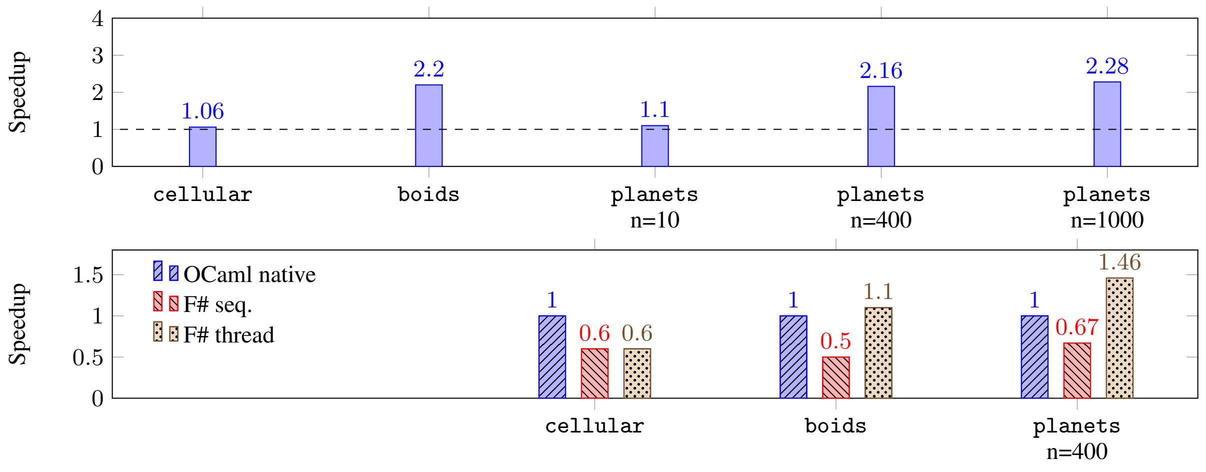


Figure 4: Experimental results of the parallel runtime in F# (4 threads, 2 processors with 2 cores each, F# 3.0)

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 comunicate 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 list of control nodes representing the path from the process to 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 useful 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.[10] 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 features 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 ML and reactive functions, for example, to model dynamically reconfigurable systems. This features has been used to program an interactive debugger for ReactiveML in ReactiveML itself [27].

The efficient runtime is crucial for applications with tens of thousands of reactive threads and where the number of active ones varies dynamically (imagine the simulation of a burning forest where every tree is a process [11] or the simulation of a sensor network with sensors whose batteries may run out [33]). Finally, choosing OCaml as the target language simplified the implementation by allowing the use of continuations. Still, we believe that the same ideas could be incorporated into other languages.

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.[11]

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

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:

```
Line 2, characters 2-13:
Warning:
    This expression may be an instantaneous loop.
```

Being able to define dedicated compile time analyses was an important motivation for building a language rather than defining an embedding in OCaml or a library. The interaction with imperative features from the host language, like exceptions, is also better controlled. In the Async library, for example, the `try`/`with` construct of OCaml does not catch exceptions raised by a lightweight thread and thus a program using this construct fails at runtime. A dedicated `try_with` construct must be used instead.[12] Such programs do not type check in ReactiveML.

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

```
Line 3, characters 4-27:
This expression must be instantaneous.
```

Notice that this example is not deterministic if raising an exception can immediately interrupt the enclosing block. The `do`/`until` construct supersedes `try`/`with`.

The language approach also facilitates compiler optimisations. Nonetheless, ReactiveML only implements simple ones like the fusion of parallel constructs and rewriting. Finally, ReactiveML programs are easier to read and write than ones expressed directly as continuations (which resemble the code generated by ReactiveML). On the other hand, implementing a language requires much more work, and making ReactiveML an extension of the full and evolving OCaml language, is too big an effort. We thus decided to develop an independent compiler that generates OCaml source code rather than to maintain a fork of the OCaml compiler. The advantage is that we are much more independent of the evolution of the host language and our compiler is much smaller. It provides us with a good research platform. The disadvantage is that ReactiveML only provides a subset of OCaml (functors, objects, and GADTs are not, for instance, supported).

A noticeable characteristic of ReactiveML is that (cooperative) 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 `;`.

[12] See Chapter 18, "Exception Handling", in the book *Real World OCaml*: `https://realworldocaml.org/v1/en/html/concurrent-programming-with-async.html`.

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:

```
signal s in present s then () 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.[13]

```
signal s default 0 gather (+) in
emit s 1 ||
await s(x) in emit s (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:

```
await s(x) in print_int x
```

which is not equivalent to:

```
signal l default 0 gather (+) in
await s(x) in emit l x
||
await l(y) 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 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.

[13] Recall that `signal s default 0 gather (+)` signifies that multiple values emitted simultaneously on `s` 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],[14] a shallow embedding of the Antescofo[15] 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 interpretor 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 an 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

[1] J. Arias, M. Desainte-Catherine, S. Salvati, and C. Rueda. Executing hierarchical interactive scores in ReactiveML. In *Journées d'Informatique Musicale*, 2014.

[2] G. Baudart, F. Jacquemard, L. Mandel, and M. Pouzet. A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music. In *Thirteen International Conference on Embedded Software*, 2013.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[4] G. Berry. Esterel on hardware. In *Mechanized reasoning and hardware design*, pages 87–104. Prentice-Hall, Upper Saddle River, NJ, 1992.

[5] G. Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, 1993.

[6] G. Berry. The constructive semantics of pure Esterel, 1996.

[14] `http://reactiveml.org/reactive_asco`

[15] `http://repmus.ircam.fr/antescofo`

[7] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone. Esterel: a formal method applied to avionic software development. *Science of Computer Programming*, 36(1):5 – 25, 2000.

[8] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] G. Berry, C. Nicolas, and M. Serrano. Hiphop: a synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, pages 49–56. ACM, 2011.

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

[11] F. Boussinot. Reactive Programming of Cellular Automata. RR 5183, INRIA, 2004.

[12] F. Boussinot. Fairthreads: Mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, 2006.

[13] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. Softw. Eng.*, 22(4), 1996.

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

[15] P. Caspi and M. Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1996.

[16] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[17] C. Deleuze. Programmation réactive en OCaml. *Journal Européen des Systèmes Automatisés*, 43(7-8-9):757–771, 2009.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[19] L. Hazard. Simple. An efficient implementation of Junior.

[20] L. Hazard, J.-F. Susini, and F. Boussinot. The Junior reactive kernel. RR 3732, INRIA, 1999.

[21] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 4.02 Documentation and user's manual*. INRIA, 2014.

[23] L. Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[24] L. Mandel and F. Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Proceedings of Synchronous Languages, Applications, and Programming*. Electronic Notes in Theoretical Computer Science, 2005.

[25] L. Mandel and C. Pasteur. Reactivity of cooperative systems. In *Proceedings of 21st International Static Analysis Symposium*, 2014.

[26] L. Mandel, C. Pasteur, and M. Pouzet. Time Refinement in a Functional Synchronous Language. In *15th International Symposium on Principles and Practice of Declarative Programming*, 2013.

[27] L. Mandel and F. Plateau. Interactive programming of reactive systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems*, 2008.

[28] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, 2005.

[29] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[30] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, 2006.

[31] C. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[32] J. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, 1993.

[33] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. GLONEMO: Global and accurate formal models for the analysis of ad hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks*, 2006.

[34] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12. ACM, 2008.

[35] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[36] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *International Conference on Programming Language, Design and Implementation*, 2000.

[37] M. Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, 1980.